

DIGITAL Visual Fortran

Programmer's Guide

Date: September, 1997

Software Version: DIGITAL Visual Fortran Version 5.0

Operating Systems: Microsoft® Windows NT on Alpha Systems
Microsoft Windows NT on Intel® Systems
Microsoft Windows NT on Windows 95® Systems

Digital Equipment Corporation
Maynard, Massachusetts

Copyright Page

Digital Equipment 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 Digital or an authorized sublicensor.

Copyright © 1997, Digital Equipment Corporation, All Rights Reserved.

Visual Fortran Home Page, Photographs: Copyright © 1997 PhotoDisc, Inc.

Visual Fortran Home Page, Image: CERN, European Laboratory for Particle Physics: ALICE detector on CERN's future accelerator, the LHC, Large Hadron Collider.

AlphaGeneration, DEC, DEC Fortran, DIGITAL, OpenVMS, VAX, VAX FORTRAN, and the DIGITAL logo are trademarks of Digital Equipment Corporation.

ActiveX, Excel, Internet Explorer, Microsoft, MS, Microsoft Developer Studio, Microsoft Press, MS-DOS, NT, PowerPoint, Visual Basic, Visual C++, Visual J++, Win32, Win32s, Windows, Windows 95, and Windows NT are either registered trademarks or trademarks of Microsoft 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 is a trademark of Visual Numerics, Inc.

Intel and Pentium are registered trademarks of Intel Corporation.

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

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Ltd.

All other trademarks and registered trademarks are the property of their respective holders.

Introduction to the Programmer's Guide

The *Programmer's Guide* contains the following information ([this color](#) denotes a link):

- Topics discussing Visual Fortran language elements (including language extensions to the Fortran 90 Standard):
 - [Features of Fortran 95](#)
 - [Features of Fortran 90](#)
 - [Program Structure, Characters, and Source Forms](#)
 - [Declaring and Using Data](#)
 - [Arrays and Pointers](#)
 - [Execution Control](#)
 - [Program Units and Procedures](#)
 - [Files, Devices, and I/O Hardware](#)
 - [Input/Output Editing](#)
 - [Input/Output Statements](#)
 - [General Compiler Directives](#)
- Topics discussing Run-Time Library procedures:
 - [Portability Library](#)
 - [Using QuickWin](#)
 - [Using Dialogs](#)
 - [Drawing Graphics Elements](#)
 - [Using Fonts from the Graphics Library](#)
- Topics discussing how to build programs:
 - [Writing New Code: Design Considerations](#)
 - [Building Programs and Libraries](#)
 - [Advanced Applications](#)
 - [Using COM and Automation Objects](#)
 - [Using the Compiler and Linker from the Command Line](#)
 - [Compiler and Linker Options](#)
 - [Performance: Making Programs Run Faster](#)
 - [Creating Multithread Applications](#)
 - [Programming with Mixed Languages](#)
- Topics discussing programming issues:
 - [Portability](#)
 - [Using National Language Support Routines](#)
 - [The Floating-Point Environment](#)
 - [Handling Run-Time Errors](#)
 - [Converting Unformatted Numeric Data](#)
 - [Using Visual Fortran Tools](#)
 - [PView and WinDiff](#)
- Topics discussing how to use IMSL library routines:
 - [Using the IMSL Mathematical and Statistical Libraries](#)

- Several appendixes discussing the following topics:
 - [Compatibility Information](#)
 - [FORTRAN 77 Syntax](#)
 - [ASCII and Key Code Charts \(WNT and W95\)](#)
 - [Hexadecimal-Binary-Octal-Decimal Conversion](#)
 - [Data Representation](#)
 - [Summary of Language Extensions](#)
- A [Glossary](#)

Note: Visual Fortran contains many extensions to the full ANSI standard language. In this book, the Visual Fortran extensions to the Fortran 90 standard appear in this color.

Programmer's Guide Conventions

This section discusses the following:

- [General typographic conventions](#)
- [Typographic conventions for data types](#)
- [Platform labels](#)

General Typographic Conventions

The *Programmer's Guide* uses the following general typographic conventions. (Text in [this color](#) denotes a link.)

When you see this	Here is what it means
Extensions to Fortran 90	Dark teal type indicates extensions to the Fortran 90 Standard. These extensions may or may not be implemented by other compilers that conform to the language standard.
OUT.TXT, ANOVA.EXE, COPY, LINK, FL32	Uppercase (capital) letters indicate filenames and MS-DOS®-level commands used in the command console. Uppercase is also used for command-line options (unless the application accepts only lowercase).
! Comment line WRITE (*,*) 'Hello & &World'	This kind of type is used for program examples, program output, and error messages within the text. An exclamation point marks the beginning of a comment in sample programs. Continuation lines are indicated by an ampersand (&) after the code at the end of a line to be continued and before the code on the following line.
AUTOMATIC, INTRINSIC, WRITE	Bold capital letters indicate Fortran 90 statements, functions, subroutines, and keywords. Keywords are a required part of statement syntax, unless enclosed in brackets as explained below. In the sentence, "The following steps occur when a DO WHILE statement is executed," the phrase DO WHILE is a Fortran 90 keyword.
other keywords	Bold lowercase letters are used for keywords of other languages. In the sentence, "A Fortran 90 subroutine is the equivalent of a function of type void in C," the word void is a keyword of C.

<i>expression</i>	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.
[options]	Bold square brackets indicate the square bracket characters are required in the syntax.
{ <i>choice1</i> <i>choice2</i> }	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.
Repeating elements...	Three dots following an item indicate that more items having the same form may be entered.
CALL Num (i, *10) ... SUBROUTINE Num (i, *)	A row of three dots in an example indicates that part of the example has been intentionally omitted.
KEY NAMES	<p>Small capital letters are used for the names of keys and key sequences, such as ENTER and CTRL+C.</p> <p>A plus (+) indicates a combination of keys. For example, CTRL+E means to hold down the CTRL key while pressing the E key.</p> <p>The carriage-return key, sometimes marked with a bent arrow, is referred to as ENTER.</p> <p>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).</p> <p>The key names used in this manual correspond to the names on the IBM® Personal Computer keys. Other machines may use different names.</p>

Typographic Conventions for Data Types

The *Programmer's Guide* uses the following typographic conventions for data types:

- Uppercase (capital) letters for data types indicate a specific data type, such as INTEGER(2), REAL, CHARACTER(8), and so on. If a data type appears in uppercase without a kind parameter, it is the default kind for that data type.
- Lowercase letters for data types indicate that any kind of that data type is allowed. For example, if an argument is of type integer, the argument can be any of the following integer data types: INTEGER, INTEGER(1), INTEGER(2), or INTEGER(4).

Elemental procedures accept array arguments as well as scalar arguments. The array arguments and return values must all have the same shape, and the procedure is performed on an element by element basis. For example:

```
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))
```

The following example shows how this book's typographic conventions are used to indicate the syntax of the **PARAMETER** statement:

```
PARAMETER[ ( ]
  vname=const [, vname=const ]...[ ) ]
```

This syntax listing shows that when using the **PARAMETER** statement, you must first enter the word **PARAMETER**. Then you can optionally enter a left parenthesis ((), followed by a *vname* that you specify, followed by an equals sign (=) and a constant value. If you want to specify more *vname=const*, you must enter a comma, followed by another *vname=const* sequence. Because the *vname=const* [, *vname=const*] sequence is followed by three dots (...), you can enter as many of those sequences (a comma, followed by a *vname=const*) as you want. The statement terminates with an optional closing right parenthesis ()). The dark teal brackets ([and]) indicate the parentheses are optional only as 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, Windows 95® on Intel® is a platform.

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

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

Features of Fortran 95

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

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

- The FORALL statement and construct

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

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

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

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

- PURE user-defined procedures

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

DIGITAL Fortran previously provided pure procedures as a language extension.

- ELEMENTAL user-defined procedures

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

- Pointer initialization

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

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

- Derived-type structure default initialization

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

- Automatic deallocation of allocatable arrays

Arrays declared using the `ALLOCATABLE` attribute can now be automatically deallocated in cases where Fortran 90 would have assigned them undefined allocation status. For more information, see [Association Status and Definition](#) under Arrays and Pointers.

DIGITAL Fortran previously provided this feature.

- `CPU_TIME` intrinsic subroutine

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

- Enhanced `CEILING` and `FLOOR` intrinsic functions

`KIND` can now be specified for these intrinsic functions.

- Enhanced `MAXLOC` and `MINLOC` intrinsic functions

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

- Enhanced `SIGN` intrinsic function

The `SIGN` function can now distinguish between positive and negative zero (if the processor is capable of doing so).

- Printing of -0.0

A floating-point value of minus zero (-0.0) can now be printed if the processor can represent it.

- Enhanced `WHERE` construct

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

- Generic identifier allowed in `END INTERFACE` statement

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

- Zero-length formats

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

- Comments allowed in namelist input

Fortran 95 allows comments (beginning with !) in namelist input data. DIGITAL Fortran previously provided this feature as a language extension.

- New obsolescent features

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

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

Deleted Features in Fortran 95

Some language features, considered redundant in FORTRAN 77, are not included in Fortran 95. However, they are still **fully supported** by DIGITAL Fortran:

- ASSIGN and assigned GO TO statements
- Assigned FORMAT specifier
- Branching to an END IF statement from outside its IF block
- H edit descriptor
- PAUSE statement
- Real and double precision DO control variables and DO loop control expressions

DIGITAL Fortran flags these features if you specify the /stand compiler option.

Obsolescent Features in Fortran 95

Some language features, considered redundant in Fortran 90 are identified as obsolescent in Fortran 95.

Fortran 90 offers other methods to achieve the functionality of the following obsolescent features:

- **Alternate returns**

The recommended method to replace this functionality is to 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**

The recommended method to replace this functionality is to use an **IF** statement or construct.

- **Assumed-length character functions**

The recommended methods to replace this functionality is to 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

The recommended method to replace this functionality is to use the Fortran 90 forms of

specifying a length selector in **CHARACTER** declarations.

- **Computed GO TO** statement

The recommended method to replace this functionality is to 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**

The recommended method to replace this functionality is to use an **END DO** statement or a **CONTINUE** statement.

- **Statement functions**

The recommended method to replace this functionality is to use an internal function.

DIGITAL Fortran flags these features if you specify the [/stand](#) compiler option.

Features of Fortran 90

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

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

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

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

Organizing Programs in Fortran

Program units are the fundamental components of a Fortran program. A *program unit* can be a main program, an external subprogram, a module, or a block data program unit. An *executable program* consists of exactly one main program unit and any number of other kinds of program units. For more information on main program units, see [Program Units and Procedures](#).

A *subprogram* can be a function or a subroutine. An *external subprogram* is one which is not contained within a main program, a module, or another subprogram. An *internal subprogram* is one that is contained within a main program or another subprogram. A *module subprogram* is contained in a module, but is not an internal subprogram. Subprograms, also referred to as procedures, are discussed in Program Units and Procedures. The order of statement execution within a program is described in [Statements](#).

A *module* is a file containing definitions accessible to other program units. The definitions can be data object declarations, type definitions, procedure definitions, or procedure interface blocks. Modules can be used to:

- Replace include files or common blocks
- Contain data, definitions, or procedure libraries used by many programs
- Define new data types and operations allowed on that type
- Provide encapsulation (keeping data together with procedures that operate on them)

A module can exist as a separate file, or it can be combined with other distinct program units. For more information on modules, see [Program Units and Procedures](#).

New Fortran 90 statements that affect organization of program units are listed below.

New Statements in Fortran 90	Purpose
CONTAINS	Declares internal procedures
INTERFACE	Defines interface to procedures

MODULE	Defines a module program unit
RECURSIVE	Declares a procedure to be recursive
RESULT	Defines a result variable for a function
USE	Allows access to a module

Scope and Association

Scope defines the extent to which a variable or name is known in a program. The scope of a global name (such as the name of a common block) is an entire program, and it includes any external routines which also have access to that common block. The scope of a local variable name can be as small as a statement or a **DO** loop.

Association refers to ways that extend the scope of a name. Association allows a named entity (such as a data object or a procedure) to be known by different names in the same scoping unit, or by different names in several scoping units. For example, the following code shows a subroutine that defines an argument *a*, and a program that calls the subroutine, passing to it the variable *n* as an argument:

```
PROGRAM SAMPLE
  INTEGER n
  CALL suba(n)
  PRINT *, n
END

SUBROUTINE suba(a)
  INTEGER a
  . . .
END
```

The variable *n* has a scope of the main program only; the variable *a* has a scope of the subroutine only. However, *n* and *a* are said to be associated. It is association that allows *a*'s value from `suba` to be passed to *n* in program `SAMPLE`.

Names and variables can be associated by name association, pointer association, or storage association. Name association is further broken down into argument association, host association, and use association. The concepts of scope and association are covered in [Program Units and Procedures](#).

Program units are composed of separate and distinct *scoping units*. A scoping unit is a part of a program in which a name has a fixed meaning. For example, an internal function is the scoping unit for all dummy arguments it uses internally. A scoping unit can be:

- A derived-type definition
- A procedure interface body, excluding any derived-type definitions and procedure interface bodies it contains
- A program unit or subprogram, excluding derived-type definitions, procedure interface bodies, and subprograms it contains

Scoping units can contain other scoping units. In this case, the surrounding scoping unit is called the *host scoping unit*. For more information on scoping units, as well as program structure and using modules, see [Program Units and Procedures](#).

Procedures in Fortran 90

A *procedure* contains a sequence of computations, and can be invoked during program execution. A procedure can be either a function or a subroutine. Functions and subroutines both have the same *characteristics* in Fortran 90. The characteristics of a procedure include:

- Classification of the procedure as either a function or a subroutine
- The characteristics of its arguments (such as attributes, type, type parameters, shape, and intent)
- The characteristics of the result value, for functions

A *function* is invoked by reference within an expression, or by a defined operation within an expression. The function computes a value which is then used in evaluating the expression. The variable that returns the value of a function is the *result variable*. A *subroutine* is invoked with a **CALL** statement or a defined assignment statement.

Arguments can be optional, or can be specified as keywords. Fortran 90 allows recursive procedures.

Procedures are classified as external, module, or internal. An *external procedure* is defined outside of the main program unit or any of its modules or subprograms. External procedures can be invoked by a main program or by any of its procedures. A *module procedure* is a procedure defined by a module subprogram. An *internal procedure* is a procedure defined as an internal subprogram. The program that contains the internal procedure is called the *host*. An internal procedure is local to its host, meaning it is accessible to the host and other internal procedures, but not accessible elsewhere. For detailed information on procedures, see [Program Units and Procedures](#).

A *procedure interface block* describes the interface to a set of procedures. It can present a single generic name (or may define an operator or an assignment) that invokes one of the procedures.

For a list of new intrinsics, see: [New Intrinsic Procedures](#)

New Intrinsic Procedures

Some intrinsic procedures that are new to Fortran 90 were implemented in earlier versions of DIGITAL Fortran. The following tables list new Fortran 90 intrinsic procedures. See the *Reference* for descriptions of each procedure.

New Intrinsic Subroutines in Fortran 90		
DATE_AND_TIME	MVBITS	RANDOM_NUMBER
RANDOM_SEED	SYSTEM_CLOCK	

New Intrinsic Functions in Fortran 90		
ADJUSTL	ADJUSTR	ALL
ANY	ASSOCIATED	BIT_SIZE
BTEST	CEILING	COUNT
DOT_PRODUCT	FLOOR	IAND
IACHAR	IBCLR	IBITS
IBSET	IEOR	IOR

ISHFT	ISHFTC	KIND
LEN_TRIM	MATMUL	MAXLOC
MAXVAL	MINLOC	MINVAL
NOT	PRESENT	PRODUCT
REPEAT	SCAN	SELECTED_INT_KIND
SELECTED_REAL_KIND	SUM	TRANSFER
TRANSPOSE	TRIM	VERIFY

New Numeric Functions in Fortran 90		
DIGITS	EPSILON	FRACTION
RADIX	RANGE	RRSPACING
SCALE	SET_EXPONENT	SPACING

Controlling Program Flow

A *block* construct is a sequence of statements treated as an integral unit, bounded by statements such as **IF** or **DO**, and their corresponding **END IF** or **END DO** statements. Block constructs control the flow of execution of statements in a Fortran program. The following block control constructs are part of standard Fortran 90:

- **CASE** construct
- Block **DO** construct, **DO WHILE**, or iterative clause in a **DO** construct
- **CYCLE** statement in a **DO** construct
- **EXIT** statement in a **DO** construct

Fortran allows block control constructs to be named, as in the following example:

```
PROOF_DONE: DO
  READ (IUN, '(1X,G14.7)', IOSTAT = IOS) X
  IF (IOSTAT .NE.0) EXIT PROOF_DONE
  IF (X .GT. MAX) CYCLE PROOF_DONE
  CALL SUBA(X)
END DO PROOF_DONE
```

The **DO** construct in this example is named `proof_done`. Although the name is not required, it marks the start and end of the block. In the example, the **EXIT** and **CYCLE** statements also name the block construct. Block names can be particularly useful for controlling execution in nested loops. Named block constructs are particularly useful for nested constructs. Block control constructs are discussed in [Execution Control](#).

Data Concepts

The data environment is defined by nonexecutable statements. A *data type* is a named category of data characterized by a set of values, together with a way to denote the values. A data type also has a set of operations that interpret and manipulate the values. Data types can be either intrinsic or derived.

An *intrinsic type* is one that is defined in the ANSI Standard for Fortran 90 and is always accessible. Intrinsic types are INTEGER, REAL, COMPLEX, CHARACTER, and LOGICAL. A *derived type* is

a data type defined in your program, composed of smaller units which can be either intrinsic or derived types. A variable declared as a derived type is known as a *structure*. Structures are defined and declared with **TYPE** statements. *Derived type components* are the individual elements that make up a derived type.

Note: The statement **STRUCTURE ... END STRUCTURE** is a DIGITAL extension to FORTRAN 77. It is equivalent to the Fortran 90 derived type declaration that uses the **TYPE ... END TYPE** statement. When the word "structure" appears in lowercase and is not in bold type, it refers to the Fortran 90 concept, not the Visual Fortran statement.

A variable declared as a derived type can be composed of any combination of the intrinsic types, arrays, as well as other derived types. Earlier versions of DIGITAL Fortran offered the **STRUCTURE** statement to declare and define compound variable types. It is still supported in Visual Fortran and described in the *Reference*. However, new code should use the Fortran 90 **TYPE** statement instead. For details on defining new data types, see [Declaring and Using Data](#).

Attributes and kind or length type parameters further describe each intrinsic data type. The kind type parameter **KIND** can be used with all intrinsic types and specifies the memory storage in bytes. Each data type has a default **KIND**. The length type parameter **LEN** indicates the length of string. Attributes such as **ALLOCATABLE**, **POINTER** and **SEQUENCE** specify other properties of the data type.

Intrinsic numeric functions such as **AINT(A)** or **CMPLX(X)** also accept the **KIND** parameter to qualify an argument. For example, **AINT (A, KIND=1)** specifies a result that is 1 byte long.

Scalar data is any data that is not an array. Scalars can be either intrinsic or derived types. A structure is scalar even if it has arrays as components.

A *data entity* is a data object, the result of the evaluation of an expression, or a function result. A data entity always has a type. The term *data object* refers to constants, variables, and subobjects. A *subobject* is a part of a named object that can be referenced and defined independently. Examples of subobjects are array elements and structure components. A constant subobject is a portion of a constant. The referenced part may depend on the value of a variable. For example:

```
CHARACTER (LEN = 10), PARAMETER :: DIGITS = '0123456789'
CHARACTER (LEN = 1) :: DIGIT
INTEGER = I
. . .
DIGIT = DIGITS(I:I)
```

DIGITS is a named constant and **DIGITS (I:I)** designates a constant subobject of **DIGITS**.

Intrinsic type attributes that are new with Fortran 90 are listed in the following table. You can use them either as statements or as attributes in type declarations. More information on data types is in [Declaring and Using Data](#).

New Type Attributes in Fortran 90	
ALLOCATABLE	DIMENSION
EXTERNAL	INTENT (IN, OUT, INOUT)
INTRINSIC	PARAMETER
OPTIONAL	PRIVATE

POINTER	SAVE
PUBLIC	TARGET
SEQUENCE	

Array Operations in Fortran 90

An *array* is a set of scalar data, all of the same type and type parameters, whose elements are arranged in a rectangular pattern. An *array element*, one of the individual elements in the array, is a scalar. Arrays have *extent* (number of elements), *rank* (number of dimensions), and *size* (total number of elements).

Fortran 90 includes many enhancements to the treatment of arrays. The main enhancements are:

- Entire arrays can be treated as a single object
- Arrays can be zero-size
- The result of a function can be an array
- Arrays can have the POINTER attribute

Dynamic storage allocation is a feature of standard ANSI Fortran 90. [Arrays and Pointers](#) discusses arrays and dynamic storage allocation.

New Array Manipulation Functions		
CSHIFT	EOSHIFT	LBOUND
MERGE	PACK	RESHAPE
SHAPE	SIZE	SPREAD
UBOUND	UNPACK	

Pointers

Pointers are variables with a POINTER attribute. Pointers allow a variable to be known by several names, or allow one variable to point alternately to different objects. The POINTER attribute allows association between a data object and a *target*. Pointers also allow data to be accessed and processed dynamically. For more information on pointers, see [Declaring and Using Data](#).

Note: A pointer in Fortran 90 is not the same as a pointer in C. Information on pointers is in [Arrays and Pointers](#).

Input and Output Facilities

Fortran 90 provides additional facilities for input and output. New clauses for the **OPEN** and **INQUIRE** statements let you specify any of the following:

- File permissions
- Status of a file on opening or closing
- Whether input records are to be padded with blanks
- Delimiting character for list-directed or namelist-directed data

READ and **WRITE** statements offer connection specifiers that allow formatted sequential data

transfer to be either advancing or non-advancing. Other specifiers let you identify where in the program to transfer control when an end-of-record is encountered.

The **NAMELIST** statement declares a group name for a set of variables so they can be read or written with a single namelist-directed **READ** or **WRITE** statement. Although this is new to standard ANSI Fortran 90, it was implemented in earlier versions of DIGITAL Fortran.

Input and output issues are discussed in [Input/Output Statements](#); [Files, Devices and I/O Hardware](#); and [Input/Output Editing](#).

Source Form

Program source code can be written in two formats: free form or fixed form. Fixed form conforms exactly to the FORTRAN 77 standard. Free form is allowed in Fortran 90 and has the following characteristics:

- Default line length is 132 characters
- All characters between an exclamation point (!) and the end of the line are comments; comments can begin anywhere on a line
- Blanks are significant in some cases
- Position on the line has no special meaning
- Multiple statements can appear on one line, separated by semicolons

You can write code that satisfies rules for both fixed and free form. Program units with different source formats can be compiled in the same project. For information on source code format, see [Source Forms](#).

Syntax and Usage

The ANSI standard for Fortran 90 requires that compilers detect and report syntax and usage that has been designated obsolescent. Compilers must also detect and report non-standard syntax. You have the option to turn off error reporting for both non-standard syntax and obsolescent features.

Obsolescent Features in Fortran 90

Although no processor-independent elements of previous versions of Fortran have been omitted from Fortran 90, the ANSI standard has labeled certain features obsolescent since they have been replaced by newer features. An obsolescent feature is one that is redundant, but is used frequently and still supported in this version. These are:

- Nonblock **DO** construct
- **ASSIGN** statement
- Assigned **GO TO** statement
- Assigned **FORMAT** specifiers
- Arithmetic **IF** statement
- **PAUSE** statement
- Alternate **RETURN**

- **H** edit descriptors

DIGITAL Fortran flags these features if you specify the [/stand](#) compiler option.

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](#)

A program can contain any number of **INCLUDE** lines. An **INCLUDE** statement causes the compiler to replace the line with source text from a separate file. For more information, see the [INCLUDE statement](#) in the *Reference*.

Program Structure

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

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

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

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

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

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

The following sections discuss [Statements](#), [Names](#), and [Keywords](#).

Statements

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

Order of Statements in a Program Unit

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

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

Required Order of Statements

Comment Lines, INCLUDE Statements, and Directives	OPTIONS Statements		
	PROGRAM, FUNCTION, SUBROUTINE, MODULE, or BLOCK DATA Statement		
	USE Statements		
	NAMELIST, FORMAT, and ENTRY Statements	IMPLICIT NONE Statements	
		PARAMETER Statements	IMPLICIT Statements
		PARAMETER and DATA Statements	Derived-Type Definitions, Interface Blocks, Type Declaration Statements, Statement Function Statements, and Specification Statements
		DATA Statements	Executable Statements
	CONTAINS Statement		
	Internal Subprograms or Module Subprograms		
	END Statement		

ZK-6516A-GE

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

Statements Restricted in Scoping Units

Scoping Unit	Restricted Statements
Main program	ENTRY and RETURN statements
Module [1]	ENTRY , FORMAT , OPTIONAL , and INTENT statements, statement functions, and executable statements
Block data program unit	CONTAINS , ENTRY , and FORMAT statements, interface blocks, statement functions, and executable statements
Internal subprogram	CONTAINS and ENTRY statements
Interface body	CONTAINS , DATA , ENTRY , SAVE , and FORMAT statements, statement functions, and executable statements
[1] The scoping unit of a module does not include any module subprograms that the module contains.	

Names

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

A name can contain letters, digits, an underscore (`_`), and the dollar sign (\$) special character. A name can contain up to 31 characters; the first character must be a letter.

Note Be careful when defining names that contain dollar signs.

On OpenVMS systems, naming conventions reserve names containing dollar signs to those created by DIGITAL. On DIGITAL UNIX, Windows NT, and Windows 95 systems, a dollarsign 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 <code>_</code> or <code>\$</code> .
_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 allow you to 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 *Reference* show the dummy keywords you can use for each Fortran procedure.

Character Sets

DIGITAL Fortran supports the following characters:

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

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

- Other printable characters

Printable characters include the tab character (09 hex), ASCII characters with codes in the range 20(hex) through 7E(hex), [and characters in certain special character sets](#).

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

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

For more detailed information on character sets and default character types, see [Declaring and Using Data and Using National Language Support Routines](#). For more information on the ASCII character set, see [ASCII and Key Code Charts](#).

Source Forms

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

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

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

- Comment indicator

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

An all blank line is also a comment line.

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

For more information, see comment indicators in [free source form](#), or [fixed and tab source forms](#).

- Statement separator

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

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

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

- Continuation indicator

A statement can be continued for more than one line by placing a continuation indicator on the line. [DIGITAL Fortran allows up to 511 continuation lines in a source program.](#)

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

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

For more information, see continuation indicators in [free source form](#), or [fixed and tab source forms](#).

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

Indicators in Source Forms

Source Item	Indicator [1]	Source Form	Position
Comment	!	All forms	Anywhere in source code
Comment line	!	Free	At the beginning of the source line
	!, C, or *	Fixed	In column 1
		Tab	In column 1
Continuation line [2]	&	Free	At the end of the source line
	Any character except zero or blank	Fixed	In column 6
	Any digit except zero	Tab	After the first tab
Statement separator	;	All forms	Between statements on the same line
Statement label	1 to 5 decimal digits	Free	Before a statement
		Fixed	In columns 1 through 5
		Tab	Before the first tab
A debugging statement [3]	D	Fixed	In column 1
		Tab	In column 1
[1] If the character appears in a Hollerith or character constant, it is not an indicator and is ignored. [2] For all forms, up to 511 continuation lines are allowed. [3] Fixed and tab forms only.			

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.

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.

Free Source Form

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

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

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

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

The blanks are required after `INTEGER`, `TO`, `20`, and `DO`.

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

Optional Blanks	Required Blanks
BLOCK DATA	CASE DEFAULT
DOUBLE COMPLEX	DO WHILE
DOUBLE PRECISION	IMPLICIT <i>type- specifier</i>

ELSE IF	IMPLICIT NONE
END BLOCK DATA	INTERFACE ASSIGNMENT
END DO	INTERFACE OPERATOR
END FILE	MODULE PROCEDURE
END FORALL	RECURSIVE FUNCTION
END FUNCTION	RECURSIVE SUBROUTINE
END IF	RECURSIVE <i>type-specifier</i> FUNCTION
END INTERFACE	<i>type-specifier</i> FUNCTION
END MODULE	<i>type-specifier</i> RECURSIVE FUNCTION
END PROGRAM	
END SELECT	
END SUBROUTINE	
END TYPE	
END WHERE	
GO TO	
IN OUT	
SELECT CASE	

For information on statement separators (;) in all forms, see [Source Forms](#).

Comment Indicator

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

Continuation Indicator

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

The following shows a continued statement:

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

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

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

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

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

If you indent the continuation line of a character constant, an ampersand must be the first character of the continued line; otherwise, the blanks at the beginning of the continuation line will be included as part of the character constant. For example:

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

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

For details on the general rules for all source forms, see [Source Code Format](#).

Fixed and Tab Source Forms

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

In fixed [and tab](#) source forms, there are restrictions on where a statement can appear within a line.

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

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

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

Comment Indicator

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

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

Continuation Indicator

In fixed [and tab](#) source forms, a continuation line is indicated by one of the following:

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

The compiler considers the characters following the continuation indicator to be part of the previous line. Although Fortran 90 permits up to 19 continuation lines in a fixed-form program, [DIGITAL Fortran allows up to 511 continuation lines.](#)

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

The statement label field of a continuation line must be blank ([except in the case of a debugging statement](#)).

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

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

Use this same method when initializing data with long character [or Hollerith](#) constants. For example:

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

[Hollerith constants must be converted to character constants before using the concatenation method of line continuation.](#)

Debugging Statement Indicator

In fixed [and tab](#) source forms, the statement label field can contain a statement label, a comment indicator, [or a debugging statement indicator](#).

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

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

By default, the compiler treats debugging statements as comments. However, you can specify the [/d lines](#) option to force the compiler to treat debugging statements as source text to be compiled.

The following sections discuss [Fixed-format lines](#) and [Tab-format lines](#).

For details on the general rules for all source forms, see [Source Code Format](#).

Fixed-Format Lines

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

The column positions for each field follow:

Field	Column
-------	--------

Statement label	1 through 5
Continuation indicator	6
Statement	7 through 72 (or 132 with the /extend source compiler option)
Sequence number	73 through 80

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

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

For details on the general rules for all source forms, see [Source Code Format](#).

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

Format using TAB Character

Character-per-Column Format

C	[TAB]	FIRST	VALUE																
10	[TAB]	I = J + 5 * K +																	
[TAB]		1 L * M																	
[TAB]		IVAL = I + 2																	

ZK-0614-GE

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

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

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

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

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

For details on the general rules for all source forms, see [Source Code Format](#).

For details on the general rules for fixed and tab source forms, see [Fixed and Tab Source Forms](#).

Source Code Useable for All Forms

Source code can be written so that it is useable for all source forms (free, fixed, or tab).

The following restrictions must be followed:

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

The following example is valid for all source forms:

Column:

12345678...

73

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

Declaring and Using Data

This chapter discusses [intrinsic](#) and [derived](#) data types, their [attributes](#) and [storage](#). It also discusses [expressions](#) and [assignments](#) that manipulate data, [type declaration statements](#), [data initialization](#), and [defining variables](#).

Overview of Data Types

A data type has four properties:

- A name. The names of intrinsic data types are predefined by the Fortran language; names of derived types are set in type definition statements.
- A set of values. Each data type has a valid set of values. The logical type is limited to only two values: **.TRUE.** and **.FALSE.**. Integer and real types have allowed ranges of values. Complex and derived types have sets of values that are combinations of the values of their individual components.
- A way of representing constant values for the type.
- A set of operations to manipulate and interpret the values. A variable's data type determines the operations that can be used to manipulate it. You can augment the intrinsic set with operations and operators that you define. This is discussed in [Defined Operators and Expressions](#).

Intrinsic data types are defined by the language, while derived data types are specific to the application: you define them in your program. The Intrinsic Data Types section lists the name, values, and way of representing each intrinsic data type. The valid operations for each data type are covered in [Intrinsic Operators](#).

The [Derived Types](#) section covers declaring a derived type, assigning values, and attributes.

Data objects are declared using the name of a data type. A data object is a constant, a variable, or part of a constant or variable. Once you have defined a derived type, you can declare objects to be of that type.

A *subobject* is a part of a named object that you refer to and define independently of the other portions. A subobject can be an array element, one element of a structure, or a portion of a character string. Arrays are more fully described in [Arrays and Pointers](#).

Intrinsic Data Types

The intrinsic data types are:

- Integer
INTEGER, INTEGER(1), INTEGER(2), INTEGER(4), and INTEGER(8) (on Alpha only)
- Real
REAL, DOUBLE PRECISION, REAL(4), and REAL(8)
- Complex
COMPLEX, COMPLEX(4), **DOUBLE COMPLEX**, and COMPLEX(8)

- Character
CHARACTER[*n,] where *n* is the length of the string
- Logical
LOGICAL, LOGICAL(1), LOGICAL(2), LOGICAL(4), and LOGICAL(8) (on Alphaonly)

Each numeric type includes a zero value considered to be neither positive nor negative. The value of a signed zero is the same as that of an unsigned zero.

Each intrinsic type also has a KIND parameter, which further describes the data type. Used with numeric types, KIND describes precision and decimal exponent range. Each data type has a default KIND. There is only one KIND for character types. The default KIND parameters are listed in the following table. You can find valid KIND values for each intrinsic type in the sections that describe that data type.

Memory Requirements and Default Kinds			
Type	Kind	Bytes	Notes
BYTE	1	1	Same as INTEGER(1).
INTEGER	2, 4, or 8	2, 4, or 8	Depending on default integer, INTEGER can have two, four, or eight bytes. The default allocation is four bytes.
INTEGER(1)	1	1	
INTEGER(2)	2	2	
INTEGER(4)	4	4	
INTEGER(8)	8	8	Alpha only.
REAL	4 or 8	4 or 8	Depending on default real, REAL can have four or eight bytes. The default allocation is four bytes.
REAL(4)	4	4	
DOUBLE PRECISION	8	8	Same as REAL(8).
REAL(8)	8	8	
COMPLEX	4 or 8	8 or 16	Depending on default real, COMPLEX can have eight or sixteen bytes. The default allocation is eight bytes.
COMPLEX(4)	4	8	
DOUBLE COMPLEX	8	16	Same as COMPLEX(8).
COMPLEX(8)	8	16	
CHARACTER	1	1	CHARACTER and CHARACTER(1) are the same. (1) is the KIND parameter, not the string length.
CHARACTER* <i>len</i>	1	<i>len</i>	<i>len</i> is the string length; 1 to 65535 on Intel processors, 1 to 2**31-1 on Alpha processors.
LOGICAL	2, 4, or 8	2, 4, or 8	Depending on default integer, LOGICAL can have two, four, or eight bytes. The default allocation is four bytes.
LOGICAL(1)	1	1	
LOGICAL(2)	2	2	
LOGICAL(4)	4	4	
LOGICAL(8)	8	8	Alpha only.

Integer Data Type

The **INTEGER** statement specifies that the named variables and functions are integer type. Integers can be specified as **INTEGER**, **INTEGER(1)**, **INTEGER(2)**, **INTEGER(4)**, or **INTEGER(8)** (on Alpha only). You can also use the alternate specifications of **INTEGER*1**, **INTEGER*2**, **INTEGER*4**, or **INTEGER*8**. You can change the result of a default specification by using the /integer size:size compiler option or the INTEGER compiler directive.

The syntax for the **INTEGER** type declaration statement is:

```
INTEGER [ ( [ KIND = ] kind-value ) ] [ [, attribute-list ] :: ] entity-list
```

The numbers 1, 2, 4, or 8 in parentheses represent the *kind-value*. The numbers following the asterisk (*) of the alternate type specifications are length indicators, not *kind-values*.

If the kind type parameter is not included, the default *kind-value* is 4. 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.

An integer value is a binary representation of the corresponding integer. The following table shows the different types of integers, how many bytes of memory each type occupies, and the range of each type.

Table: Integer Value Ranges		
Data type (kind)	Bytes	Range*
INTEGER(1)	1	Signed: -128 to 127 Unsigned: 0 to 255
INTEGER(2)	2	Signed: -32,768 to 32,767 Unsigned: 0 to 65535
INTEGER(4)	4	Signed: -2,147,483,648 to 2,147,483,647 Unsigned: 0 to 4,294,967,295
INTEGER(8) (Alpha only)	8	Signed: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

*: Unsigned ranges are allowed for assignment to variables of this type, but the data type is treated as signed in arithmetic operations.

The examples which follow 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
```

For more information on the integer data type, see [Integer Constants](#).

Integer Constants

An integer constant is represented by an optional plus (+) or minus (-) sign preceding the digits, followed by an optional integer kind specifier.

Examples of signed and unsigned integer literal constants are:

```
123      +123      -123      12_2
-1234567890_4  12_SHORT
```

where SHORT is a named integer constant whose value is an integer kind.

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:

[sign] [[base] #] constant

The optional *sign* is a plus or minus sign. The *base* can be any constant from 2 through 36. If *base* is omitted but *#* is specified, the integer is interpreted in base 16. If both *base* and *#* are omitted, the integer is interpreted in base 10. For bases 11 through 36, the letters A through Z represent numbers greater than 9. For base 36, for example, 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.

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

```
I = 2#11111001111001111001111
m = 7#45644664
J = +8#17171717
K = #3CF3CF
n = +17#2DE110
L = 3994575
index = 36#2DM8F
```

Integer constants of any base must be in the ranges specified in [this table](#). However, for numbers with a radix other than 10, the compiler reads out-of-range numbers up to 232. They are interpreted as the negative numbers with the corresponding internal representation. For example, 16#FFFFFFFF results in an arithmetic value of -1.

In a **DATA** statement, you can use an unsigned binary, octal, or hexadecimal literal constant if it corresponds to an integer scalar variable. The respective constants are entered in apostrophes (') or quotation marks (") following the character designator. The following are example cases:

```
INTEGER i, j, k
DATA i /B'110010' /      ! binary value of decimal 50
```


If the kind specifier is absent, the default *kind-value* is single-precision real (REAL(4)). You can use the **DOUBLE PRECISION** statement to declare double-precision real (8 byte). [You can change the result of a default specification by using the `/real_size:size` compiler option or the `REAL` compiler directive.](#)

If your program names DOUBLE as a constant having the value of the kind parameter of the double-precision real type (DOUBLE = 8), the statement **REAL** (DOUBLE) can also declare an object to be double precision. REAL(KIND(0.0)) is also equivalent to REAL(4) (or REAL(8) if the `/real_size:64` option is in effect).

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

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

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

An attribute-oriented example is:

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

For more information on real data types, see [Real Constants](#).

Real Constants

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

Syntax

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

$$[s]n[n\dots][_k]$$

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

$$[s]n[n\dots]E[s]nn\dots[_k]$$

$$[s]n[n\dots]D[s]nn\dots$$

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)** or 8 for **REAL(8)**. It must be preceded by an underscore (_).

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

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

A real constant with no exponent part is (by default) a single-precision (**REAL(4)**) constant.

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.

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 or D) 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.

The following table shows the different types of real values, how many bytes of memory each type occupies, the decimal precision, and the range of each type.

Table: Real Value Ranges			
Data type (kind)	Bytes	Precision	Range
REAL(4)	4	leftmost 7 digits	Negative numbers from approximately -3.40282347E+38 to -1.17549435E-38 Zero Positive numbers from approximately +1.17549435E-38 to +3.40282347E+38
REAL(8) or DOUBLE PRECISION	8	leftmost 15 digits	Negative numbers from approximately -1.7976931348623158D+308 to -2.2250738585072013D-308 Zero

		Positive numbers from approximately +2.2250738585072013D-308 to +1.7976931348623158D+308
--	--	--

REAL(4) Constants

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

See Also: [DOUBLE PRECISION](#), *Programmer's Guide: Real Data Type*, [The Floating-Point Environment](#).

Examples

The following double-precision real constants all represent fifty-two one-thousandths (52/1000 or .052):

```
5.2D-2    +.00052E+2_8    .052D0 52D-3    .052_8    52.000E-3_dbl
```

In the preceding example, the named integer constant `dbl` must have been previously defined to specify double precision (`PARAMETER dbl = 8`).

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

Valid

- 3.14159
- 3.14159_4
- 621712._4
- .00127
- +5.0E3
- 2E-3_4

Explanation

- | | |
|---|---|
| <ul style="list-style-type: none"> Invalid 1,234,567. 325E-47 -47.E47 625._6 100 \$25.00 | <ul style="list-style-type: none"> Commas not allowed. Too small for REAL; this is a valid DOUBLE PRECISION constant. Too large for REAL; this is a valid DOUBLE PRECISION constant. 6 is not a valid kind for reals. Decimal point missing; this is a valid integer constant. Special character not allowed. |
|---|---|

Complex Data Type

The **COMPLEX** or **COMPLEX(4)** (**COMPLEX*8**) data type is an ordered pair of single-precision real numbers. The **DOUBLE COMPLEX** or **COMPLEX(8)** (**COMPLEX*16**) data type is an ordered pair of double-precision real numbers. For example:

```
COMPLEX(4) c
c = (3.0, 4.0)
```

The first number in the pair is the real part of a complex number, and the second number is the

imaginary part. Both the real and imaginary components of a complex number are of the same kind. Single-precision complex numbers occupy 8 bytes of memory. Double-precision complex numbers occupy 16 bytes of memory.

Because a complex number is represented as an ordered pair of real or double precision values, you must treat a complex number as two real or double-precision numbers in formatted I/O. If you write a complex number with list-directed I/O, the output is the pair of values with parentheses around them and a comma between them. Any other special formatting must be handled in the **FORMAT** statement.

The syntax for the **COMPLEX** type declaration is:

```
COMPLEX [ ( [ KIND = ] kind-value ) ] [ [ , attribute-list ] :: ] entity-list
```

An entity-oriented example is:

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

An attribute-oriented example is:

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

For more information on complex data types, see [Complex Constants](#).

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.

A complex constant has the following form:

```
( c , c )
```

c

Is as follows:

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

The following are examples of complex constants:

```
( 1.0 , -1.0 )      ( 4 , 4.2E3 )      ( 5.0_8 , 8.3E9_8 )
```

For rules on forming real constants, see [Real Constants](#).

See also [COMPLEX](#) in the *Reference*.

Character Data Type

The character type has a set of values composed of characters making up a string. The length of a

string is the number of characters in the string, which is the declared length for a static string or the actual length of a dynamic string. Character variables occupy 1 byte of memory for each character and are assigned to continuous bytes, independent of word boundaries. The type specifier is the keyword **CHARACTER**, as in the following example:

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

The syntax for the **CHARACTER** type declaration statement is:

```
CHARACTER [ type-parameter ] [ [ , attribute-list ] :: ] variable-name
```

The *type-parameter* can take one of two forms:

```
[ LEN = ] value  
*character-length
```

The *value* and *character-length* can be specified either as a constant value or as a named constant whose value has been defined. The *character-length* can also be expressed as an (*). Sample character declarations are shown in the examples that follow.

Attributes are described in the section entitled [Data Attributes](#).

You do not need to specify a KIND parameter for a character data type since there is only one character kind. Several Multi-Byte Character Set (MBCS) functions are available to manipulate special non-English characters. These are described in [Using National Language Support Routines](#).

You cannot allocate character strings unless they have a length fixed at zero. An array of character strings is not the same as a single character string. You can declare an array of character strings, all of the same length. Arrays of fixed-length strings can be used and accessed, allocated and deallocated, like any other array as described in [Arrays and Pointers](#).

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
```

If you are declaring a character-type statement function, or a dummy argument of a statement function, you must use a constant value instead of the asterisk (*) length indicator. An asterisk can be used to indicate length only in the following cases:

- To declare the dummy argument of a procedure. The dummy argument takes the length of the associated actual argument.
- To declare a named constant. The length is that of the constant value.
- To declare the result variable for an external function. In this case, any program unit invoking

the function must declare the function name with a length type parameter value other than an asterisk (*), or access the declaration by host or use association. When the function is invoked, the length of the result variable in the function is the value specified in the type parameter.

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

```
CHARACTER*32 string
CHARACTER string*32
CHARACTER string*(const+5)
```

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

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

For more information on the character data type, see:

- [Substrings](#)
- [Character Constants](#)
- [C Strings](#)
- [Converting Characters to Numeric Data Types](#)

Substrings

Even though a single character string is a scalar, you can still specify and use a part of the string, called a *substring*. A substring is a contiguous portion of a character string. Substrings are defined and accessed with notation similar to accessing sections of arrays. Instead of a first and second dimension bound, however, you specify a start and end character position, separated by a colon (:).

The syntax for expressing a character substring is:

```
[first-position] : [end-position]
```

The default start position is one; the default end position is the length of the string. If the given start position exceeds the end position, the substring has length zero (0). The following examples demonstrate substrings:

```
CHARACTER(10) string
CHARACTER(5) substring
CHARACTER(1) char
string = "Jane Doe "
substring = string(:5)      ! returns 'Jane '
substring = string(6:)     ! returns 'Doe '
substring = string(3:7)    ! returns 'ne Do'
substring = string(6:6)    ! returns 'D '
n = 7
char = 'abcdefghijkl'(n:n) ! returns 'g', the nth (7th) character
                        ! of the string constant
```

The strings and the substrings taken from them can be arrays, in which case the way of specifying arrays and array sections is the same, and substring specifiers follow the array specifiers. For example:

```
CHARACTER(8) A(5)          ! A five-element array of strings
                          ! containing 8 characters.
CHARACTER(3) B(5)         ! A five-element array of strings
                          ! containing 3 characters.
```

```

CHARACTER(5) substring
substring = A(3)(2:6) ! Returns the 2nd through 6th
                       ! characters of A's third element.
B(1:2) = A(4:5)(1:3) ! Puts the 1st through 3rd
                     ! characters of A's 4th element
                     ! into B's 1st element, and puts
                     ! the 1st through 3rd characters
                     ! of A's 5th element
                     ! into B's second element.
    
```

For more information on arrays, see [Arrays and Pointers](#).

Character Constants

A character constant is a sequence of ASCII characters enclosed in a pair of apostrophes (') or quotation marks ("). The string delimiters are not stored with the string. Nonprintable characters can be included by using the methods discussed in [Special Characters](#).

The character constant can be immediately followed by a C to specify a C-type character string.

A character constant has the following form:

```
' [ char ] ... ' [ C ]
```

char

A character from the ASCII set in fixed-form source or a graphic character from the character set in free-form source.

C

The C string specifier.

For compatibility with older Fortran code, Visual Fortran also supports the Hollerith notation for assigning the value of a character string.

To represent an apostrophe within a string delimited by apostrophes, specify two consecutive apostrophes with no blanks between them. To represent a quotation mark within a string delimited by quotation marks, specify two consecutive quotation marks with no blanks between them.

A zero-length character string is specified by two consecutive apostrophes or quotation marks with no blanks between them.

Blank characters and tab characters are permitted in character constants and are significant. The case of alphabetic characters is also significant. [You can use C strings to define strings with nonprintable characters.](#)

The length of a character constant is equal to the number of characters between the delimiters. (A pair of apostrophes in a string delimited by apostrophes counts as a single character. A pair of quotation marks in a string delimited by quotation marks counts as a single character.)

Some sample character constants are shown in the following table.

String	Constant
'String'	String

'1234!@#\$',	1234!@#\$,
'Blanks count'	Blanks count
' '' ''	' ''
'Case Is Significant'	Case Is Significant
" ' ' "	' ''
" "Double" " quotes count as one"	"Double" quotes count as one

If a character constant extends across a line boundary, its value includes any blanks remaining in the line. This result can be avoided by breaking the line as shown:

```
Heading (secondcolumn) = 'Acceleration of particles '//
&'from Group A'
```

The same statement in free format would be as follows:

```
Heading (secondcolumn) = 'Acceleration of particles ' &
& '// 'from Group A'
```

C Strings

String values in the C language are terminated with null characters (**CHAR(0)**), and may contain nonprintable characters (such as newline and backspace). Nonprintable characters are specified using the backslash (\) as an escape character, followed by a single character indicating the nonprintable character desired. This type of string is specified in Visual Fortran 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 valid escape sequences. If a string contains an escape sequence that isn't in this table (such as \z), the backslash is ignored.

Sequence	Character
\a	Bell
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\xhh	Hexadecimal bit pattern
\\	Backslash
\ooo	Octal bit pattern

A C string must also be a valid Fortran string. Therefore, if the string is delimited by apostrophes ('), all apostrophes in the string itself must be represented by a pair of apostrophes. The escape sequence \'string causes a compiler syntax error because Fortran interprets the apostrophe as the end of a string. The correct form is \'string. If the string is delimited with quotation marks ("), a quotation mark can be entered by using a pair of quotation marks. C strings and ordinary strings differ only in how you specify the value of the string. The compiler treats them identically.

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 `o` octal digit must be in the range 0 - 7, and each `h` hexadecimal digit must be in the range 0 - F. For example, the C strings `'\010C` and `'\x08'C` both represent a backspace character followed by a null character.

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

Converting Characters to Numeric Data Types

You can use internal **READ** and **WRITE** statements to make type conversions from a string to a numeric data type (such as integer, complex, or real) or from a numeric data type to a string. The following example converts a string to a real number:

```
PROGRAM testget
REAL x
CHARACTER (50) text
text = "12345.67"
READ (text,*) x
WRITE (*,*) x
END
```

More examples are in [Using Dialogs](#).

Logical Data Type

The **LOGICAL** statement specifies that the named variables and functions are of logical type. Logical variables can be specified as **LOGICAL**, **LOGICAL(1)**, **LOGICAL(2)**, **LOGICAL(4)**, or **LOGICAL(8)** (on Alpha only). You can also use the alternate specifications of **LOGICAL*1**, **LOGICAL*2**, **LOGICAL*4**, or **LOGICAL*8**. If the kind specifier is absent, the default is **KIND(4)**, and the variables or functions are default logical type.

The syntax of the **LOGICAL** type declaration statement is:

```
LOGICAL [ ( [ KIND = ] kind-value ) ] [ [ , attribute-list ] :: ] entity-list
```

An entity-oriented example is:

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

An attribute-oriented example is:

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

Logical Constants

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

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

k

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

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

Mixing Logical and Numerics

Logical values, both variables and expressions, can be used with arithmetic operators, and can be assigned to integer variables. When used with an arithmetic operator, the logical value is treated as an identical integer value. Logical values cannot be used in any other place where arithmetic values are expected, such as in a **UNIT=** specifier in an **OPEN** statement, or in a call to a procedure expecting a real value.

You can use integer values as logical values in the following cases:

- **IF** statements and constructs
- **DO WHILE** statements and constructs
- Assignment to a logical variable

In these cases, the integer value is treated as an identical logical value (the bits which make up the integer value are treated the same as bits in a logical value). Real, complex, and other numeric values cannot be used as logical values in the three ways listed.

Conversely, integer values can be used in logical expressions, where a logical value is expected. Integer variables cannot be used in relational expressions. For information on expressions, see [Expressions](#).

The following are all valid uses of logicals and integers:

```
INTEGER a
LOGICAL b
b = 14
a = b
WRITE (*,*) a,b
END
!
INTEGER c,x
READ (*,*) c
IF (c) THEN
  x = 0
ELSE
  x = 15
END IF
WRITE (*,*) x
END
```

Derived Types

You can create additional (derived) data types from intrinsic data types or previously defined derived

types. You must provide a name for your derived type, which cannot be the name of an intrinsic data type or the name of another accessible derived type. Once you have created a derived type, you can declare variables or named constants to be of that derived type, so you can work with data objects which themselves are complex structures. A scalar derived-type data object is called a *structure*.

The **TYPE ... END TYPE** statement defines a derived type, not a variable, while the **TYPE** statement, followed by the name of the type, declares an object to be of that type.

The following example of a simple derived type defines member to consist of a 4-byte integer named age and a 20-character string named name:

```
TYPE member
  INTEGER age
  CHARACTER (LEN = 20) name
END TYPE member
```

An example declaration for a variable, using the first example, is:

```
TYPE (member) :: george
```

This establishes that the variable `george` has the characteristics of `member`, and consists of two parts: `age` and `name`.

You identify a structure element by specifying the sequence of elements needed to reach it, separated by a percent sign (%) or a period. In the example, the reference to the integer `age` element for `george` can be either `george%age`, or `george.age`.

If you use a period to delimit record elements, an element cannot have the name of a relational or logical operator (**NOT**, **AND**, **OR**, **GE**, **EQ**, and so on). If you use the name of a relational or logical operator and a period as a separator, the name is interpreted as a relational operator.

The components that you list for the structure are not necessarily stored in that order unless the declaration includes a **SEQUENCE** statement. For more information about data storage, see [Storage Association](#).

You can make a derived-type data object an array when you specify the derived type. You can use **DIMENSION** with an array specification or place an array specification following the name of the derived-type variable or named constant. In the following example, `a` and `b` are both variable arrays of derived type `pair`:

```
TYPE (pair)
  INTEGER i, j
END TYPE
TYPE (pair), DIMENSION (2, 2) :: a, b(3)
```

You can use derived-type objects as components of other derived-type objects. For example, you could specify two derived types as:

```
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

```

Visual Fortran supports other user-defined structures that are not standard Fortran 90. These structures are similar to derived types except that they are declared with a **STRUCTURE** block. These structures can contain mapped elements. A map specifies that one or more variables are stored contiguously in memory. The variables can be of any type, including other structures. A map can only appear within a **UNION** block, and a **UNION** block can only appear within a **STRUCTURE**. For example:

```

STRUCTURE /test/
  UNION
    MAP
      INTEGER(4) j, k, m
      CHARACTER(21) name
    END MAP
    MAP
      REAL(4) a, b, c
      COMPLEX(8) z
    END MAP
  END UNION
END STRUCTURE

```

In the preceding example, the 4-byte integers j, k, and m appear first in memory, followed immediately by the 21-character variable name. The 4-byte real variables a, b, and c share the same memory space as j, k, and m, respectively, while the 8-byte complex variable z shares the first 8 bytes of name's memory storage.

When maps are combined in a union, the variables overlap each other as they do in an **EQUIVALENCE** statement. In the following example, there are several definitions for a piece of memory, each consisting of three contiguous variables. Because these sets of variables are not all the same length, the shorter ones will not use all of the memory allocated for the longest **MAP** in the **UNION**. Which bytes are unused depends upon the value of the **PACK** compiler directive.

The following is an example of a map from the DFLIB.F90 module file (in the \DF\INCLUDE subdirectory):

```

STRUCTURE /MTH$E_INFO/
  INTEGER*4 ERRCODE          ! INPUT : One of the MTH$ values in DFLIB
  INTEGER*4 FTYPE           ! INPUT : One of the TY$ values in DFLIB
  UNION
    MAP
      REAL*4 R4ARG1         ! INPUT: First argument

```

```

    CHARACTER*12 R4FILL1
    REAL*4 R4ARG2           ! INPUT: Second argument (if any)
    CHARACTER*12 R4FILL2
    REAL*4 R4RES           ! OUTPUT: Desired result
    CHARACTER*12 R4FILL3
END MAP
MAP
    REAL*8 R8ARG1           ! INPUT: First argument
    CHARACTER*8 R8FILL1
    REAL*8 R8ARG2           ! INPUT: Second argument (if any)
    CHARACTER*8 R8FILL2
    REAL*8 R8RES           ! OUTPUT: Desired result
    CHARACTER*8 R8FILL3
END MAP
MAP
    COMPLEX*8 C8ARG1        ! INPUT: First argument
    CHARACTER*8 C8FILL1
    COMPLEX*8 C8ARG2        ! INPUT: Second argument (if any)
    CHARACTER*8 C8FILL2
    COMPLEX*8 C8RES         ! OUTPUT: Desired result
    CHARACTER*8 C8FILL3
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

```

For details on creating derived types, see [Derived Type](#) in the *Reference*. The related derived-type declarations are described in the [TYPE](#) entry in the *Reference*.

The following is also discussed in this section:

- [Default Initialization](#)
- [Determination of Derived Types](#)
- [Record Structures](#)

Default Initialization

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

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

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

Explicit initialization in a type declaration statement overrides default initialization.

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

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

Pointers can have an association status of associated, disassociated, or undefined. If no default

initialization status is specified, the status of the pointer is undefined. To specify disassociated status for a pointer component, use =>NULL().

Examples

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

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

Consider the following:

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

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

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

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

TYPE (MGR_REPORT) STARTUP
```

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

Determination of Derived Types

A derived data type can be defined only once in a program unit. The name can be used independently in other units with the same or different structures, as long as there is no host or use association between the program units.

Two entities have the same data type if they are declared with the same derived-type definition.

This agreement in type can be important in subroutine and function calls, as shown in the following example:

```
TYPE dual
  REAL y, z
END TYPE dual
TYPE (dual) :: x
CALL NEXTSUB (x)
. . .
CONTAINS
  SUBROUTINE NEXTSUB (a)
    TYPE (dual) :: a
    . . .
```

```
END SUBROUTINE NEXTSUB
```

In this example, using a derived-type object as the calling argument assures consistent argument lists to guard against a loss of data or the linker issuing an "unsatisfied external" error message because it does not recognize the subroutine as matching the call statement.

In the example, the definition of derived-type `dual` is recognized in subroutine `NEXTSUB` because it is contained within the main program. The variables `x` and `a` have the same type because they reference the same type definition. They also would have the same type if the derived-type definition was in a module and both `NEXTSUB` and the main program used that module.

If you do not have a host association and you are not using a module, you need to use the `SEQUENCE` property to work with the same derived type in different procedures. For an example, see the program `DTYPEARG.F90` in the Tutorial subdirectory in `/DF/Samples`. For other examples, see `DTYPEMOD.F90` and `DTYPECOM.F90`, also in the Tutorial subdirectory in `/DF/Samples`.

Derived-Type Values

The allowed values of a specific derived type are based on the allowed values for its ultimate intrinsic components. See [Intrinsic Data Types](#) for allowed values for intrinsic data types.

When you create a derived-type definition, you implicitly create a corresponding *structure constructor* that allows a scalar value of derived type to be constructed from a series of values, one for each defined component. This is a convenient alternative to using individual assignment statements for each component, which is also a valid method.

Where n components have been defined, the structure constructor has the form:

```
type-name ( expr1, expr2, ... , exprn )
```

Each of the expressions in the sequence specifies a component value. They must appear in the order given in the derived-type definition and must be consistent with the values allowed for each component. The derived type must be defined before you can use a structure constructor.

The following example shows a value assignment using a constructor corresponding to the derived-type member:

```
george = member ( 33, 'George Brown' )
```

You can make a derived-type object a named constant by using the `PARAMETER` attribute or statement with a structure constructor. An example is:

```
TYPE pair
  INTEGER i, j
END TYPE
TYPE (pair) p
PARAMETER (p = pair (9, 2))
TYPE (pair), PARAMETER :: q = pair (7, 3)
```

The complete example is in `DERIVED.F90` in the `/DF/SAMPLES/TUTORIAL` subdirectory.

If a derived-type component is an array, an array constructor is used to assign the component values. Array constructors are described in [Arrays and Pointers](#).

Record Structures

The record structure was defined in earlier versions of DIGITAL Fortran as a language extension. It is still supported in Visual Fortran, although its functionality has been replaced by standard Fortran 90 derived types. Record structures in existing code can be easily converted to Fortran 90 derived type structures for portability, but can also be left in their old form. In most cases, a DIGITAL Fortran record and a Fortran 90 derived type can be used interchangeably. The following section describes conversion information.

For more information on record structures, see the *Reference* entries for [STRUCTURE](#) and [RECORD](#).

Conversion to Fortran 90 Derived Types

DIGITAL Fortran record structures, using only intrinsic types, easily convert to Fortran 90 derived types. The conversion can be as simple as replacing the keyword **STRUCTURE** with **TYPE** and removing slash (/) marks. An example conversion is shown in the following table.

Record Structure	Fortran 90 Derived-Type
STRUCTURE /employee_name/ CHARACTER*25 last_name CHARACTER*15 first_name END STRUCTURE	TYPE employee_name CHARACTER*25 last_name CHARACTER*15 first_name END TYPE
STRUCTURE /employee_addr/ CHARACTER*20 street_name INTEGER(2) street_number INTEGER(2) apt_number CHARACTER*20 city CHARACTER*2 state INTEGER(4) zip END STRUCTURE	TYPE employee_addr CHARACTER*20 street_name INTEGER(2) street_number INTEGER(2) apt_number CHARACTER*20 city CHARACTER*2 state INTEGER(4) zip END TYPE

The record 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 in the following table.

Record Structure	Fortran 90 Derived-Type
STRUCTURE /employee_data/ RECORD /employee_name/ name RECORD /employee_addr/ addr INTEGER(4) telephone INTEGER(2) date_of_birth INTEGER(2) date_of_hire INTEGER(2) social_security(3) LOGICAL(2) married INTEGER(2) dependents END STRUCTURE	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

Type Declaration Statements

To use a variable or named constant, you must explicitly declare its type or accept the implicit type for that name. You can also specify other properties for these objects. A *declaration* provides the type, attributes, and other properties of an object; a *definition* provides its value.

The following is an example of a type declaration:

```
REAL real_val
```

The variable `real_val` is defined in a statement such as:

```
real_val = 25.44
```

The section *Defining Variables* discusses when a variable becomes defined and when it is undefined.

A type declaration statement has the general form:

```
type-spec [ [, attributes ] ... :: ] entity-declaration-list [ /c-list/ ]
```

type-spec

The data type, whether intrinsic or derived.

attributes

The set of attributes to be assigned to the data objects.

entity-declaration-list

The list of data object names and function names being declared.

c-list

A list of constants, as in a `DATA` statement. If *entity-declaration-list* is the name of a constant or an initialization expression, *c-list* cannot be present.

The *c-list* cannot specify more than one value unless it initializes an array. When initializing an array, the *c-list* must contain a value for every element in the array.

The type specification statement (*type-spec*) for each intrinsic type and for derived types is discussed in the appropriate section for each data type.

An explicit declaration can specify attributes for data objects, in addition to their types. Attributes describe the nature of the variable, and how it will be used. Data attributes are discussed in *Data Attributes*. A type statement can also initialize variables.

Declaration and specification statements can be either *entity-oriented*, or *attribute-oriented*. An entity-oriented specification is one where a data object's type and all of its attributes are specified in a single statement. An attribute-oriented specification is one which makes an attribute statement, listing all objects having that attribute. Examples of each are shown in the appropriate sections.

Explicit and Implicit Declarations

Assigning a value to an object implicitly declares it, if it has not been previously explicitly declared in a type statement. This is called *implicit declaration*. Explicit declaration statements are also called *type specification statements*.

Examples of type explicit declaration statements are:

```
REAL real_val
INTEGER int_val
TYPE reference
    character (len = 9) ssn
    character (len = 20) last_name
```

```

      character (len=20) first_name
END TYPE REFERENCE

TYPE (reference) person

```

In the preceding example, `int_val`, `real_val`, and `person` are all examples of explicitly-declared data objects. The data types in this example are `REAL`, `INTEGER`, and the derived type `REFERENCE`.

Fortran provides default implicit typing. If a variable, constant, or other data entity is not explicitly declared, and is not an intrinsic function, then its type and type parameters are determined implicitly by the first letter of its name. An explicit specification overrides any implicit typing.

The **IMPLICIT** statement lists a range of letters and a type, and possibly type parameters, for data objects whose names begin with one of those letters. You can also specify **IMPLICIT NONE**, which requires that you explicitly declare each variable and constant used in your program. The statement applies only in the program unit where it is used.

By default, a program unit is treated as if it had a host with the declaration:

```
IMPLICIT INTEGER (I-N), REAL (A-H, O-Z)
```

This means that variables having names beginning with the letters I through N are integers, A through H and O through Z are real numbers. For more information on default implicit assumptions, see [IMPLICIT](#) in the *Reference*.

See Also: [Declaration Statements for Noncharacter Types](#), [Declaration Statements for Character Types](#).

Declaration Statements for Noncharacter Types

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

Noncharacter Data Types

BYTE¹
LOGICAL²
LOGICAL([KIND=]1) (or **LOGICAL***1)
LOGICAL([KIND=]2) (or **LOGICAL***2)
LOGICAL([KIND=]4) (or **LOGICAL***4)
LOGICAL([KIND=]8) (or **LOGICAL***8)³
INTEGER⁴
INTEGER([KIND=]1) (or **INTEGER***1)
INTEGER([KIND=]2) (or **INTEGER***2)
INTEGER([KIND=]4) (or **INTEGER***4)
INTEGER([KIND=]8) (or **INTEGER***8)³
REAL⁵
REAL([KIND=]4) (or **REAL***4)
DOUBLE PRECISION (**REAL**([KIND=]8) or **REAL***8)
REAL([KIND=]16) (or **REAL***16)⁶
COMPLEX⁷

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

- ¹ Same as INTEGER(1).
- ² This is treated as default logical.
- ³ Alpha only.
- ⁴ This is treated as default integer.
- ⁵ This is treated as default real.
- ⁶ VMS, U*X.
- ⁷ This is treated as default complex.

In noncharacter type declaration statements, you can optionally specify the name of the data object or function as $v*n$, where n is the length (in bytes) of v . The length specified overrides the length implied by the data type.

The value for n must be a valid length for the type of v . The type specifiers BYTE, DOUBLE PRECISION, and DOUBLE COMPLEX have one valid length, so the n specifier is invalid for them.

For an array specification, the n must be placed immediately following the array name; for example, in an INTEGER declaration statement, IVEC*2(10) is an INTEGER(2) array of 10 elements.

Examples

In a noncharacter type declaration statement, a subsequent kind parameter overrides any initial kind parameter. For example, consider the following statements:

```
INTEGER(KIND=2) I, J, K, M12*4, Q, IVEC*4(10)
REAL(KIND=8) WX1, WXZ, WX3*4, WX5, WX6*4
REAL(KIND=8) PI/3.14159E0/, E/2.72E0/, QARRAY(10)/5*0.0,5*1.0/
```

In the first statement, M12*4 and IVEC*4 override the KIND=2 specification. In the second statement, WX3*4 and WX6*4 override the KIND=8 specification. In the third statement, QARRAY is initialized with implicit conversion of the REAL(4) constants to a REAL(8) data type.

Declaration Statements for Character Types

A CHARACTER type specifier can be immediately followed by the length of the character object or function. It takes one of the following forms:

Keyword Forms

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

Nonkeyword Form

```
CHARACTER*len[,]
```

len

Is one of the following:

- In keyword forms

The *len* is a specification expression or an asterisk (*). If no length is specified, the default length is 1.

If the length evaluates to a negative value, the length of the character entity is zero.

- In nonkeyword form

The *len* is a specification expression or an asterisk enclosed in parentheses, or a scalar integer literal constant (with no kind parameter). The comma is permitted only if no double colon (::) appears in the type declaration statement.

This form can also (optionally) be specified following the name of the data object or function (*v*len*). In this case, the length specified overrides any length following the CHARACTER type specifier.

The range for *len* in both forms is 1 to 2**31-1 for Windows NT systems on Alpha processors; 1 to 65535 for Windows NT systems on Intel processors.

k

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

Rules and Behavior

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

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

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

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

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

Examples

The following example declares an array NAMES containing 100 32-character elements, an array SOCSEC containing 100 9-character elements, and a variable NAMETY that is 10 characters long and has an initial value of 'ABCDEFGHIJ'.

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

The following example includes a CHARACTER statement declaring two 8-character variables, LAST and FIRST.

```
INTEGER, PARAMETER :: LENGTH=4
```

```
CHARACTER*(4+LENGTH) LAST, FIRST
```

The following example shows a **CHARACTER** statement declaring an array **LETTER** containing 26 one-character elements. It also declares a dummy argument **BUBBLE** that has a passed length defined by the calling program.

```
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
```

Data Initialization and the DATA Statement

Variables can be given initial values in type-declaration statements and by using **DATA** statements.

You can give initial values to the variables in your program by using a type declaration statement. An *entity-oriented* declaration is one where a data object's type and all of its attributes are specified in a single statement. Initialization of a variable in an entity-oriented type declaration follows the same rules as an assignment statement. For more information, see [Assignment Statements](#).

You can also initialize variables using a **DATA** statement. The **DATA** statement has the form:

```
DATA data-object / data-value-list / [ [ , ] data-object / data-value-list / ] ...
```

data-object

A variable or data implied **DO**.

data-value-list

The data constants to be assigned, which can include the asterisk (*) repeat integer.

The non-negative repeat integer specifies how many times the following constant is used. The repeat integer can be an integer constant or a named constant. The data constant can be any valid data type.

The variables become initialized with their corresponding constants in accordance with the rules of intrinsic assignment.

When you construct a **DATA** statement, you must make sure that when the lists are expanded, the combined length of the data objects matches the combined length of data values.

You must also be careful that corresponding data types in the lists are compatible. Character and logical variables must have corresponding constants of their type. Real and complex variables must have corresponding constants of integer, real, or complex type. Integer variables can have corresponding constants of integer, real, or complex type, or binary, octal, or hexadecimal literal constants. A derived-type variable must have a corresponding constant of the same type. If a variable in a **DATA** statement is typed implicitly, then any subsequent type statements that include that variable must agree with the implicit typing.

If you assign values in a **DATA** statement to an array, or part of one, it must have had its array properties defined earlier.

In the following examples, 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 Types. 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.

```
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' /
```

The **DATA** statement allows multiple data object and data value lists in a single statement. Such repeated specifications can be a source of confusion and user errors. Using separate **DATA** statements usually results in cleaner code and fewer coding errors. The first **DATA** statement in the previous example could be written as:

```
DATA name / 'JOHN DOE' /
DATA miles / 10*0 /
```

See also DATA in the *Reference*.

Data Attributes

Attributes, in general, describe properties that determine how a data object can be used in a program. You can use one or more statements to assign attributes to data objects (For example, one statement could declare a real type and another declare dimensions.) However, a particular attribute of the object can be declared only once.

The Visual Fortran attributes are listed in the following table.

Attribute name	Description	Can be used with
ALLOCATABLE	Specifies that the bounds for an array are to be determined on execution of the ALLOCATE statement.	Arrays
AUTOMATIC	Declares a variable to be on the stack, rather than at a static memory location.	Variables
DIMENSION	Specifies an array.	Constants or variables
EXTERNAL	Declares a name to be an external function.	Functions or subroutines
INTENT	Specifies intended use of a procedure's dummy argument.	Function or subroutine dummy arguments
INTRINSIC	Declares a name to be an intrinsic function.	Functions or subroutines
OPTIONAL	Allows a procedure to be called without referring to	Function or subroutine

	this dummy argument.	dummy arguments
PARAMETER	Declares a name to be a constant.	Constant data objects
POINTER	Declares a data object to be a pointer.	Variables
PRIVATE	Restricts access to an entity in a module to the module itself.	Constants, variables, or module procedures
PUBLIC	Declares an entity in a module to be available outside that module.	Constants, variables, or module procedures
SAVE	Retains a variables's value, definition, association, and allocation status after the routine in which they are declared completes execution.	Variables or common blocks
STATIC	Specifies that a variable has a storage class of static.	Variables
TARGET	Declares a data object to be a target.	Variables
VOLATILE	Declares an object to be entirely unpredictable and prevents the object from being optimized during compilation.	Data objects or /common-block/

The PARAMETER, PUBLIC, PRIVATE, SAVE, **STATIC**, and **AUTOMATIC** attributes are discussed in the following sections. The POINTER, TARGET, and ALLOCATABLE attributes are discussed in Arrays and Pointers. The EXTERNAL, INTRINSIC, INTENT, and OPTIONAL attributes are discussed in Program Units and Procedures. The DIMENSION and VOLATILE attributes are discussed in the *Reference*.

Components of a derived type can include only the DIMENSION and POINTER attributes in their definition.

See also: Specifying Properties by Using a Compiler Directive.

The PARAMETER Attribute and Statement

Named constants are declared and defined in type declaration statements containing the PARAMETER attribute and by using **PARAMETER** statements. The defined values cannot be changed during program execution.

PARAMETER can be used in both attribute and statement form. The attribute-oriented **PARAMETER** statement has the form:

PARAMETER [(*named-constant* = *initialization expression* [, ...] [])]

If the *named-constant* in a **PARAMETER** statement is typed implicitly, any subsequent type declaration statement that includes that constant must confirm the implied type and the values of any implied type parameters.

Use of the optional parentheses can be controlled by using the /noaltparam compiler option. The default is /altparam, which allows the option.

```
PARAMETER (xmax = 50.)
PARAMETER y = 2.1*2.0
```

The form for using the PARAMETER attribute in an entity-oriented type declaration statement is:

type-spec, PARAMETER [, *attribute*] :: *named-constant* = *initialization expr*

More than one *named-constant* can be specified in a single type declaration statement. For example:

```
REAL, PARAMETER :: xmax = 50., y = 2.1*2.0
```

The *named-constant* must have already been defined in either the same statement or in a prior type declaration statement, or it should be accessible through either host or use association. For information on host and use association, see [Program Units and Procedures](#).

```
REAL, PARAMETER :: xmax = 50., y = 2.1*2.0
```

See also [PARAMETER](#) in the *Reference*.

The PUBLIC and PRIVATE Attributes and Statements

Entities with the PUBLIC attribute are accessible in other program units by the USE statement. Entities with the PRIVATE attribute are not accessible outside the module. Entities without an accessibility specification have the default accessibility, which is PUBLIC unless the default has been changed by a PRIVATE statement. For more information about the USE statement and modules, see [Program Units and Procedures](#).

The form for using the access specification statement is:

PUBLIC | **PRIVATE** [[::] *access-identification-list*]

Examples of accessibility statements are:

```
MODULE EX
  PRIVATE
  PRIVATE :: x, y, z
  PUBLIC  :: a, b, c, assignment (=), operator (+)
```

The form for using the access specification in a type declaration statement is:

type-spec, **PUBLIC** | **PRIVATE** [, *attribute-list*] :: *entity-list*

The *entity-list* can be a named variable, procedure, derived type, named constant, or namelist group. It can also be a generic name.

Some examples of specifying the PUBLIC and PRIVATE attribute are:

```
REAL, PRIVATE :: x, y, z
REAL, DIMENSION (10, 10), PUBLIC :: a, b
```

An object must not be given the PUBLIC attribute if its type already has the PRIVATE attribute.

Accessibility attributes can be used only in modules. If a **PUBLIC** or **PRIVATE** statement is used without an access identifier list, the statement sets the default accessibility that applies to all potentially accessible entities in the scoping unit. This means that every variable, constant, and subprogram contained in the module has that attribute unless you explicitly declare specific ones to have the nondefault attribute. Only one **PUBLIC** or **PRIVATE** statement without an access identifier list is permitted in a program unit.

For example, derived types defined in a module are by default accessible in any unit that uses the module. The default can be changed to limit accessibility to the module itself. A particular derived type can be declared private, or it can be public while some of its components are private.

See also PUBLIC and PRIVATE in the *Reference*.

The SAVE Attribute and Statement

You can use SAVE to preserve the values that variables have in procedures. Objects having the SAVE attribute retain their association status, allocation status, definition status, and value following execution of a **RETURN** or **END** statement in the scoping unit containing their declaration. Saved objects in the scoping unit of a module retain their properties when any procedure that accesses the module in a **USE** statement executes a **RETURN** or **END** statement. Objects declared with the SAVE attribute in a subprogram are shared by all instances of the subprogram.

You can save all objects in a program unit by using **SAVE** without an entity list. If you do this, no other explicit occurrence of the SAVE attribute or **SAVE** statement is allowed in the same scoping unit. A **SAVE** statement with an empty entity list is treated as though it contained the names of every allowed object in the same scoping unit.

The SAVE attribute cannot be used for an object in a common block, a dummy argument, a procedure, a function result, or an automatic data object. The SAVE attribute has no effect in the specification part of a main program.

You can save an entire common block even though you cannot specify the SAVE attribute for individual items within a common block. All objects within a saved common block are saved. If a common block is declared saved outside of the main program, it must be declared to have the SAVE attribute in every scoping unit in which that common block appears except in the scoping unit of the main program. The current values of objects in a saved common block when a **RETURN** or **END** statement is executed are available to the next scoping unit in sequence that uses the common block.

If a named common block is given the SAVE attribute in the main program, the current values of the common block are available to each scoping unit that specifies that named common block. The definition status of each object in the common block depends on the association that has been established for the common block storage sequence.

The SAVE attribute can be assigned in a type definition statement or in a **SAVE** statement. The form of the statement is:

```
SAVE [ [ :: ] entity-list ]
```

The *entity-list* includes object names and common block names delimited by slashes (/).

An example of a **SAVE** statement is:

```
SAVE a, b, c, / blocka /, d
```

Specify the SAVE attribute in the following form:

```
type-spec, SAVE [ [, attribute-spec ] :: ] entity-list
```

An example of an SAVE attribute specification is:

```
REAL, SAVE :: a, b
```

See also [SAVE](#) in the *Reference*.

The STATIC Attribute and Statement

The **STATIC** attribute specifies that a variable has a storage class of static, which means that the variable remains in memory for the duration of program execution. Its value is retained between calls to the containing procedure. This attribute is equivalent to the Fortran SAVE attribute and the C static attribute. It can be expressed as either a statement or as an attribute. The **STATIC** statement has the form:

```
STATIC [ :: ] variable_name
```

The variable name list can include object names and common block names delimited by slashes (/).

An example of a **STATIC** statement is:

```
STATIC :: a, b, c
```

Specify the **STATIC** attribute in the following form:

```
type-spec, STATIC [ [, attribute-spec ] :: ] entity-list
```

An example of an **STATIC** attribute specification is:

```
INTEGER, STATIC :: d
```

See also [STATIC](#) in the *Reference*.

The AUTOMATIC Attribute and Statement

This attribute can be specified in a type declaration or as a statement. It declares variables to be on the stack, rather than at a static memory location. In Visual Fortran, all variables are static by default. A variable declared as automatic has no fixed memory location; a section of stack memory is allocated for the variable as needed. Automatic variables within procedures are discarded when the procedure completes execution. The **AUTOMATIC** statement has the following form:

```
AUTOMATIC [ [ :: ] entity-list ]
```

The *entity-list* consists of variable names or array specifications. If no variable names follow the **AUTOMATIC** statement, then all variables in that program unit are considered automatic.

An example of an **AUTOMATIC** statement is:

```
AUTOMATIC :: e, f, g
```

Specify the **AUTOMATIC** attribute in the following form:

```
type-spec, AUTOMATIC [ [ , attribute-spec ] :: ] entity-list
```

An example of an AUTOMATIC attribute specification is:

```
INTEGER, AUTOMATIC :: h
```

See also [AUTOMATIC](#) in the *Reference*.

Specifying Properties by Using a Compiler Directive

The **ATTRIBUTES** compiler directive provides additional features beyond standard Fortran 90. This directive allows your program, for example, to use the calling conventions of Microsoft C, and to pass arguments by value or by reference. For more information on mixed-language programming, see [Programming With Mixed Languages](#).

The properties described in this section cannot be included in any entity-oriented type declarations. They are specified by using the **ATTRIBUTES** directive, with the following syntax:

```
cDEC$ ATTRIBUTES attr-list :: object-list
```

c

Is a c, C, !, or *.

attr-list

The properties being specified; one or more of the following options: ALIAS, C, DLLEXPORT, DLLIMPORT, EXTERN, REFERENCE, STDCALL, VALUE, or VARYING.

object-list

The data objects and procedures being declared to have the properties.

ATTRIBUTES options can be used in subroutine and function definitions, in type declarations, and with the **INTERFACE** and **ENTRY** statements. These properties applied to entities available to a procedure through host or use association are carried through the association. In the following example, the call to happy within go uses the C option specified in the interface block.

```
MODULE mod
INTERFACE
  SUBROUTINE happy
!DEC$ ATTRIBUTES C :: happy
  END SUBROUTINE
END INTERFACE
CONTAINS
  SUBROUTINE go
  CALL happy(12)
  END SUBROUTINE
END MODULE
```

For more information, see [ATTRIBUTES](#) and [General Compiler Directives](#).

Storage Association

Storage sequences describe relationships between variables, common blocks, and result variables. Storage is *associated* when different data objects share the same storage units. Storage association is

similar to name or pointer association in that the same value can be referenced by more than one name. With storage association, however, variables can overlap one another, or their memory storage can begin and end in different locations. The **EQUIVALENCE**, **COMMON**, **SEQUENCE**, and **ENTRY** statements can cause variables to be storage associated.

The **ENTRY** statement causes the result variables of a function to be storage associated. All subprogram entries are storage associated with the function result.

For more information on **ENTRY**, see [Program Units and Procedures](#) and [ENTRY](#) in the *Reference*.

Storage Units and Storage Sequence

Storage association is described in terms of storage units. A storage unit contains a single value, and can be character, numeric, or unspecified. The table [Memory Requirements and Default Kinds](#) shows variable types and their associated storage unit size.

An ordered sequence of storage units comprises a *storage sequence*. Several storage sequences can be used consecutively to create a composite sequence. The order of units in a composite sequence is that of its component sequences in succession, ignoring any zero-sized sequences. Common block storage sequences are described in The **COMMON** Statement.

Two objects are said to be storage associated if the i th storage unit of one is the same as the j th storage unit of the other. This causes the $(i+k)$ th unit of the first to be the same as the $(j+k)$ th unit of the other, as long as $(i+k)$ and $(j+k)$ are less than or equal to the length of the first and second objects, respectively. For example:

```
REAL A (4), B
COMPLEX C (2)
DOUBLE PRECISION D
EQUIVALENCE (C (2), A (2), B), (A, D)
```

The second storage unit of C (element C(2)) occupies the same memory location as the second and third units of A (elements A(2) and A(3)) because a complex number is made of two real numbers. The second unit of A (element A(2)) occupies the same location as all of B, and D occupies the same location as the first two units of A (elements A(1) and A(2)).

Two entities are *totally associated* if they have the same storage sequence in memory, or *partially associated* if they share only part of a storage sequence. In the previous example, A(2) and B are totally associated. The following are partially associated:

- A(1) and C(1)
- A(2) and C(2)
- A(3) and C(2)
- B and C(2)
- A(1) and D
- A(2) and D
- B and D
- C(1) and D
- C(2) and D

Associated character entities can overlap, as in the following example:

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

A, B, and C are partially associated.

The COMMON Statement

The **COMMON** statement allows two or more program units to directly share variables, without having to pass them as arguments. It allows for storage association among variables in different program units.

The module feature of Fortran also allows different program units to share variables, but does not cause storage association. Although common blocks can be used in modules, they cannot also be declared in a program unit that uses that module. The association created by use of modules is use association rather than storage association. See [Program Units and Procedures](#) for more information on use association.

Data objects that are storage associated with an entity in a common block are considered to be in that common block.

The syntax of the **COMMON** statement is:

```
COMMON [ / [ block-name ] / ] object-list [ [ , ] / [ block-name ] / object-list ] ...
```

block-name

The name of the common block.

object-list

The names of the common block data objects.

Any blank or named common block can appear more than once in a program unit. Each time the same common name is used in one program unit, it is treated as a continuation of the list for that name. Similarly, each blank common block object list in a scoping unit is treated as a continuation of blank common.

A variable can be used in only one common block within a program unit.

```
COMMON / BLOCKA / A, B, C (10, 30)
COMMON I, J, K
```

Common Block Storage Sequence

A storage sequence is formed for each common block, as follows:

1. A storage sequence is formed from the lists for the common block. The order is the same as the order of the appearance of the object lists.
2. The storage sequence formed in (1) includes all storage units associated with it by equivalence. The sequence can be extended only by adding storage units beyond the last storage unit. Data objects associated with an entity in a common block are considered to be in that common block.

You can place a derived-type object in a common block, but it must be a sequence type. If

derived-type objects of numeric sequence type or character sequence type appear in common, it is as if the individual components were enumerated directly in the common list.

Size of a Common Block

The size of a common block is the size of its common block storage sequence, including any extensions resulting from equivalence association. A named common block must be the same size in all program units that access it. Blank common blocks can be of different sizes, but both must start with the same sequence of items.

Common Association

All common blocks with the same name (including blank common) have the same first storage unit. If they are zero-sized, common blocks with the same name are storage associated with one another. This results in the association of objects in different program units.

The variables that you place in common statements in different program units must be associated with objects of the same type and kind. You cannot, for example, associate a logical type with a double precision type.

The following example shows a valid association between subroutines in different program units. The object lists agree in number, type, and kind of data objects:

```

SUBROUTINE unit1
REAL(8)      x(5)
INTEGER      J
CHARACTER    str*12
TYPE(member) club(50)
COMMON / blocka / x, j, str, club
...

SUBROUTINE unit2
REAL(8)      z(5)
INTEGER      m
CHARACTER    chr*12
TYPE(member) myclub(50)
COMMON / blocka / z, m, chr, myclub
...

```

The EQUIVALENCE Statement

The **EQUIVALENCE** statement causes all listed items to have the same first memory location. The association is local to the program unit, unless one of the equivalenced entities is also in a common block. If the **EQUIVALENCE** statement appears in a module used by the main program, then the scope of the association extends to the main program. For more information on the concept of scope, see [Program Units and Procedures](#).

If the objects have differing types or type parameters, the association does not cause any type conversion or imply mathematical equivalence.

When you create an **EQUIVALENCE** statement, you define a storage association between variables. All of the nonzero-sized sequences in an equivalence set, if any, have the same first storage unit, and all of the zero-sized sequences, if any, are storage associated with one another and with the first storage unit of any nonzero-sized sequences.

The syntax of the **EQUIVALENCE** statement is:

```
EQUIVALENCE ( object, object-list ) [ , ( object, object-list ) ] ...
```

object

One of the sharing objects, which can be a variable name, an array element, or a character substring.

object-list

One or more additional objects which share the same storage unit.

When you construct equivalence lists in standard Fortran 90, you must have all items in the lists correspond one-for-one in type and kind. You cannot, for example, equivalence a single precision array with a double precision array because the storage units for the array elements have different word lengths.

You cannot construct equivalences that contradict the storage units of variables. A variable cannot occupy more than one storage unit, for example. Consecutive array elements must be stored in sequence, so you cannot force the elements of arrays out of their storage sequence. For more information, see the **EQUIVALENCE** statement in the *Reference*.

An example of an **EQUIVALENCE** statement is:

```
CHARACTER (LEN = 4) :: a, b
CHARACTER (LEN = 3) :: c(2)
EQUIVALENCE (a, c(1)), (b, c(2))
```

This causes the following alignment:

1	2	3	4	5	6	7
a(1:1)	a(2:2)	a(3:3)	a(4:4)			
			b(1:1)	b(2:2)	b(3:3)	b(4:4)
c(1)(1:1)	c(1)(2:2)	c(1)(3:3)	c(2)(1:1)	c(2)(2:2)	c(2)(3:3)	

In this example, the fourth element of a, the first element of b, and the first element of c(2) share the same storage unit.

The **NAMELIST** Statement

A **NAMELIST** statement groups named data objects so they can be referred to by a single group name. The **NAMELIST** statement is used primarily for transferring data.

When entering data for namelist variables, enter the variables in any order. The variables are output in the same order in which they are listed in the **NAMELIST** statement.

A namelist variable is like any other variable with regard to use or host association and data typing. You can include a namelist variable in more than one namelist group, and you can use a namelist group name more than once within a program unit. Each time you use the group name, the new variable lists are appended to the original list.

A sample program, **NAMELIST.F90**, is included in the `/DF/SAMPLES/TUTORIAL` subdirectory.

Namelists are more fully described in [Input/Output Editing](#). See also [NAMELIST](#) in the *Reference*.

Expressions

Individual units of meaning such as variables, constants, function references, and operators are combined together to form expressions. An *expression* is a formula for computing a value. Operators specify the actions to be performed on the operands. In the following expression, for example, the slash (/) is an operator and chickens and coops are operands:

```
chickens / coops
```

An expression can be part of an assignment statement, it can be the control part of an **IF** statement, or it can be an argument to a procedure. In the following example, the entire line is an assignment statement, and the portion to the right of the equal sign (=) is an expression:

```
cost = 10.95 * chickens / coops
```

Intrinsic Operators

There are four categories of intrinsic operators, as shown in the following list.

Category	Intrinsic operators	Used with
Numeric	** , * , / , + , - unary + , unary -	Integer, real, or complex values in any combination.
Character	//	Character of any length.
Relational	.EQ. , .NE. , = , / = .GT. , .GE. , .LT. , .LE. , > , > = , < , < =	Both of integer or real type or both of character type; yields logical value. .EQ. and .NE. also accept complex types.
Logical	unary .NOT. , .AND. , .OR. , .XOR. , .EQV. , .NEQV.	Logical or integer values.

A unary operator has a single operand; binary operators require two operands:

```
-(a+b)        ! - is unary, + is binary
```

You can create *defined operations* in addition to the intrinsic operations. You can make a defined operation resolvable into one or more intrinsic operations, similar to the way that derived data types can be resolved into intrinsic data types. A defined operation is defined by a function and an interface block. See [Defined Operators and Expressions](#), for more information on how to do this.

Evaluation of Operations

You must give a value to any variables used as operands before referring to them. You cannot use undefined mathematical operations. The following operations are prohibited:

- Division by zero
- Raising a zero-value operand to a negative or zero power
- Raising a negative-value operand to a non-integer power

The compiler does not necessarily evaluate all parts of expressions. For example, in an expression that contains a multiplication by zero, the expression in parentheses might not be evaluated:

```
(37.8 / scale**expo + factor) * 0.0
```

Similarly, if a false value is part of an expression including the **.AND.** operator, the expression might not be evaluated. In the following example, the expression (switch .EQ. on) might not be evaluated:

```
((3 .LE. 1) .AND. (switch .EQ. on))
```

Precedence of Operators

When several operators appear in the same expression without controlling parentheses, their effects are evaluated in a specific order. The following table shows the set of expression operators and their precedence in descending order.

Table: Expression Operators and Precedence (in descending order)		
Category of operator	Operator	Action with equal precedence
	any user-defined unary operator	N/A
Numeric	**	Right-to-left
Numeric	* or /	Left-to-right
Numeric	unary + or -	N/A
Numeric	binary + or -	Left-to-right
Character	//	Left-to-right
Relational	.EQ., .NE., .LT., .LE., .GT., .GE., =, /=, <, <=, >, >=	N/A
Logical	.NOT.	N/A
Logical	.AND.	Left-to-right
Logical	.OR.	Left-to-right
Logical	.XOR., .EQV., or .NEQV.	Left-to-right
	any user-defined binary operator	Left-to-right

For example, the following two expressions are equivalent, since the unary minus has a lower precedence than exponentiation:

```
-a**2  
-(a**2)
```

Creating Expressions

An expression is formed from operands, operators, and parentheses. An expression can be enclosed in parentheses and treated as a simple operand. More complex expressions can be made by including operators, as in the following examples:

```
a + b
```

```
(a - b) * c
a ** b
c .AND. d
f // g
```

Any variable, array, array element, derived-type component, or function that is referred to in an expression must be defined at the time the reference is made, or the results are undefined.

Expression Data Types and Conformability

The data type of an expression depends on its operators and operands. Its shape depends on the operators and the shape of the arrays in the expression. The data type of the result of an expression can be either an intrinsic type or a derived type.

If an expression contains two operands of the same type, but different kinds, the result is of the kind having the greater precision. For example, if a single precision value is added to a double-precision value, the result is a double precision real number. See [Type Conversion of Numeric Operands](#) for more detail.

For all intrinsic binary operations, both operands must be the same shape if they are arrays, or must both be scalar.

Scalar and Array Expressions

Expressions are classified as either scalar or array expressions.

An example scalar expression, where q and r are scalars, is:

```
q + 2.3 + r
```

An example array expression, where a and b are arrays declared with dimension $a(10)$ and $b(10)$, is:

```
a + b
```

In this example, the two arrays have the same shape. This is required for two array operands.

If one operand is a scalar and the other an array, the scalar is treated as though it is of the same shape as the array, with each of its elements equal to the scalar. An example expression is:

```
a + r
```

In this example, if r is a scalar and a is an array, the value of r is added to each element of a .

Numeric Expressions

A numeric expression produces an integer, a real number (single or double precision), a complex number, or an array of those types. Numeric expressions are built from the following operands:

- Numeric constants
- Symbolic names for numeric constants
- Constant subobjects
- Variable references
- Array references

- Array element references
- Function references
- Derived type references
- Structure component

The numeric operands can be mixed types and kinds: integer, real, double-precision real, and complex. The nature of the result is determined by the type and kind of the variable being defined. When the type or kind are mixed, the operands are converted to match that of the operand of greatest precision before the operation is performed.

Table: Arithmetic Operators		
Operator	Operation	Precedence
**	Exponentiation	1 (highest)
/	Division	2
*	Multiplication	2
-	Subtraction (binary) or negation (unary)	3
+	Addition (binary) or identity (unary)	3

When consecutive operations are of equal precedence, the leftmost operation is often performed first. For example, the expression `first/second*third` is equivalent to `(first/second)*third`. When there are two consecutive exponentiation operations, the rightmost operation is performed first. For example, the following expressions are equivalent:

```
first**second**third
first**(second**third)
```

Fortran does not allow two numeric operators to appear consecutively. For example, Fortran prohibits `first**-second`, but permits `first**(-second)`.

The following are examples of the precedence of numeric operators.

Expression	Equivalent expression
<code>3 * 7 + 5</code>	<code>(3 * 7) + 5</code>
<code>- one**two</code>	<code>-(one**two)</code>
<code>+ x / y</code>	<code>+(x / y)</code>
<code>area / g - qu**2**fact</code>	<code>(area / g) - (qu**(2**fact))</code>

Integer Division

When one integer is divided by another, the truncated quotient of the two operands is returned. Thus, `7/3` evaluates to 2, `(-7)/3` evaluates to -2, and both `9/10` and `9/(-10)` evaluate to zero. In general, the expression `a*i/j` does not give the same result as `a*(i/j)` because of the different sequence of operations in the expressions.

For example, look at the following assignment statement:

```
i = 1/4 + 1/4 + 1/4 + 1/4
```

Division has higher precedence than addition, so the expression is equivalent to `(1/4) + (1/4) + (1/4) + (1/4)`. The quotient of `1/4` is truncated, and the result is 0. The assignment, therefore, sets `i` equal to zero.

Type Conversion of Numeric Operands

When all operands of a numeric expression are of the same data type and kind, the value returned by the expression is also of that type and kind. When the operands are of different data types and kinds, the type and kind of the value returned by the expression are those of the highest-ranked operand. The exception to the rule is operations involving both REAL(8) numbers and COMPLEX(4) numbers, which yield COMPLEX(8) results.

The following table shows the ranking of each data type:

Rank	Data Type
Highest	COMPLEX(8)
·	COMPLEX(4)
·	REAL(16) (VMX, U*X)
·	REAL(8)
·	REAL(4)
·	INTEGER(8) (Alpha only)
·	INTEGER(4)
·	INTEGER(2)
·	INTEGER(1)
·	LOGICAL(8) (Alpha only)
·	LOGICAL(4)
·	LOGICAL(2)
Lowest	LOGICAL(1) or BYTE

For example, when an operation is performed on an INTEGER(2) operand and a REAL(4) operand, the INTEGER(2) operand is first converted to REAL(4). The result of the operation is also a value of data type REAL(4). Similarly, in an operation on a real number and a complex number, the real number is first converted to a complex number with a zero imaginary part, and the result of the operation is also complex.

All intermediate numeric integer calculations are performed with INTEGER(4) precision and therefore do not overflow INTEGER(2) precision.

The following table summarizes the data conversion rules for numeric assignment statements.

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

(KIND=4)	fraction (.0) to E and assign to V	V	portion of E to V; LS portion of E is rounded	E to V; LS portion of E is rounded	E to V; imaginary part of E is not used	of the real part of E to V; LS portion of the real part of E is rounded; imaginary part of E is not used
REAL (KIND=8)	Append fraction (.0) to E and assign to V	Assign E to MS portion of V; LS portion of V is 0	Assign E to V	Assign MS portion of E to V; LS portion of E is rounded	Assign real part of E to MS of V; LS portion of V is 0; imaginary part of E is not used	Assign real part of E to V; imaginary part of E is not used
REAL (KIND=16) (VMS, U*X)	Append fraction (.0) to E and assign to V	Assign E to MS portion of V; LS portion of V is 0	Assign E to MS portion of V; LS portion of V is 0	Assign E to V	Assign real part of E to MS of V; LS portion of V is 0; imaginary part of E is not used	Assign real part of E to MS portion of V; LS portion of real part of V is 0; imaginary part of E is not used
COMPLEX (KIND=4)	Append fraction (.0) to E and assign to real part of V; imaginary part of V is 0.0	Assign E to real part of V; imaginary part of V is 0.0	Assign MS portion of E to real part of V; LS portion of E is rounded; imaginary part of V is 0.0	Assign MS portion of E to real part of V; LS portion of E is rounded; imaginary part of V is 0.0	Assign E to V	Assign MS portion of real part of E to real part of V; LS portion of real part of E is rounded. Assign MS portion of imaginary part of V; LS portion of imaginary part of E is rounded
COMPLEX (KIND=8)	Append fraction (.0) to E and assign to V; imaginary part of V is 0.0	Assign E to MS portion of real part of V; imaginary part of V is 0.0	Assign E to real part of V; imaginary part is 0.0	Assign MS portion of E to real part of V; LS portion of E is rounded; imaginary part of V is 0.0	Assign real part of E to MS portion of real part of V; LS portion of real part is 0. Assign imaginary part of E to MS portion of imaginary part of V; LS portions of imaginary part is 0.	Assign E to V

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

In converting from single precision to double precision, either as real or complex types, the added digits are generally nonzero digits. This means a conversion increases the precision, but not the accuracy, of a converted number or the resultant of an operation. For example:

```
PROGRAM datconv
! demonstrate data precision when converting types
```

```
REAL(4) a
REAL(8) c
```



```

a = 1.11
c = a

WRITE (*,10) a
WRITE (*,20) c

10 FORMAT (' a:', F9.7)
20 FORMAT (' c:', F18.16)

END

```

The output for this program is:

```

a: 1.1100000
c: 1.1100000143051150

```

In this example, although *c* is of greater precision than *a*, its accuracy is no better. If this is important to you, you might have to work with double precision from the beginning of the program. You can use either the **cDEC\$ REAL:8** compiler directive or a compiler option to declare all default real variables double precision. You can also explicitly declare them using the **KIND** type parameter.

Character Expressions

A character expression produces a value that is of type **CHARACTER**. There are six operands used in character expressions:

- Character constants
- Character variable references
- Character array-element references
- Character function references
- Character substrings
- Character structure component references

Character operands can be individual characters, printable or not, and strings of any lengths.

The only character operator is the concatenation operator (`//`).

The expression *first* // *second* produces a character string equal to the value of *first* concatenated on the right with the value of *second*. For example, the expression 'AB '//CDE' produces the string:

```
'AB CDE'
```

When two or more string variables are concatenated, the resulting string is as long as the declared lengths of the string variables. Leading and trailing blanks are not discarded. For example:

```

CHARACTER*10 first
CHARACTER*6 second
first = 'heaven'
second = ' sent'
WRITE (*, *) first//second

```

The result is a 16-character string.

```

heaven      sent
-----1-----2

```

Note that there are five spaces between 'heaven' and 'sent'.

You can manipulate character strings using the functions **LEN** and **LEN_TRIM**, which determine the length of a string and the length without trailing blanks, respectively.

If you concatenate C strings, remember a null character (\0) is automatically appended to each C string. For example, the expression:

```
'hello 'C // 'world'C
```

is equivalent to the following C string:

```
'hello \0world'C
```

The C compiler and libraries treat this string as just 'hello' because C strings are terminated by the null character, \0'.

Special Characters

When you are building a character string, you can also include nonprinting or other special characters. Special characters can be embedded in the string while viewing it in Microsoft Developer Studio (see the Microsoft Developer Studio documentation for more information).

You can also use the intrinsic function **CHAR** to assign ASCII values (such as form feed, or printer control sequences) to a character string.

For more discussion of special characters, see [Using National Language Support Routines](#).

Evaluation of Character Expressions

A character expression might not be fully evaluated. For example:

```
CHARACTER (LEN = 2) c1, c2, c3, cf    ! cf is a character function
c1 = c2 // cf (c3)
```

In this example, the function `cf` might not be evaluated since, because they are of equal lengths, `c2` fully defines `c1`.

Relational Expressions

Relational expressions compare the values of two numeric or character expressions. You cannot compare a numeric variable with a character or logical variable in standard Fortran 90.

In Visual Fortran, a numeric expression can be compared with a character expression. The numeric expression is treated as if it were a character expression (that is, a sequence of byte values). The two expressions must be identical on a byte-by-byte basis, or they are not equal.

For example, if 'A' were assigned to a 4-byte integer, the ASCII value of the letter A (hex 41) would be the variable's least significant byte, and the other bytes would be zeros. If 'A' were assigned to a character variable four characters long, the ASCII value of the letter A (hex 41) would be the variable's most significant byte because character variables are left-justified. Therefore, the two variables would not be equal, even though they held the same nominal value.

A relational expression produces a result of type LOGICAL (**.TRUE.** or **.FALSE.**). Relational expressions can use any of the operators shown in the following table to compare values. The relational operators are not case sensitive.

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

All relational operators are binary operators and appear between their operands. A relational expression cannot contain another relational expression, so there is no relative precedence or associativity among the relational operands. The following program fragment is therefore invalid:

```
REAL(4) a, b, c, d
IF ((a .LT. b) .NE. c) d = 12.0 ! Invalid expression
```

If a is less than b, then after the first part of the expression is evaluated, the expression is:

```
.TRUE. .NE. c
```

However, c is a numeric expression, and you cannot compare a numeric expression to **.TRUE.** To compare relational expressions and logical values, use the logical operators described in [Logical Expressions](#). Visual Fortran allows logical values in relational expressions. This is described in [Logical Truth](#).

Relational expressions with numeric operands may have one operand that is an integer and one that is a real number. In this case, the integer operand is converted to a real number before the relational expression is evaluated. You can also have a complex operand, in which case the other operand is first converted to complex. However, you can use only the **.NE., /=**, **.EQ.**, and **= =** operators with complex operands.

Relational expressions with character operands compare the position of their operands in the ASCII collating sequence. One operand is less than another if it appears earlier in the collating sequence. For example, the expression ('apple'.LT.'banana') returns the value **.TRUE.**, and the expression ('Keith'.GE.'Susan') returns the value **.FALSE.**

If operands of unequal length are compared, the shorter operand is extended to the length of the longer operand by the addition of blanks on the right. As an example, the expression ('Annette'.GT.'Ann') is true because it is evaluated as ('Annette'.GT.'Ann')

Logical Truth

Since Visual Fortran allows integers and logicals to be used interchangeably, it is possible to include logical values in relational expressions. A logical variable is considered equivalent to **.TRUE.** if and only if at least one bit which makes up the variable has the value 1. All bits of a logical variable are evaluated, not just the lower 8 bits. A logical variable is considered equivalent to **.FALSE.** if and

only if all bits which make up the variable have the value 0.

The result of any logical operation is either **.TRUE.**, represented by 1, or **.FALSE.**, represented by 0. The constant **.TRUE.** is represented by 1, and the constant **.FALSE.** is represented by 0.

Logical Expressions

A logical expression produces a logical value. There are seven operands used in logical expressions:

- Logical constants
- Logical variable references
- Logical array-element references
- Logical function references
- Relational expressions
- [Integer constants or variables](#)
- Logical structure component references

Other logical expressions are constructed from the operands in the preceding list by using parentheses and the logical operators in the following table. When two consecutive operations are of equal precedence, the operation is performed left-to-right.

Table: Logical Operators		
Operator	Operation	Precedence
.NOT.	Negation	1 (highest)
.AND.	Conjunction	2
.OR.	Inclusive disjunction	3
.XOR.	Exclusive disjunction	4
.EQV.	Equivalence	4
.NEQV.	Nonequivalence	4

The **.AND.**, **.OR.**, **.XOR.**, **.EQV.**, and **.NEQV.** operators are binary operators and appear between their logical expression operands. The **.NOT.** operator is unary and precedes its operand. If switch is **.TRUE.**, then (**.NOT.** switch) is **.FALSE.**.

Standard logical operators allow only arguments of the **LOGICAL** type. [Visual Fortran](#) also permits integer arguments, which can be integer constants, integer variables, integer structure components, or integer expressions. Operations are "bitwise." For example, the expression `k .XOR. m` performs an "exclusive-OR" comparison on matching bits in the operands, and sets or clears the corresponding bit in the integer value it returns. If both operands are not of the same integer kind, the lower-precision operand is converted to the higher-precision kind.

Note that the result of comparing two integer expressions with a logical operator is of **INTEGER** type, not **LOGICAL**.

The **.NOT.** operator can appear next to any of the other logical operators, but two **.NOT.** operators cannot be adjacent. The following statement, for example, is allowed:

```
logvar = a .AND. .NOT. b
```

Logical operators have the same meaning as in standard mathematical semantics. The **.OR.** disjunction operator is inclusive, while the **.XOR.** operator is exclusive. For example:

```
.TRUE. .OR. .TRUE.
```

evaluates to **.TRUE.**. However, with the exclusive operator:

```
.TRUE. .XOR. .TRUE.
```

evaluates to **.FALSE.**. The **.XOR.** and **.NEQV.** operators yield the same result; **.XOR.** is an extension.

The values of logical expressions are shown in Table 3.9.

Table: Values of Logical Expressions				
If operands a and b are:	Then the expressions below evaluate as shown:			
	a .AND. b	a .OR. b	a .EQV. b	a .XOR. b or a .NEQV. b
Both true	True	True	True	False
One true, one false	False	True	False	True
Both false	False	False	True	False

The following examples demonstrate precedence in logical expressions:

```
LOGICAL stop, go, wait, a, b, c, d, e

! The following two statements are equivalent:
stop = a .AND. b .AND. c
stop = (a .AND. b) .AND. c

! The following two statements are equivalent:
go = .NOT. a .OR. b .AND. c
go = (.NOT. a) .OR. (b .AND. c)

! The following two statements are equivalent:
wait = .NOT. a .EQV. b .OR. c .NEQV. d .AND. e
wait = ((.NOT. a) .EQV. (b .OR. c)) .NEQV. (d .AND. e)
```

The following example demonstrates the use of integers in logical expressions to perform byte masking:

```
INTEGER(2) lowerbyte, dataval, mask
mask = #00FF      ! mask the most-significant byte
dataval = #1234
lowerbyte = (dataval .AND. mask)
WRITE (*, '(1x, 2Z4)') dataval, lowerbyte
```

The output is:

```
1234 34
```

Defined Operators and Expressions

You can create functions to define custom operators, or you can extend the intrinsic operators to new

data types, by using the **INTERFACE OPERATOR ()** statement. These functions can use either one or two operands and have a defined operation name or symbol. A function and an interface block compose a defined operation. The operator can be an intrinsic operator, and the defined operation can be either unary or binary. You can extend an intrinsic operator, allowing it to accept operands of types and ranks otherwise incompatible with it.

For example, you can extend the definition of the intrinsic addition (+) symbol to operate on every variable contained in a derived type.

The form of defined operator expressions is essentially the same as for intrinsic expressions:

op *y* (unary operator)

x op y (binary operator)

op

The operator name or symbol.

x

The first binary operand.

y

The unary operand or the second binary operand.

You can specify more than one function for an operator in an interface block.

The operators <, <=, >, >=, ==, and /= always have the same interpretations as the operators **.LT.**, **.LE.**, **.GT.**, **.GE.**, **.EQ.**, and **.NE.**, respectively. If you chose to redefine <=, for example, that same redefinition would apply to **.LE.**

Defining Unary Operators

A function defining a unary operation must have the following features:

- The operation expression has the form: *op y*.
- The function is specified with a **FUNCTION** or **ENTRY** statement including one dummy argument, which must have the **INTENT(IN)** attribute.
- An interface block provides the function with a generic specification.
- The type, type parameter, rank, and shape (if it is an array) of the operand *y* matches that of the dummy argument.

Defining Binary Operators

A function defining a binary operation must have the following features:

- The operation expression has the form: *x op y*.
- The function is specified with a **FUNCTION** or **ENTRY** statement including two dummy arguments, which must have the **INTENT(IN)** attribute.
- An interface block provides the function with a generic specification.
- The types, type parameters, ranks, and shapes (if they are arrays) of the operands *x* and *y* match those of their respective dummy arguments.

The operands correspond to the dummy arguments, allowing the operands' values to be passed to the

function for evaluation.

```
INTERFACE OPERATOR (*)
  FUNCTION Boolean_And (b1, b2)
    LOGICAL, INTENT (IN) :: b1 (:), b2 (SIZE (b1))
    LOGICAL :: Boolean_And (SIZE (b1))
  END FUNCTION Boolean_And
END INTERFACE
```

This allows you to use the asterisk (*) operator, for example:

```
sensor (1:n) * action (1:n)
```

as an alternative to the function call:

```
Boolean_And (sensor (1:n), action (1:n)) ! SENSOR and ACTION are
                                           ! of type LOGICAL
```

The following example of a defined operator is from PERCENT.F90 in the /DF/SAMPLES/TUTORIAL subdirectory:

```
INTERFACE OPERATOR (.c.)
  FUNCTION CENT (x)
    REAL, INTENT (IN) :: x
    REAL cent
  END FUNCTION CENT
END INTERFACE
```

```
FUNCTION CENT (x)
REAL, INTENT (IN) :: x
REAL cent
cent = x * 100.
END FUNCTION
```

Defining Assignments

You can extend assignments by using the **INTERFACE ASSIGNMENT** statement with a subroutine you write. Each subroutine must have two nonoptional dummy arguments. The first argument must have **INTENT(OUT)** or **INTENT(INOUT)** and the second must have **INTENT(IN)**. A defined assignment is treated as a reference to the subroutine, with the left-hand side as the first argument and the right-hand side enclosed in parentheses as the second argument. The **ASSIGNMENT** generic specification specifies that the assignment operation is extended or redefined if both sides of the equal sign (=) are of the same derived type.

An example of the **ASSIGNMENT** generic specification is:

```
INTERFACE ASSIGNMENT ( = )

  SUBROUTINE Bit_To_Numeric (n, b)
    INTEGER, INTENT (OUT) :: n
    INTEGER, INTENT (IN) :: b (:)
  END SUBROUTINE Bit_To_Numeric

  SUBROUTINE Char_To_String (s, c)
    USE String_Module ! Contains definition of type
    STRING
    TYPE (STRING), INTENT (OUT) :: s ! A variable-length string
    CHARACTER (*), INTENT (IN) :: c
  END SUBROUTINE Char_To_String
```

```
END INTERFACE
```

Example assignments are:

```
kount = sensor (j:k) ! Calls Bit_To_Numeric (kount, (sensor (j:k)))
note = '89AB'       ! Calls Char_To_String (note, ('89AB'))
```

Assignment Statements

Variables are defined or redefined with assignment statements. The intrinsic assignment operator is the equal sign (=). The general form of an assignment statement is:

variable = expression

The variable and the expression can both be intrinsic types, or they can be the same derived type. The variable cannot be an assumed-size array. The variable and the expression must conform in shape.

The allowed combinations of types are listed in the following table.

Type of variable	Type of expression
Integer	Integer, real, complex
Real	Integer, real, complex
Complex	Integer, real, complex
Character	Character
Logical	Logical
Derived type	Same derived type as the variable

Both the variable and the expression can contain references to any part of the variable. For example, in the following character assignment:

```
string (2:5) = string (1:4)
```

there is no conflict in assigning the first character to the second character, assigning the second to the third, and so on. For example, if the string originally had the value 'ABCDEF', the assignment would result in the value 'AABCDF'.

If a variable is a subobject, the assignment affects only that part of the object; it does not affect the status or value of other parts. For example, assigning values to a section of an array does not alter the remainder of the array.

Character Assignments

You can make character assignments where the variable and expression strings have different lengths. If the variable string is longer than the expression, the expression string is extended to the right with blank characters to match the length of the variable. If the variable is shorter than the expression string, the expression string is truncated from the right to the length of the variable.

Differences in Kind in Assignments

If you assign an integer variable the value of an integer of higher kind, you can lose useful information if the value exceeds the limit of the lesser kind. For example:

```
int4 = 123456    ! int4 is integer(4), int2 is integer(2)
int2 = int4      ! int2 has a resulting value of -7616, not 123456
```

Assignments of real variables referencing constants or variables of a lesser kind will have the precision of the variable being defined, but the accuracy of the lesser kind. For example:

```
a = 1.11          ! a is double precision, constant is single
b = 1.11d0        ! b is double precision, constant is double
a: 1.1100000143051147 ! result is single precision accuracy
b: 1.11000000000000001 ! result is double precision accuracy
```

Derived-Type Assignments

Assignments with derived-type data objects are done on a component by component basis. You cannot use the intrinsic assignment if the variables are not of the same derived type. For example, if *c* and *d* are of the same derived type composed of a pointer *p*, integer *s*, logical *t*, character *u*, and another derived type *v*, consider the assignment:

```
c = d
```

This results in assigning *d.p* to *c.p*, *d.s* to *c.s*, *d.t* to *c.t*, *d.u* to *c.u*, and *d.v* to *c.v* using the respective intrinsic and derived-type assignments. If *d* is of a different derived type than *c*, you need to define a custom assignment routine to handle it.

You can make assignments by individual components of derived-type objects. In the preceding example, you could redefine the integer *s* with a value from *d* by using the statement:

```
c.s = d.s
```

in:

```
c = ( p, s, t, u, v )
```

For more information on structure constructors, see [Derived-Type Values](#).

Defined Assignment Statements

You can create a subroutine that defines assignment between operands of different type, in the same way you can define new custom operations. A defined assignment is characterized by a subroutine and an interface block that specifies **ASSIGNMENT** (=).

The two operands must not both be numeric, both be real, or both be character. If they are, the assignment is intrinsic. Intrinsic assignment can only be extended, it cannot be redefined.

For examples of how to create a defined assignment operator, see [Defined Operators and Expressions](#).

Defining Variables

During the course of a program, variables become defined, undefined, and redefined. The status of other associated variables also changes with them. An object with subobjects, such as an array, can only be defined when all of its subobjects are defined. Conversely, when at least one of its subobjects are undefined, the object itself, such as an array or derived type, is undefined. This section describes how variables are defined and how they become undefined or unpredictable.

A variable is *defined* when you give it a value. You can accomplish this through intrinsic assignment statements, **DATA** statements, or any I/O statement. Dummy function arguments take on the value of the corresponding actual argument when you reference a procedure; loop variables are defined during the execution of a **DO** statement. Variables that are storage-associated with defined variables also become defined.

Undefined Variables

Variables are undefined when their values are unpredictable. The most common ways variables become undefined are the following:

- Reaching an end-of-file while trying to read data. In this case, all variables specified by the input list or namelist-group of the statement become undefined.
- An allocatable array becomes undefined when it is either allocated or deallocated. (Allocatable arrays are defined when values are stored into them.) See [Arrays and Pointers](#), for information on allocatable arrays.
- When a pointer's association status becomes undefined or disassociated, the pointer becomes undefined. See [Arrays and Pointers](#), for information on pointers.
- Certain error-reporting functions report only the value for the last error, not the status of the last subroutine or function call. More information on handling errors can be found in [Handling Run-Time Errors](#) and the [IERRNO](#) entry in the *Reference*.
- During an assignment statement, all or part of the variable that precedes the equal sign (=) may be undefined. If part of the assignment expression is not evaluated, variables contained in that part may become undefined.
- Execution of an **ASSIGN** statement causes the variable in the statement to become undefined as an integer. Variables associated with that variable also become undefined.

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

When a function or subroutine is invoked, the following occurs:

- Optional dummy arguments that are not associated with real arguments are undefined.
- A dummy argument with **INTENT(OUT)** is undefined, and any actual arguments associated with it become undefined.
- A subobject of a dummy argument is undefined if the corresponding subobject of the actual argument is undefined.
- The result variable of a function is undefined.

When a **RETURN** or **END** statement in a subprogram is executed, all local variables become undefined except for the following:

- Variables with the **SAVE** attribute.

- Variables in blank (unnamed) common.
- Variables in a named common block that appears both in the subprogram and in at least one other scoping unit that makes either a direct or indirect reference to the subprogram.
- Variables accessed from the host scoping unit.
- Variables accessed from a module that also is referenced directly or indirectly by at least one other scoping unit that refers either directly or indirectly to the subprogram.
- Variables in a named common block that are initially defined and which have not been subsequently defined or redefined.

Arrays and Pointers

An *array* is a variable with at least one dimension. An array can be static or dynamic. If it is static, a fixed amount of memory storage is set aside for it at compile time and is not released until the program exits. The size of a static array cannot be changed while the program is running. If an array is dynamic, its memory storage can be assigned, altered, and removed as a program executes. Allocatable and automatic arrays are the only arrays that are dynamic.

All pointers are dynamic variables. Memory storage is allocated for them as your program runs. If a pointer is also an array, the size of each dimension can be changed while the program is running, just as with dynamic arrays. Pointers can point at arrays or at scalar variables (variables with no dimensionality). No storage space is set aside for a pointer until it is allocated with an **ALLOCATE** statement or until it is assigned to an allocated target.

Allocatable arrays are similar to pointer arrays except that they cannot point to another array. *Automatic arrays* are similar to allocatable arrays, but differ in that they are automatically allocated and deallocated for you, whenever you enter and leave (respectively) a procedure. The common feature among pointers, allocatable and automatic arrays is that they are dynamic data objects in Fortran 90.

This chapter discusses how to [specify arrays](#), how to [assign values to arrays](#), how to [use arrays](#), and how to [specify pointers and targets](#).

It also describes [array elements and sections](#), [pointer assignments](#), [dynamic association of arrays and pointers](#), [character strings](#), and [DIGITAL Fortran pointers](#).

Array Properties and Specifications

The number of dimensions in an array is called its *rank*. The total number of elements in an array is the *size* of the array. The total number of elements in a particular dimension is the *extent* of that dimension.

The *shape* of an array is determined by its rank and the extent of each dimension. For example:

```
REAL A(10, 3, 2)
```

A is an array of rank 3, with a size $10 \times 3 \times 2 = 60$ and a shape that is ten by three by two, or (10, 3, 2).

Each dimension can be specified with a *lower bound* and an *upper bound*, separated by colons (:). For example:

```
REAL B(0:9, -1:1, 4:5)
```

The first number in a dimension specification is the lower bound and the second is the upper bound. For all arrays, the lower bound can be omitted when the array is declared, and for certain arrays (assumed-shape, assumed-size and deferred-shape) the upper bound can also be omitted.

By default, the lower bound is 1, but it can be set to any positive or negative integer, or zero. In the preceding example, array B has the same rank, size, and shape as array A. However, the three

dimensions of array A all have the default lower bound of 1, while array B has specified lower bounds of 0 for the first dimension, -1 for the second dimension and 4 for the third dimension.

An array can be specified with a type declaration, a DIMENSION statement, a POINTER statement, or an ALLOCATABLE statement. For example, the following are all valid array declarations:

```

REAL          A(10, 2, 3)          ! Type declaration
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
REAL, ALLOCATABLE :: E(:, :, :)  ! Type declaration with
                                   ! ALLOCATABLE attribute
REAL, POINTER :: F(:, :)         ! Type declaration with
                                   ! POINTER attribute

```

An array's rank (number of dimensions) must always be specified. If an array is neither allocatable, a pointer nor a dummy argument, it must have its rank, size, and shape declared. There are four forms of arrays:

- Explicit-shape arrays
This form of array has all its properties specified: a fixed rank, size, and shape.
- Assumed-shape arrays
This form of array is a dummy argument that takes its shape from the actual array passed to it.
- Assumed-size arrays
This form of array is a dummy argument that takes its size from the actual array passed to it.
- Deferred-shape arrays
This form of array is either an array pointer or an allocatable array whose shape and size are determined when it is allocated or associated with an allocated target.

When a dynamic array is declared, one or more of its properties (size and shape) can be unspecified and determined later in program execution.

Explicit-Shape Arrays

An explicit-shape array has an extent specified for each of its dimensions. In addition, it can have optional lower bounds declared for some or all of its dimensions. For example, 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. Hence, 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.

The upper and lower bounds can be specified by variables or expressions in functions and subroutines. The bounds are determined by evaluating the variables or expressions at the time of entry into the subprogram. Subsequent changes in the expression variables in the subprogram have no effect on the array bounds. For example:

```

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

```

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

An explicit-shape array using variables or expressions must be a dummy argument, a function result, or an automatic array. An automatic array is an explicit-shape array that is declared in a procedure subprogram, is not a dummy argument, and has bounds that are nonconstant expressions. The arrays A and B in the example are automatic arrays.

Assumed-Shape Arrays

An assumed-shape array is a dummy argument in a subroutine or function that takes its shape from the actual array passed to it. Its rank is specified by colons (:), but the extent of each dimension is undetermined. For example:

```

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

```

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

```

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

```

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

An assumed-shape array can have optional lower bounds. In this case, the array has the shape of the actual array passed to it, but different upper and lower bounds. For example:

```

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)

```

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

Assumed-Size Arrays

An assumed-size array is a dummy argument in a subroutine or function that takes its size from the actual array passed to it. All the properties of an assumed-size array are specified (number of dimensions, extent of dimensions and bounds) except for the upper bound of the last dimension.

When an assumed-size array is declared, its last upper bound is an asterisk (*). For example:

```
SUBROUTINE ASSUME(A)
REAL A(2, 2, *)
```

An assumed-size array can differ in rank and shape from the actual array passed to it. Only its size is determined by the actual array. The elements in the actual array are associated with those in the assumed-size array in column-major order, varying the leftmost subscript fastest. The last dimension of the assumed-size array is extended to accommodate all the elements of the actual array passed to it, thus giving it the size of the actual array. For example, consider SUBROUTINE ASSUME above is called with array X:

```
REAL X(7)
CALL ASSUME(X)
```

The elements of array X correspond to A in this order:

```
X(1) = A(1, 1, 1)
X(2) = A(2, 1, 1)
X(3) = A(1, 2, 1)
X(4) = A(2, 2, 1)
X(5) = A(1, 1, 2)
X(6) = A(2, 1, 2)
X(7) = A(1, 2, 2)
```

As this example shows, the last dimension of A is not necessarily a complete dimension. The seven elements of X will not make a complete third dimension because the size of X, seven, is not divisible by the product of the specified extents of A, two-by-two. So, array A never has a determined shape. Because assumed-size arrays have no shape, the whole array cannot be accessed by passing only its name to a procedure, except in subroutines and functions that do not require shape, such as the intrinsic function LBOUND.

For example, you cannot use the intrinsic function SIZE with an assumed-size array name only. You can use **SIZE** to determine the extent along one of the fixed dimensions, in which case you must pass the dimension number along with the array name. You cannot use **SIZE** to determine the extent of the last dimension of an assumed-size array.

An assumed-size array can be broken into subsections that are entirely determined, but this is sometimes awkward. To specify all the defined elements in the example array A above, you need three subsections:

```
A(1:2, 1:2, 1) and A(1:2, 1, 2) and A(1, 2, 2)
```

Because of the undefined shape and awkward subscripts of assumed-size arrays, it is usually better to use assumed-shape arrays.

The rank of an assumed-size array is the number of fully specified dimensions plus one for the last dimension. In the above example, the rank of A is three, even though the third dimension is not complete.

You can specify lower bounds for any of the dimensions of an assumed-size array, including the last. For example:

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

Deferred-Shape Arrays

A deferred-shape array is an array pointer or an allocatable array. Its extent in each dimension is set when the array or pointer is allocated, or when the pointer is associated with an allocated target. When a deferred-shape array is declared, its rank is specified by colons (:), but the extent of each dimension is undetermined. For example:

```
REAL, ALLOCATABLE :: A(:, :, :)  
REAL, POINTER :: B(:, :)  
INTEGER, ALLOCATABLE, TARGET :: K(:)
```

Allocatable arrays and pointer arrays must be declared with deferred shape. An allocatable array is declared with an **ALLOCATABLE** statement, a **DIMENSION** statement, a **TARGET** statement, or the **ALLOCATABLE** attribute in a type declaration.

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

```
DIMENSION P(:, :, :)  
POINTER P  
  
TARGET B(:, :)  
ALLOCATABLE B
```

Until the size, shape, and bounds of a deferred-shape array are specified, no part of it can be referenced except as an argument to an intrinsic inquiry function that queries for argument presence, association status, or type properties. If the deferred-shape array is an allocatable array, its size, shape, and bounds are set in an **ALLOCATE** statement when the array is allocated. If the deferred-shape array is an array of pointers, its size, shape, and bounds are set in an **ALLOCATE** statement or in the pointer assignment statement when the pointer is associated with an allocated target. A pointer and its target must have the same rank.

For example:

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

Array Elements and Sections

A *scalar* is a single data object, such as a number or a derived type. An array is a collection of scalars. The individual scalars in the array are called elements. A scalar has rank zero. An array has at least one dimension (rank one). In DIGITAL Fortran, the maximum rank of an array is seven.

An array section is a subset of elements in an array. Array section elements can be any elements in an

array; they do not need to be consecutive or follow a regular pattern. All elements and array sections in an array have the same type and kind; for example, all INTEGER(2), all REAL(8), all character strings of the same length, or all the same derived-type.

A whole array or any part of it can be referenced and used in your program. To reference a whole array, use the array name. For example:

```
REAL A (10), B(10)
A = 3.0      ! Sets all ten values of A to 3.0
B = SQRT(A) ! Sets all ten values of B to the square root of 3.0
```

If you want to reference a particular element or array section in an array, you use subscripts to indicate the part of the array you want. For example, A(1) refers to the first element of A, and B(3:4) refers to the third and fourth elements of B. An array reference with no subscripts refers to the whole array.

You can use subscripts to specify the elements or array sections you want. For example, B(1:10:2) refers to elements 1, 3, 5, 7, and 9 of B.

See also [Array Elements](#), [Array Sections](#), [Subscript Triplets](#), and [Vector Subscripts](#).

Array Elements

Array elements are referenced by subscripts. For example:

```
REAL A(12, 8, 4)
A(1, 1, 1) = 0
```

This assigns the value 0 to the first element of array A.

In memory storage, array element $n+1$ follows array element n , $n+2$ follows $n+1$, and so on. The elements are organized in a linear sequence, even with multidimensional arrays, because computer memory has only one dimension. The leftmost array subscript is incremented first, then the next subscript to the right, and so on to the rightmost subscript. This is called column-major order. For example, the eight elements of an array dimensioned as A(2, 2, 2) are stored in memory in the following order:

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

The subscripts of an array must be separated by commas. A *subscript* is an integer constant, variable, or expression. A subscript can be positive, negative, or zero, but it must be within the bounds of the dimension it references. The number of subscript expressions must equal the number of dimensions in the array declaration. 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
```

Array Sections

You can access and use a subset of the elements of an array, called an *array section*. If you specify a specific value for each dimension the result is an array element (or simply a scalar). If you specify a triplet or vector subscript for one or more dimensions, the result is a collection of elements called a section or sub-section.

An array section is itself an array. The elements of the array section do not have to be contiguous in the array they are taken from or fit a regular pattern. A section of an array is defined by a list of subscripts, by subscript triplets and by vector subscripts. For example, if array A is dimensioned as follows:

```
REAL A(2,3,4)
```

then A(1,2,3) and A(k,m,k) are array elements while A(1:2,2,2), A(1,1,4:2:-1), and A(1,2:3, (/2,4/)) are array sections.

At least one subscript of the array section must be a triplet or a vector subscript. Subscript triplets define an array section with a regular pattern. Vector subscripts can define an array section with any pattern.

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, called the *stride*, between them. For example:

```
REAL A(10)
A(3:5:2) = 1.0   ! Triplet subscript sets elements
                 ! A(3) and A(5) to 1.0
```

The syntax for a subscript triplet is:

```
[lower-bound] : [upper-bound] [:stride]
```

The subscript triplet specifies a sequence of regularly spaced array elements. The sequence begins with the lower bound and increments (or decrements) by the stride to the last integer not greater (or less) than the upper bound. The separators between values in the triplet are colons (:). All the subscripts are optional, except when referencing the last dimension of an assumed-size array where you must include the second bound.

If you leave out the first bound, it defaults to the lower bound of the corresponding dimension; if you leave out the second bound, it defaults to the upper bound of the corresponding dimension; if you leave out the stride, it defaults to 1. If you leave out all subscripts, the section defaults to the entire extent in that dimension. For example:

```
REAL A(10)
A(1:5:2) = 3.0   ! Sets elements A(1), A(3), A(5) to 3.0
A(:5:2) = 3.0   ! Same as the previous statement
                 ! because the lower bound defaults to 1
A(2::3) = 3.0   ! Sets elements A(2), A(5), A(8) to 3.0
```

```

A(7:9) = 3.0      ! The upper bound defaults to 10
                  ! Sets elements A(7), A(8), A(9) to 3.0
                  ! The stride defaults to 1
A(:) = 3.0       ! Same as A = 3.0; sets all elements of
                  !   A to 3.0

```

In an array section of a multidimensional array, each dimension of the section can be declared with subscript triplets. If the array section is referenced in a statement or procedure, the array section must contain as many subscript entries as there are declared dimensions in the array it is a section of. For example:

```

REAL A(8, 3, 5)
A(1:2, 2:3, 4) = 3.0

```

There are three dimensions in A and three subscript entries in the array section A(1:2, 2:3, 4). The first subscript triplet (1:2) specifies the first and second element space in the first dimension of A. The second subscript triplet (2:3) specifies the second and third element space of the second dimension of A, the third subscript (4) is not a triplet, so it refers to a single element space in the third dimension. The elements of A(1:2, 2:3, 4) are:

```

element 1 = A(1,2,4)      element 3 = A(1,3,4)
element 2 = A(2,2,4)      element 4 = A(2,3,4)

```

This array section is a two-dimensional array with the shape (2, 2).

The triplet stride must not be zero. When the stride is positive, the array subsection sequence begins with the first bound and increments by the stride to the last integer not greater than the second bound. If the first bound is greater than the second bound, the sequence is empty (a zero-sized array).

When the triplet stride is negative, the sequence begins with the lower bound and decrements by the stride to the last integer not less than the upper bound. For example, for an array declared B(10), the section B(9:2:-2) is an array whose elements are: B(9), B(7), B(5), and B(3), in that order. If the upper bound is less than the lower bound, the sequence is empty.

The selected elements must be within the declared bounds of the array, but triplet values can be outside the bounds. For example, for the array B(10), the section B(3:15:6) is an array whose elements are B(3) and B(9), in that order.

Vector Subscripts

Subscript triplets specify array elements in increasing or decreasing order at the given stride, while vector subscripts specify elements in any order. A *vector subscript* is a one-dimensional array (a vector) of integer values that selects a section of a whole array. For example:

```

REAL A(10), B(5, 5)
INTEGER I(4), J(3)
! Define vector I.
I = (/5, 3, 8, 2/)
! Define vector J.
J = (/3, 1, 5/)
! Sets elements A(5), A(3), A(8) and A(2) to 3.0
A(I) = 3.0
! Sets elements B(2,3), B(2,1) and B(2,5) to 3.0
B(2,J) = 3.0

```

The values in a vector subscript must be within the declared bounds for the dimension referenced.

A *many-one array section* has a vector subscript with one or more repeated values. For example:

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

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

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

Because there are duplicate elements in A(3,K), it is a many-one array section. A many-one array section cannot appear on the left side of the equal sign (=) in an assignment statement or as an input item in a **READ** statement because the result depends on the the order of evaluation of the subscripts, which is unpredictable.

An array section with a vector subscript cannot be an internal file, the target of a pointer, or the actual argument of a dummy array if the dummy array is redefined within the subprogram. An array section with a vector subscript can be the actual argument passed to a dummy argument that is referenced only (has the INTENT(IN) attribute) and is not changed in value and does not become undefined in the subprogram.

Assigning Values to Arrays

You can assign values to arrays with ordinary assignment statements or with array constructors. An *array constructor* is a sequence of values or implied-**DO** expressions enclosed in parentheses () and slashes (/). For example:

```
INTEGER A(6)
A(1) = 1
A = (/1, 2, 3, 4, 5, 6 /)    ! The array constructor assigns values
                           ! to array A
```

The constructor sets all six elements of A to the constructor values in order: A(1)=1, A(2)=2, A(3)=3, and so on. It takes six ordinary assignment statements to assign the same values.

An array constructor has the form:

(/value-list/)

Note that no spaces are allowed between a parenthesis and the slash.

The values in the value list can be scalars, implied-**DO** loops, or arrays of any rank. All values in the list must have the same type and kind and be separated by commas. If an array appears in the value list, its elements are taken in column-major order. If you want to assign values to an array that is not a vector, you can use the function RESHAPE to put your data values into an array of the same shape. For 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 /)

```

The value list of an array constructor can also contain an implied-**DO** expression. For example:

```

INTEGER A(6)
REAL R(8)
LOGICAL L(10)
! Implied-DO.
A = (/ (I, I = 1, 6) /)
! Implied-DO with COS function.
R = (/ (COS(REAL(K)*3.1416/180.0), K=1,8) /)
! Implied-DO used to assign logical values.
L = (/ (.TRUE., N=1,5), (.FALSE., N=6,10) /)

```

In this example, $A(1)=1$, $A(2)=2$, and so on. $R(1)=\cos(1.0*3.1416/180.0)$, $R(2)=\cos(2.0*3.1416/180.0)$, and so on. The first five elements of L are set to `.TRUE.`, and the last five elements of L are set to `.FALSE.`.

In an implied-**DO**, the control variable (I, K and N in the preceding example) must be a scalar integer, but the implied-**DO** assignment can be any value or expression that is the same type as the array. In the preceding example, integer variables are assigned to array A, real expressions are assigned to array R, and logical constants are assigned to array L.

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

```

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

```

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

Array constructors are one-dimensional. Multiple array constructors can be used to define a multidimensional array, or the intrinsic function **RESHAPE** can be used. For example:

```

INTEGER B(2, 3), D(3, 6)
! Implied-DO to assign values to first row.
B(1,:) = (/ (K, K = 1, 3) /)
! List of scalars to assign values to second row.
B(2, :) = (/5, 81, 17/)
! Implied-DO plus reshaping.
D = RESHAPE ( (/ (I, I= 1,18) /), (/3, 6/) )

```

This example assigns B the values:

```

1  2  3
5 81 17

```

and assigns D the values:

```

1  4  7 10 13 16
2  5  8 11 14 17
3  6  9 12 15 18

```

The following are alternative forms for array constructors:

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

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

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

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

You can control the application of an assignment statement to an array by using a masked array assignment.

Masked Array Assignment

You can control the application of an assignment statement to an array by using a masked array assignment. With it, you apply a logical test to an array on an element-by-element basis. When the logical test is **.TRUE.**, an assignment is made for that element. For example:

```
REAL A (5)
A = (/ 89.5, 43.7, 126.4, 68.3, 137.7 /)
WHERE (A > 100.0) A = 100.0
```

In this example, the elements of **A** with values greater than 100.0 are set to 100.0. Elements with values less than or equal to 100.0 remain unchanged. After the **WHERE** statement is executed, the values of **A** are, in order:

```
89.5, 43.7, 100.0, 68.3, 100.0
```

The logical test in a **WHERE** statement, such as **A > 100.0**, is called a mask because it separates those elements that meet the test from those that don't. You can think of the mask as an array that has the same size and shape as the array in the test (array **A** in the example) containing true elements wherever the test is true and false elements wherever the test is false. In the preceding example, the mask would be:

```
FALSE FALSE TRUE FALSE TRUE
```

A masked array assignment can also contain a **WHERE** block and an **ELSEWHERE** block, rather than a single **WHERE** statement. The **WHERE** line contains the logical test. Assignments in the **WHERE** block are made only to elements in positions where the test is true. **ELSEWHERE** statements apply only to elements in positions that are not true in the **WHERE** block. The **WHERE** block and optional **ELSEWHERE** block are terminated by an **END WHERE** statement. For example:

```
REAL A (5), B(5), C(5)
A = (/ 89.5, 43.7, 126.4, 68.3, 137.7 /)
B = 0.0
C = 0.0
WHERE (A > 100.0)
  A = 100.0
  B = 2.3
```

```

ELSEWHERE
  A = 50.0
  C = -4.6
END WHERE

```

First, each assignment statement in the **WHERE** block is evaluated in order, then each assignment statement in the **ELSEWHERE** is evaluated in order. In the **WHERE** block, elements of A with values greater than 100.0 are set to 100.0. At the same time, elements in the same positions in B have their values set to 2.3. In the **ELSEWHERE** block, only elements of A and C in positions that do not correspond to true positions in the **WHERE** block are set to the new values. In the preceding example, the arrays end up with these values:

Array	Element 1	Element 2	Element 3	Element 4	Element 5
A	50.0	50.0	100.0	50.0	100.0
B	0.0	0.0	2.3	0.0	2.3
C	-4.6	-4.6	0.0	-4.6	0.0

In the **WHERE** and **ELSEWHERE** blocks, all arrays assigned values must be the same size and shape as the array in the logical test. Also, only the **WHERE** statement can be used as a labeled statement that other statements can branch to.

For information on a generalized form of masked assignment, see [FORALL](#).

Operations Using Arrays

Visual Fortran permits operations on whole arrays and array sections as a single object. For example, every element in an array can be set to the same value by equating the array name to a constant, or two conforming arrays can be added element-by-element with the addition operator (+) and their array names without subscripts:

```

REAL A (5), B(5), C(5)
A = 4.0
B = 17.0
C = A + B

```

All the arithmetic operators (+, -, *, /, **), logical operators (such as **.AND.**, **.OR.**, **.NOT.**), and relational operators (such as **.LT.**, **.EQ.**, **.GT.**), plus many intrinsic functions accept array-name arguments and perform the operation on the array element-by-element. Intrinsic functions that operate on array-name arguments are called *elemental functions*. For example:

```

REAL A (5), B(5), C(5)
INTEGER D(5)
DATA PI /3.14159265/
A = (/ ((REAL(I) * PI/180.0), I = 1, 5) /)
B = COS(A)
C = SQRT(A)
D = CEILING (A * 180.0)

```

COS, **SQRT** and **CEILING** are elemental intrinsic functions.

When two or more arrays appear in an assignment statement or expression, such as $B = \text{COS}(A)$, they must have the same size and shape. Such arrays conform to each other. For example:

```

REAL A (2, 3), B(2, 3), C(2:3, 6:8) ! All conform

```

```
REAL D (4, 5), E (5, 4), F(5, 2, 2) ! Do not conform
```

Arrays that conform have the same number of dimensions, and the extent in each dimension must be the same. The result of any attempt to combine nonconforming arrays in numeric expressions or to assign one to another is undefined.

When operating on arrays, an element-by-element assignment does not cause a conflict because the right-hand side of the assignment statement is evaluated before the assignment is made. For example, the elements of an array can be reversed without conflict:

```
REAL X (10)
X (1:10) = X (10:1:-1) ! Reversal of array elements using
                       ! subscript triplet notation
```

Array sections can be used in expressions and assignments if the specified parts conform. In this way, parts of nonconforming arrays can be used with each other. For example, a small whole array can operate with a conforming section of another array having more dimensions and greater extents:

```
REAL A (5), B(4, 7)
A = 20.0           ! Whole array assignment
B = 5.0           ! Whole array assignment
A = A - B (2, 1:5) ! Whole array and conforming section
                  ! in arithmetic expression
```

In the last statement, since the first dimension of B is a constant, it is treated as an array of rank one, and the specified section conforms with A.

There are also a number of Fortran 90 intrinsic functions that perform array operations. They are listed in the following table (square brackets [] denote optional arguments):

Intrinsic Functions for Array Operations	
Name	Description
<u>ALL</u>	ALL (<i>mask</i> [, <i>dim</i>]). Determines whether all array values meet the conditions in <i>mask</i> along (optional) dimension <i>dim</i> .
<u>ANY</u>	ANY (<i>mask</i> [, <i>dim</i>]). Determines whether any array values meet the conditions in <i>mask</i> along (optional) dimension <i>dim</i> .
<u>COUNT</u>	COUNT (<i>mask</i> [, <i>dim</i>]). Counts the number of array elements that meet the conditions in <i>mask</i> along (optional) dimension <i>dim</i> .
<u>CSHIFT</u>	CSHIFT (<i>array</i> , <i>shift</i> [, <i>dim</i>]). Performs a circular shift along (optional) dimension <i>dim</i> .
<u>DOT_PRODUCT</u>	DOT_PRODUCT (<i>vector_a</i> , <i>vector_b</i>). Performs dot-product multiplication on vectors (one-dimensional arrays).
<u>EOSHIFT</u>	EOSHIFT (<i>array</i> , <i>shift</i> [, <i>boundary</i>] [, <i>dim</i>]). Shifts elements off one end of <i>array</i> along (optional) dimension <i>dim</i> and copies (optional) <i>boundary</i> values in other end.
<u>LBOUND</u>	LBOUND (<i>array</i> [, <i>dim</i>]). Returns lower dimensional bound(s) of an array along dimension <i>dim</i> (optional).
<u>MATMUL</u>	MATMUL (<i>matrix_a</i> , <i>matrix_b</i>). Performs matrix multiplication on matrices (two-dimensional arrays).
<u>MAXLOC</u>	MAXLOC (<i>array</i> [, <i>mask</i>] [, <i>dim</i>]). 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 , meeting conditions in (optional) <i>mask</i> .
<u>MAXVAL</u>	MAXVAL (<i>array</i> [, <i>dim</i>] [, <i>mask</i>]). Returns the maximum value in an array along (optional) dimension <i>dim</i> that meets conditions in (optional) <i>mask</i> .
<u>MERGE</u>	MERGE (<i>tsource</i> , <i>fsource</i> , <i>mask</i>). Merges two arrays according to conditions in <i>mask</i> .
<u>MINLOC</u>	MINLOC (<i>array</i> [, <i>mask</i>] [, <i>dim</i>]). 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 , meeting conditions in (optional) <i>mask</i> .
<u>MINVAL</u>	MINVAL (<i>array</i> [, <i>dim</i>] [, <i>mask</i>]). Returns the minimum value in an array along (optional) dimension <i>dim</i> that meets conditions in (optional) <i>mask</i> .
<u>PACK</u>	PACK (<i>array</i> , <i>mask</i> [, <i>vector</i>]). Packs an array into a vector (one-dimensional array) of (optional) size <i>vector</i> using <i>mask</i> .
<u>PRODUCT</u>	PRODUCT (<i>array</i> [, <i>dim</i>] [, <i>mask</i>]). Returns the product of elements of an array along (optional) dimension <i>dim</i> that meet conditions in (optional) <i>mask</i> .
<u>RESHAPE</u>	RESHAPE (<i>source</i> , <i>shape</i> [, <i>pad</i>] [, <i>order</i>]). Reshapes an array with subscript <i>order</i> (optional), padded with array elements <i>pad</i> (optional).
<u>SHAPE</u>	SHAPE (<i>source</i>). Returns the shape of an array.
<u>SIZE</u>	SIZE (<i>array</i> [, <i>dim</i>]). Returns the extent of <i>array</i> along dimension <i>dim</i> (optional).
<u>SPREAD</u>	SPREAD (<i>source</i> , <i>dim</i> , <i>ncopies</i>). Replicates an array by adding a dimension.
<u>SUM</u>	SUM (<i>array</i> [, <i>dim</i>] [, <i>mask</i>]). Sums array elements along dimension <i>dim</i> (optional) that meet conditions of <i>mask</i> (optional).
<u>TRANSPOSE</u>	TRANSPOSE (<i>matrix</i>). Transpose a two-dimensional array.
<u>UBOUND</u>	UBOUND (<i>array</i> [, <i>dim</i>]). Returns upper dimensional bounds of an array along dimension <i>dim</i> (optional).
<u>UNPACK</u>	UNPACK (<i>vector</i> , <i>mask</i> , <i>field</i>). Unpacks a vector (one-dimensional array) into an array under <i>mask</i> padding with values from <i>field</i> .

Specifying Pointers and Targets

A pointer, like an allocatable array, permits data to be accessed and processed dynamically. It has no storage space allocated for it initially and must not be referenced until memory is associated with it. Memory is associated with it by assigning it to a target or by allocating memory to it. It can change targets during execution of a program.

A pointer can be a scalar, an array, a derived-type component, or a whole derived type, and it can point to scalars, arrays, derived-type components, and derived types. A pointer can also point at an array element or array subsection, but an array element or array subsection cannot be a pointer.

A pointer has the `POINTER` attribute, and can only point to an object that has the `TARGET` attribute, or to another pointer already associated with a target.

You can specify a pointer either with the `POINTER` attribute in a declaration or with a **POINTER** statement. If the pointer is an array, it must be declared with a deferred-shape specification. For example:

```
REAL, POINTER :: A (:,:) ! Type declaration with
                       ! POINTER attribute of
                       ! deferred-shape array.
```

```

REAL B, X(:, :)
POINTER B, X           ! POINTER statement declaring
                       ! scalar B and deferred-shape
                       ! array X to be pointers.

```

Likewise, you can specify a target with the **TARGET** attribute in a declaration or with the **TARGET** statement. The target can be an explicit-shape or deferred-shape array. If it is a deferred-shape array, it must also have the **ALLOCATABLE** attribute. A target cannot be an array section with a vector subscript. Some examples of specifying targets are:

```

! Type declaration with TARGET attribute of
! deferred-shape array.
REAL, ALLOCATABLE, TARGET :: C (:, :)
REAL D
REAL, ALLOCATABLE :: Y (:, :)
INTEGER K(33, 20)
! TARGET statement of scalar D, deferred-shape
! array Y and explicit-shape array K.
TARGET D, Y, K

```

An object with the **POINTER** attribute can be in a common block, but every declaration of a common block containing a pointer object must specify the same sequence of variables.

If an object has the **POINTER** attribute, it cannot have the **INTENT**, **PARAMETER** or **TARGET** attributes. If an object has the **TARGET** attribute, it cannot have the **PARAMETER** or **POINTER** attributes. Dummy arguments can have the **TARGET** attribute.

See also [POINTER](#) and [TARGET](#) in the *Reference*.

Pointer Assignments

Pointer assignment associates a pointer with an existing target. The pointer can be reassigned to other targets and it can share a target with other pointers. The [ALLOCATE](#) statement can also be used to create space for a pointer by creating a target for it.

A pointer's target must have the same type, kind, and shape as the pointer. A pointer assignment has the form of an ordinary assignment statement except the assignment symbol is `=>`. For example:

```

REAL A, X(:, :), B, Y(5, 5)
POINTER A, X
TARGET B, Y
A => B      ! Scalar pointer assignment.
X => Y      ! Array pointer assignment.

```

The pointer must have the **POINTER** attribute, and the target must have the **TARGET** or **POINTER** attribute. If the target has the **TARGET** attribute, the pointer variable is associated directly with the target. If the target has the **POINTER** attribute, the pointer variable is associated with the same target as the pointer points to, that is, with the second pointer's target.

You can assign a pointer that is currently associated with a target to a new target. In this case, the pointer is disassociated from the old target and associated with the new target. If a pointer's target is another pointer and the pointed-to pointer becomes disassociated, the pointing pointer remains associated with the pointed-to pointer but has no link to the disassociated target.

If a pointer's target is another pointer that is not associated with a target and hence undefined, the pointing pointer is also undefined. Multiple pointers can point to the same target. 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)
```

Pointers can be derived types or elements of a derived type. A pointer element within a derived type can point to the derived type. This lets you create a branching system of indexed and subindexed data, and to keep your data in hierarchical order. This data ordering is called a *linked list*. For example:

```
TYPE ORDER
  INTEGER INDEX
  TYPE(ORDER), POINTER :: NEXT
END TYPE ORDER
```

This structure can be used to branch INDEX as follows:

```
TYPE (ORDER), POINTER :: LIST
ALLOCATE(LIST)
LIST%INDEX = 1
ALLOCATE(LIST%NEXT)
LIST%NEXT%INDEX = 2
ALLOCATE(LIST%NEXT%NEXT)
LIST%NEXT%NEXT%INDEX = 3
```

The following adds a new entry to the top of this tree:

```
TYPE (ORDER), POINTER :: NEW_TOP
ALLOCATE(NEW_TOP)
NEW_TOP = ORDER(0, TREE)
```

The old first index pointed to by LIST%INDEX is now pointed to by NEW_TOP%NEXT%INDEX.

Dynamic Association of Arrays and Pointers

When you allocate storage space dynamically, the size of a variable or array is set at run-time, rather than during compilation. Dynamic allocation is done with allocatable arrays and pointers. Dynamic association can be used with scalars and arrays of any type.

When you specify an array with the ALLOCATABLE attribute, memory is not set aside for it until you specifically use an ALLOCATE statement. You can later use a DEALLOCATE statement to

disassociate memory storage with the array and free the memory space or reuse the array with a different shape for a different purpose.

When you specify a scalar or array with the **POINTER** attribute, it does not have memory set aside until it is assigned to a target that has memory space, or until it is allocated its own memory, in which case the memory space is its target. Dynamic control over the creation, association, and disassociation of pointers is provided with pointer assignment statements and with the **ALLOCATE**, **NULLIFY**, and **DEALLOCATE** statements.

Pointer assignment associates pointers with existing targets. The **ALLOCATE** statement creates targets (memory space) for pointers. The **NULLIFY** statement disassociates pointers from targets, and the **DEALLOCATE** statement deallocates targets.

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.

For more information, see [Association Status and Definition](#).

The **ALLOCATE** Statement

The **ALLOCATE** statement dynamically creates pointer targets and allocatable arrays, causing memory to be allocated to the object. An allocated object can be any named variable that has been assigned the **POINTER** or **ALLOCATABLE** attribute. For example:

```
REAL A, B(:, :), C(:), D(:, :, :)  
POINTER A, B  
ALLOCATABLE C, D  
  
READ (*, *) N, M  
ALLOCATE (A, B(N, M), C(-2:40), D(3, 3, 3))
```

When a pointer is allocated, the pointer becomes associated with the newly created space, and that space is the pointer target. Additional pointers can be associated with the same space by pointer assignment.

You can allocate a pointer that is currently associated with a target. In this case, the pointer is disassociated from the old target and associated with the new target. If you disassociate a previously allocated pointer (through reassignment or the **NULLIFY** statement), the memory space allocated to the pointer remains unavailable and is wasted.

To prevent problems, you should deallocate a pointer to disassociate it and free the memory before reassigning it.

When an array is allocated, memory is set aside to hold the size of the array you specify. The rank in the **ALLOCATE** statement must be the same as the rank of the pointer array or allocatable array

being allocated. At the time of allocation, the values of the lower and upper bounds in the **ALLOCATE** statement determine the size and shape of the array, which remain fixed as long as the array remains defined. The bounds can be any positive or negative integer, or zero. The default lower bound is 1. An allocatable array can be assigned values only after it has been allocated.

If the upper bound is less than the lower bound, the extent in that dimension is zero, and the array or pointer has zero size. Specifying a zero or negative size causes an array to be allocated, but since it has zero size, no values can be assigned to it.

You cannot allocate an array that is currently allocated. Doing so will cause a run-time error. The error status is available with the **STAT=** specifier in the **ALLOCATE** statement. If it is present, successful execution of the **ALLOCATE** statement returns zero; otherwise, it returns a positive integer. For example:

```
REAL, ALLOCATABLE :: A(:)
INTEGER ERR_ALLOC
ALLOCATE (A (5), STAT = ERR_ALLOC)
IF (ERR_ALLOC .NE. 0) PRINT *, "ALLOCATION ERROR"
```

If the **STAT=** specifier is not present and an error occurs, program execution terminates. You can use the intrinsic function **ALLOCATED** to find out if an array is already allocated. For example:

```
REAL, ALLOCATABLE :: A(:)
...
IF (.NOT. ALLOCATED(A)) ALLOCATE (A (5))
```

See also [ALLOCATE](#), [ALLOCATABLE](#), and [ALLOCATED](#) in the *Reference*.

The NULLIFY Statement

The **NULLIFY** statement disassociates a pointer from any target. You can disassociate a list of pointer objects. Each pointer object must have the **POINTER** attribute. For example:

```
INTEGER, POINTER :: A(:), B, C(:, :, : )
...
NULLIFY (A, B, C)
```

Disassociating an unassociated pointer is not an error. Pointers initially have an undefined association status. To initialize a pointer to point to no target, you can execute the **NULLIFY** statement. If a function returns a pointer, the association status of the result pointer is undefined at the start of execution. Before the function returns, it must either associate a target with this pointer or specifically define this pointer as disassociated with the **NULLIFY** statement.

See also [NULLIFY](#) in the *Reference*.

The DEALLOCATE Statement

The **DEALLOCATE** statement frees memory space associated with allocatable arrays and pointer targets and disassociates the pointers. Each object deallocated must be an allocated array or a pointer to a memory space target created with the **ALLOCATE** statement. For example:

```
REAL, POINTER :: A(:), B, C
REAL, ALLOCATABLE, TARGET :: D(:)
```

```

REAL, TARGET :: E
REAL, ALLOCATABLE :: F(:, :)
...
ALLOCATE(B, D(5), F(4, 2))
A => D
C => E
...
DEALLOCATE (B, D, F)

```

Only memory space that has been allocated with an **ALLOCATE** statement can be deallocated with a **DEALLOCATE** statement. In the preceding example, the pointer B has a memory space target created with the **ALLOCATE** statement and can be deallocated, while the pointer C is assigned to the nonallocatable target E and cannot be deallocated. Deallocating a pointer associated with an allocatable target also causes an error, for instance, if you were to attempt to deallocate the pointer A in the example above. Instead, you deallocate the target the pointer points to, array D in this example, which frees memory and disassociates the pointer.

Deallocating memory space that was not allocated with an **ALLOCATE** statement causes a run-time error. You can determine if an array or target is allocated with the ALLOCATED function, and determine the association status of a pointer with the ASSOCIATED function.

The error status is available with the **STAT=** specifier in the **DEALLOCATE** statement. If it is present, successful execution of the **DEALLOCATE** statement returns zero; otherwise, it returns a positive integer. For example:

```

REAL, ALLOCATABLE :: A(:)
INTEGER ERR_DEALL
...
DEALLOCATE (A , STAT = ERR_DEALL)
IF (ERR_DEALL .NE. 0) PRINT *, "DEALLOCATION ERROR"

```

See also ALLOCATE and *Reference: DEALLOCATE*.

Association Status and Definition

When the execution of a procedure is terminated by a **RETURN** or **END** statement, the allocation status of allocatable arrays and the association status of pointers become undefined except in the following cases:

- The array or pointer has the **SAVE** attribute.
- The pointer is the return value of a function declared to have the **POINTER** attribute.
- The array or pointer is defined in other program units or common blocks as described in Defining Variables.

The **RETURN** and **END** statements do not release the memory allocated to an array or pointer. Therefore, the memory storage allocated to an array or associated with a pointer should be deallocated before exiting the subprogram.

If the allocation status of an allocatable array becomes undefined, it cannot be referenced, defined, allocated, or deallocated. A pointer can always be nullified, allocated, or pointer assigned, but when its association status becomes undefined, it cannot be referenced or deallocated. When a pointer target becomes undefined, the pointer and the pointer association also become undefined.

The allocation status of an allocatable array is one of the following:

- Currently allocated

The array allocated was allocated by an **ALLOCATE** statement. Such an array can be referenced, defined, or deallocated.

- 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.

An allocatable array is defined when values are put into it. For example:

```
REAL, ALLOCATABLE :: A(:)
ALLOCATE (A(100))      ! A is allocated but not defined. A's
                       ! allocation status is ALLOCATED.
A(1:100) = 1          ! A is defined.
DEALLOCATE (A)        ! A is deallocated.
                       ! A's allocation status is .NOT. ALLOCATED.
```

Character Strings

Even though it is made of individual characters, a character string is considered a scalar and not an array. A character string can be an automatic object, similar to an automatic array. For example:

```
SUBROUTINE CALLCHAR (N)
  CHARACTER(N) string
```

Automatic character objects and character dummy arguments are the only character strings whose length can be dynamically set.

An array of character strings is not the same as a single character string. You can declare an array of character strings, all of the same length. Arrays of fixed-length strings can be used and accessed, allocated and deallocated, like any other array as described in the preceding sections. However, even though a character string is a scalar you can access and use substrings within it. For a description of how to specify and use character substrings, see [Substrings](#).

DIGITAL Fortran Pointers

DIGITAL Fortran pointers (also called integer pointers) differ from standard Fortran 90 pointers. An integer pointer has three components: the pointer, the pointer-based variable, and the pointed-to object.

The process of associating a pointer-based variable to the pointed-to object is two-fold. First, an integer pointer is associated with a pointer-based variable. This is done with the DIGITAL Fortran **POINTER** statement, which takes the following form:

```
POINTER (p, var) [, (p, var)]
```

The integer pointer (*p* above) cannot be explicitly typed. Its value is used as the address of *var*. The pointer-based variable (*var* above) can be any type of variable, including arrays or character

strings. The **POINTER** statement must be in the declarations section of the program or procedure.

The integer pointer is assigned memory storage associated with the pointed-to object. This is done with the **LOC** or **MALLOC** intrinsic function:

```
REAL VAR, A
POINTER (P, VAR)
P = LOC(A)
```

The integer pointer then contains the address of the pointed-to object (A in the example). Values assigned to the pointer-based variable (VAR in the example) are placed at the address held in the pointer, and hence transferred to the pointed-to object. For example:

```
REAL VAR(5), A(5)
POINTER (P, VAR)
P = LOC(A)
VAR(2) = 0.0 ! Sets A(2) to 0.0.
```

Note: You cannot use the **ALLOCATE** or **DEALLOCATE** statements with an integer pointer. You must allocate storage with **MALLOC**. After use, free the storage with the **FREE** intrinsic subroutine.

The following rules apply to integer pointers:

- Two pointers can have the same value, so pointer aliasing is allowed.
- When used directly, a pointer is treated like an integer variable. On Windows NT and Windows 95 systems, a pointer occupies one numeric storage unit, so it is a 32-bit quantity (INTEGER(4)).
- A pointer cannot be a pointer-based variable.
- A pointer cannot appear in an ASSIGN statement and cannot have the following attributes:

ALLOCATABLE	PARAMETER
EXTERNAL	POINTER
INTRINSIC	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 cannot be a function return value.

Since pointers are integer variables, normal integer arithmetic is allowed on them. They can be used anywhere an integer variable can appear. Thus, you can use integer arithmetic to modify the address. For example:

```
REAL A(10), VAR
POINTER (P, VAR)

P = LOC(A)
DO I = 1, 10
  VAR = 0.0
  P = P + 4
END DO
```

In this example, the 10 real variables in array A are set to zero by redefining the address value in the integer pointer P. The elements of A are stored consecutively in memory. The address of each real

number takes 4 bytes, so each increment of 4 to P points to the next element of A.

The following rules apply to pointer-based variables:

- A pointer-based variable is not allocated any storage. References to a pointer-based variable look to the current contents of its associated pointer to find the pointer-based variable's base address.
- A pointer-based variable cannot be data-initialized or have a record structure that contains data-initialized fields.
- A pointer-based variable can appear in only one POINTER statement.
- A pointer-based array can have fixed, adjustable, or assumed dimensions.
- A pointer-based variable cannot appear in a COMMON, DATA, EQUIVALENCE, or NAMELIST statement, and it cannot have the following attributes:

ALLOCATABLE	POINTER
AUTOMATIC	SAVE
INTENT	STATIC
OPTIONAL	TARGET
PARAMETER	

- A pointer-based variable 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 pointer-based variable is of derived type, it must be of sequence type.

See also [POINTER -- Integer](#) in the *Reference*.

Execution Control

Fortran executes statements sequentially, from first to last. You can modify this sequence by using blocks of executable procedures and by transferring control to other statements of a program. Although in theory, any Fortran statement could be considered to control execution of a program, this chapter deals primarily with statement blocks and block constructs, which include:

- Executable constructs such as **IF**, **CASE**, and **DO**
- Branching statements such as **GOTO**, **CONTINUE**, and **STOP**
- Methods of branching which are marked for obsolescence in Fortran 90, such as arithmetic **IF**, **ASSIGN**, assigned **GOTO**, and **PAUSE**

Executable Constructs and Blocks

A *construct*, also called a block construct, is a sequence of statements starting with a **CASE**, **DO**, **IF**, **WHERE**, or **FORALL** statement, and ending with an appropriate termination statement:

- CASE constructs are bounded by **SELECT CASE** and **END SELECT** statements.
- DO constructs are bounded by **DO** or **DO WHILE** and **END DO** statements.
- IF constructs are bounded by **IF** and **END IF** statements.
- WHERE constructs are bounded by **WHERE** and **END WHERE** statements.
- FORALL constructs are bounded by **FORALL** and **END FORALL** statements.

Constructs can be combined, or nested. For example, a **DO** loop can contain an **IF** construct. If you nest one block construct within another, the outer block must contain the entire construct, not just a part.

A *block* is a sequence of statements or constructs that is treated as one unit. A block can be a series of nested constructs, or it can be a set of statements within a construct. A block need not contain any executable statements.

This example shows how the terms construct and block are used:

```

IF (A .gt. 0) THEN           ! Marks the start of the outer IF construct.
  SELECT CASE (b)           ! Marks the start of the inner CASE construct.
    CASE (:0)
      . . .                 ! Any statements here form one block.
    CASE (0:)
      . . .                 ! Any statements here form another block.
  END SELECT                ! End of CASE construct.
  . . .                     ! Any code here, including the
                           ! CASE construct, form a block within the IF construct.
  . . .
END IF                       ! End of IF construct.

```

Everything between the **IF** and **END IF** form one block, including the **CASE** construct. Any statements that follow each **CASE** statement also form one block.

Some general rules apply to block constructs:

- You can transfer control within a block, or you can branch out of a block, but you may not transfer control into a block from a statement outside the block. You can exit the block from

anywhere inside it.

- Statements are executed in sequential order unless another control statement changes the order.
- You can call functions and subroutines from within a block.

For more information on constructs, see: [Naming Constructs](#).

Naming Constructs

Naming a construct is a new concept in Fortran 90. If you name a construct, the name must appear on the first statement of the construct and at the end of the construct. The name can also appear on control statements within the construct, for example, on **ELSE** and **ELSE IF** statements in an **IF** construct. The following example gives an **IF** construct the name DUTY:

```
DUTY: IF (DAY .EQ. 'SATURDAY') THEN
      CALL MOTHER(TIME, PHONE, NEWS)
END IF DUTY
```

If you are using fixed source form, the name must be placed after column six.

IF Constructs

The **IF** construct selects at most one of its blocks for execution. If there is an **ELSE** statement, at least one of the blocks within the construct is executed. Three types of **IF** clauses exist in Fortran:

- The **IF** construct is composed of blocks of code that are executed if a condition tests true
- The logical **IF** statement consists of only one statement that is executed if a condition tests true
- The arithmetic **IF** branches to other statement *labels* based on the value of an expression

The arithmetic **IF** is discussed in [Obsolescent Branching Methods](#).

The form of the **IF** construct is:

```
IF [condition] THEN
    block
[ELSE IF [condition]
    block]
[ELSE
    block]
END IF
```

The following example shows a named **IF** construct that contains another **IF** block:

```
if_construct: IF (A > 0) then
  B = C/A
  if (B > 0) then
    D = 1.0
  end if
ELSE IF (C > 0) then
  B = A/C
  D = -1.0
ELSE
```

```

    B = ABS (MAX (A, C))
    D = 0
END IF if_construct

```

An **IF** construct can contain several **ELSE IF** statements, but only one **ELSE** statement. **IF** blocks can be nested within one another. Each separate construct must be contained completely inside the adjacent outer block.

A logical **IF** statement, which controls execution of a single statement, is not a construct. The following example shows two logical **IF** statements:

```

IF (a .LT. b) temp = a      !Logical IF statement
IF (c .EQ. b) goto 100    !Logical IF statement

```

See also [IF -- Arithmetic](#), [IF -- Logical](#), and [IF Construct](#) in the *Reference*.

CASE Constructs

The **CASE** construct offers program control similar to the **IF** construct. It presents an expression for evaluation, followed by one or more blocks of code, one of which is selected to run. Value ranges to be evaluated must not overlap. The form of the **CASE** construct is:

```

[case-construct name: ] SELECT CASE (case-expression)
CASE (case-value [, case-value]...) [case-construct name]
    block
[CASE DEFAULT [case-construct name]
    block]
END SELECT [case-construct name]

```

The naming conventions that apply to the **IF** construct also apply to the **CASE** construct: if you include a case construct name in the **END SELECT** statement, it must be the same as the name given in the **SELECT CASE** statement. The same construct name can optionally appear in any **CASE** statement in the construct.

The **CASE DEFAULT** block is optional. If the case index does not match any case values listed and if there is no **CASE DEFAULT** statement, no action is taken and the program continues.

At most, one block is selected within a **CASE** construct. There is no fall-through from one block into another. After completing the block, the program exits the **CASE** construct. You do not need to explicitly exit from a block within a **CASE** construct.

When *case-expression* is evaluated, the result is called the *case index*. Values within the case selector statements can take one of four forms:

- (*case-value*)
- (*low-range*:)
- (:*high-range*)
- (*low-range*:*high-range*)

Each case value must be the same data type as the case expression. Types are limited to integer, character, and logical. For characters, the case values need not be the same length as the case

expression. Value ranges using a colon (:) are not permitted for logical data types.

If the case value range is a single value without a colon, a match occurs if the case value is equal to the case index. If only the low range is specified, a match occurs if the case index is greater than or equal to the low value; if only the high range is specified, a match occurs if the case index is less than or equal to the high range value.

A **CASE** statement cannot be the target of a branch statement. You can transfer control to an **END SELECT** statement only from within the **CASE** construct.

The following example evaluates the case expression **NET_INCOME** and defines the variable **TAX_RATE** based on its value:

```
SELECT CASE (NET_INCOME)
  CASE (50000:)
    TAX_RATE=.28
  CASE (25000:49999)
    TAX_RATE=.14
  CASE DEFAULT
    TAX_RATE=.05
END SELECT
```

In this example, the value of **NET_INCOME** is the case index. The first block is executed if the case index is greater than or equal to 50000. The second is executed if it is between 25000 and 49999. If the case index is less than 25000, the default block executes. For example, if **NET_INCOME** equals 75000, the tax rate is set to .28, and the program exits the block after the first **CASE** statement.

See also [CASE](#) and [SELECT CASE](#) in the *Reference*.

DO Constructs

The **DO** construct takes the form:

```
[name:] DO [label[,]] do-variable = lower bound, upper bound [, increment]
  [do block]
[label] END DO or CONTINUE
```

A **DO** construct begins with a **DO** statement and ends with an **END DO** or **CONTINUE**. If you do not specify a *label*, then the construct must end with **ENDDO**. If you do specify a *label*, then the **END DO** or **CONTINUE** must be identified by the same label.

Statements between **DO** and **END DO** specify the actions that are executed in the loop. **DO** blocks can contain other **DO** constructs as well as **IF** or **CASE** constructs, but any other construct must be completely contained within the **DO** block. You can break out of a **DO** loop with an **EXIT** or **CYCLE** statement.

Like other constructs, the **DO** loop can be named. If it is named, the same name must follow the **END DO**. The following example shows a standard FORTRAN 77 **DO** loop and its Fortran 90 equivalent:

```
DO 100 n = 0, stop, step
  WRITE (*,*) 'N=',n
100 CONTINUE
```

```
DO n = 0, stop, step
  WRITE (*, ) 'N=' , n
END DO
```

Loop Control

Loop control is provided in one of three ways:

- An iteration count and a **DO** variable
- Test a logical condition before each execution of the loop (**DO WHILE**)
- **DO** forever

A **DO** forever construct is the same as a **DO** construct with no loop control variable and no iteration variable. Use the **CYCLE** and **EXIT** statements to establish when to exit the **DO** forever loop.

A **DO** construct has three phases: initiation of the loop, execution of the loop range, and termination. At the initiation of a loop, the following steps take place:

1. The lower bound, upper bound, and increment are established. They should all be integers. The increment should not be zero. Data type conversion is performed, if necessary. If you do not specify an increment, the default is 1. Using default real or double-precision real in a **DO** construct is supported, but obsolescent. See [Obsolescent Branching Methods](#) for information on other Fortran statements that have been marked for obsolescence.
2. The **DO** variable is initialized with the value of the initial parameter.
3. An iteration count (also known as the trip count) is established. The iteration count is zero if the lower bound is greater than the upper bound and the increment is positive. The iteration count is also zero if the lower bound is less than the upper bound and the increment is negative.

The iteration count for the loop is calculated using the following formula:

$$\text{MAX}(\text{INT}((\text{upper bound} - \text{lower bound} + \text{increment}) / \text{increment}), 0)$$

You can increment the iteration variable inside the loop if you have not specified upper bounds, lower bounds, and an increment at the **DO** statement. Otherwise, the iteration variable cannot be modified by statements within the loop.

During the execution cycle of a **DO** construct, the following steps are performed:

- The iteration count, if any, is tested. If it is zero, the loop terminates.
- If the iteration count is nonzero, the range of the loop is executed.
- The iteration count, if any, is decremented by one, and the **DO** variable is incremented by the value of the incrementation parameter.

After the **DO** construct finishes executing, the loop control variable retains its last defined value.

For additional information on the **DO** construct, see the *Reference*.

Extended Range

A DO construct has an extended range if both of the following are true:

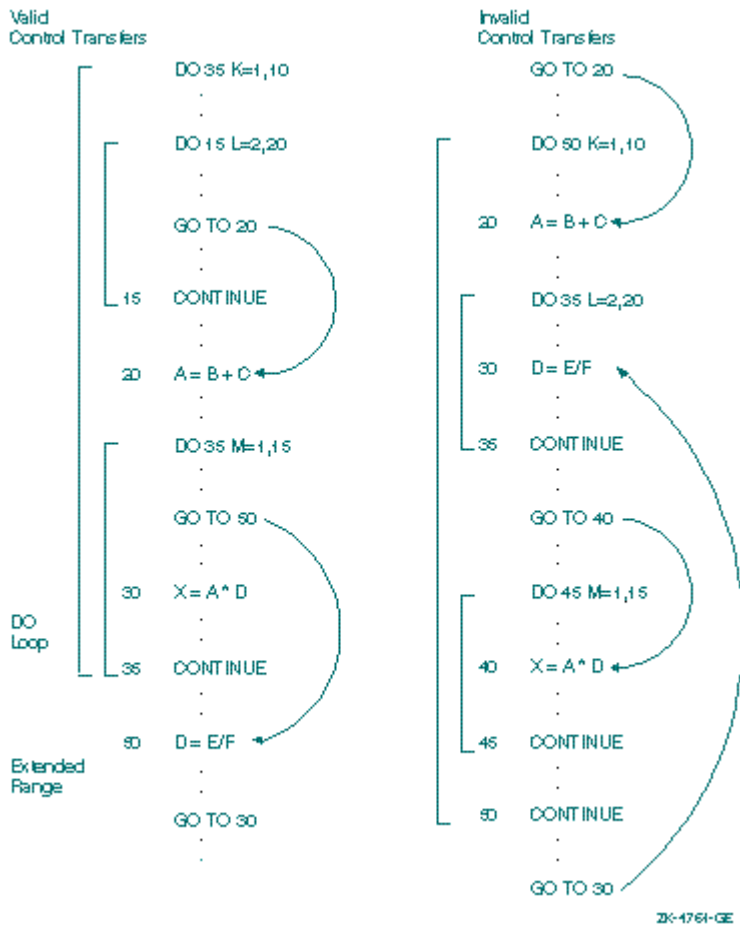
- The DO construct contains a control statement that transfers control out of the construct.
- Another control statement returns control back into the construct after execution of one or more statements.

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 illustrates valid and invalid extended range control transfers.



The CYCLE and EXIT Statements

The **CYCLE** and **EXIT** statements control how actions are performed within the loop. The **CYCLE** statement increments the loop variable while preventing the remaining statements from executing. A transfer of control to the **END DO** statement has the same effect as execution of a **CYCLE** statement. The **EXIT** statement terminates the loop completely. An **EXIT** or **CYCLE** statement within one **DO** construct can transfer control to another **DO** construct which surrounds the inner one.

The following example, CYCLE.F90, available in the /DF/SAMPLES/TUTORIAL subdirectory, shows the use of **EXIT** and **CYCLE**:

```
!   CYCLE.F90 Demonstrate the CYCLE and EXIT Statements

INTEGER i,j,k,n
PARAMETER (n = 10)                                ! Upper limit for loops.

write (*,'(/A, I2)') &
& ' Controlling loops using CYCLE and EXIT, N = ', n
write (*,900)

Loop1: DO i = 1, n
  if (i.gt.3) EXIT Loop1
  write (*,910) i

  Loop2: DO j = 1, n
    if (j.gt.2) CYCLE Loop2
    if (i.eq.2.and.j.gt.1) EXIT Loop2
    write (*,920) j

    Loop3: DO k = 1, n
      if (k.gt.2) CYCLE Loop3
      if (i.eq.1.and.j.gt.1) EXIT Loop2      ! Leave both inner loops.
      write (*,930) k

    END DO Loop3
  END DO Loop2
END DO Loop1
WRITE (*,'(/A)') ' Loops completed.'
```

```
900  FORMAT(/' Loop: 1st 2nd 3rd')
910  FORMAT(11x, i2)
920  FORMAT(21x, i2)
930  FORMAT(31x, i2)
END
```

The DO WHILE Statement

The **DO WHILE** statement provides another form of loop control, as in the following example:

```
DO WHILE (input .NE. 'n')
  WRITE (*, '(A)') 'Enter y or n: '
  READ (*, '(A)') input
END DO
```

The logical expression (input .NE. 'n' in the example) is evaluated before the program executes the code within the block. The program continues within the loop as long as the condition is true, and terminates once the condition becomes false. For example, this kind of loop control could be used to read records from a file until the end-of-file marker is reached.

No incrementation is built into the **DO WHILE** statement; anything which causes the scalar logical expression to change needs to be included in the block of code which follows. Use the **CYCLE** and **EXIT** statements to control iteration.

The same syntax, execution and loop control rules discussed in **DO** Constructs also apply to the **DO WHILE** construct.

See also [DO WHILE](#) in the *Reference*.

Branching

Branching alters the normal top-to-bottom sequence of program execution. A branch transfers control from one statement to a labeled target statement in the same program unit. Statement labels can only refer to branch target statements, **FORMAT** statements, and **DO** terminations.

A branch can transfer control out of a loop, or within the same block of a construct, but may not branch to within the range of a block construct from outside the range. Only a statement within a **CASE** construct may branch to an **END SELECT** statement. Only a statement within an **IF** construct may branch to an **END IF** statement. A branch to an **END IF** from outside the **IF** construct is an obsolescent feature.

Other forms of branching are GOTO, and CONTINUE and STOP.

GOTO and Computed GOTO

A **GOTO** statement transfers control to a target statement identified by its label, using the following syntax:

```
GOTO label
```

The *label* must appear in the same program unit as the **GOTO** statement.

Most **GOTO** statements can be replaced by other types of block construct, subroutines, or modular programming.

See also GOTO -- Computed and GOTO -- Computed in the *Reference*.

CONTINUE and STOP

The **CONTINUE** statement has no effect on program execution. The **CONTINUE** statement has been used as the terminating statement of a **DO** loop or **IF** block. The terminating statements **END DO** and **END IF** eliminate the need for **CONTINUE** statements. Support for the **CONTINUE** statement is still available for backwards compatibility, but new programs should use the block **DO** construct that terminates with **END DO**.

A **STOP** statement terminates program execution. It allows use of either a character-string message or an integer variable that provides more information about why the program ended. The default message is "Program terminated," and the default error code is zero. Your program can define other values.

In the following example, the variable `ierror`, set elsewhere in the program, detects errors; if an error exists, the program stops execution with a message:

```
IF (ierror .NE. 0) THEN  
  STOP 'ERROR DETECTED!'  
END IF
```

See also STOP in the *Reference*.

Obsolescent Branching Methods

Certain branching methods are obsolescent in the Fortran 90 standard and they may not be supported in future revisions of the standard. These obsolescent branching methods are the nonblock **DO**, assigned **GOTO**, arithmetic **IF**, and **PAUSE**.

Alternate returns have also been designated obsolescent. These are described in [Program Units and Procedures](#). For a complete list of obsolescent features, see [Obsolescent Features in Fortran 90](#).

See also [Deleted Features in Fortran 95](#), and [Obsolescent Features in Fortran 95](#).

Nonblock DO

A nonblock **DO** construct is takes the form:

```
DO label[,] [loop-control]
```

label

A statement label identifying the terminal statement.

loop-control

A DO iteration or DO WHILE statement (see [DO Constructs](#)).

The nonblock **DO** construct does not terminate with a **CONTINUE**, **GOTO**, **RETURN**, **STOP**, or **EXIT** statement. If the *do-term-action-stmt* (also called the **DO** termination) is one of these statements, the construct is considered a block **DO** construct. For example, the **DO** termination of a nonblock **DO** construct could be an assignment statement.

An example of a nonblock **DO** is:

```
      DO 100, i = 1, 100
100      a(i) = 0
```

If several **DO** loops share a single label as their termination statement, they are considered nonblock **DO** constructs. You cannot transfer control into the body of a nonblock **DO** construct or to the **DO** termination statement from outside the range of the **DO** construct.

The following example shows a common use of nonblock **DO** constructs, which is nested loops sharing a single termination statement:

```
      N = 0
      DO 100 i = 1,10
          J = i
          DO 100 k = 1,5
              L = k
100      N = N + 1      ! This statement executes 50 times.
```

ASSIGN and Assigned GOTO

The **ASSIGN** statement assigns the value of a statement label to an integer variable. It takes the form:

ASSIGN *label* TO *scalar-integer-variable*

The label must be the label of a branch target statement or **FORMAT** statement in the same program unit as the **ASSIGN** statement. You can reference the assigned variable only in an assigned **GOTO** statement or as a format specifier of an I/O statement. The *scalar-integer-variable* is a named integer of default type.

The assigned **GOTO** has the format:

GOTO *scalar-integer-variable*[,] (*label-list*)

It transfers control to the branch target statement identified by the label assigned to *scalar-integer-variable*. The value of *scalar-integer-variable* determines which label in *label-list* control transfers to. The labels listed in *label-list* need to be in the same program unit as the **GOTO** statement.

The **ASSIGN** and assigned **GOTO** statements can be replaced with a **CASE** construct that examines the value of a flag variable to indicate where to transfer control. In the case of assigned variables used as format specifiers, Fortran 90 allows character strings to be used as format specifiers.

Arithmetic IF

The arithmetic **IF** statement transfers control to up to three target statements in the same scoping unit. An example of an arithmetic **IF** statement is:

```
IF (n-10) 10,20,30
```

The branch target statements (10, 20, and 30 in the example) are executed depending on whether the value of the numeric expression is less than zero, equal to zero, or greater than zero, respectively.

Arithmetic **IF** statements can be replaced by alternate branching methods such as an **IF** construct, as in this example:

```
IF (n-10 .LT. 0) THEN
  GOTO 10
ELSE IF (n-10 .EQ. 0) THEN
  GOTO 20
ELSE
  GOTO 30
END IF
```

PAUSE

A **PAUSE** statement causes suspension of program execution. The **PAUSE** statement can take either a character constant or an integer as an argument, such as:

```
PAUSE 'Press the space bar to continue'
```

Enter a command or press Enter to return control to the program.

For more information, see [PAUSE](#) in the *Reference*.

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

Program Units and Procedures

This chapter describes how you can build a Visual Fortran program from various parts known as program units:

- [Main Program](#)
- [Modules](#)
- [Procedures](#)
- [Block Data Program Units](#)

It also discusses concepts that describe relationships between program units:

- [Association](#), which is the mechanism that allows different program units to share variables yet address them by different names without redeclaring them.
- [Scope](#), which describes the extent to which a name, whether global or local, is known.

In Fortran 90, new features such as modules allow for easy sharing of procedures and data between programs. One or more program units can use information specified in a separate module and can be dependent on that module.

For example, you can build a common block or block data unit that describes constants and variables used by a series of programs, set up a module containing commonly used functions and subroutines, and link them all to a main program which performs specific calculations.

Overview of Program Units

A program unit does not need to contain executable statements; for example, it can be a module containing interface blocks for subroutines. You can compile any of these units in separate source files and link them together later. It is not necessary to compile or recompile them as a whole. A program unit that contains internal subroutines or functions is called a *host* program. The following table defines the four types of program units:

Program Unit	Definition
Main program	The program unit that marks the beginning of execution. A main program does not have a SUBROUTINE , FUNCTION , MODULE , or BLOCK DATA statement as its first statement. A main program can have a PROGRAM statement as its first statement, but this is not required.
Procedure	A program unit that is either a subroutine or a function.
Block-data program	A program unit that provides initial values for variables in named common blocks.
Module	A program unit that contains data object declarations, type definitions, function or subroutine interfaces, and functions or subroutines accessible to other programs.

The [PROGRAM](#), [SUBROUTINE](#), [BLOCK DATA](#), [FUNCTION](#), and [MODULE](#) statements are described in detail in the *Reference*. Related information is provided in the entries for the [CALL](#), [CONTAINS](#), [INTERFACE](#), [RETURN](#), and [USE](#) statements.

A *procedure* (also referred to as subprogram) is either a function or a subroutine. Procedures can be internal, external, or module. Procedures make it easier to develop large, well-structured programs, especially in the following situations:

When you...	Procedures let you...
Have a large program	Divide a large program into parts, making the program easier to develop, test, maintain, and compile.
Intend to include certain procedures in more than one program	Create object files that contain these procedures and link them to the programs in which they are used.
Anticipate altering a procedure's implementation	Place the procedure in its own file and compile it separately. You can change the procedure, but the rest of your program need not change.

Breaking a program up into procedures provides encapsulation, so that the implementation of one routine is independent of the implementation of another. This allows changes in one routine to not affect any other routine in your program.

A *function* is a procedure invoked in an expression. It returns a value that is used in the expression. A *function result* is the value or set of values returned to the expression by the function. You invoke a *subroutine* with a **CALL** statement; it does not return a value, although it may change the values of some arguments.

Association describes how different program units share names of variables, constants, procedures, and other entities. Association can be any of the following:

- *Host association*, which allows internal subprograms to have access to named entities in a host program. These entities can be variables, constants, other procedures including interfaces, derived types, type parameters, derived-type components, and namelist groups.
- *Use association*, which is a result of a **USE** statement. A program unit that uses a module has access to all public entities in that module.
- *Argument association*, which establishes a connection between a dummy procedure argument and the corresponding actual argument.

Scope refers to the extent in which a name is recognized. Examples of scope are: an entire program, for global names; a module, for names with the **PRIVATE** attribute; or a single statement.

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.

This section also discusses program format and program execution.

Program Format

A main program is composed of the following parts, in the described sequence:

```
[ PROGRAM [ program-name ] ]
  [ specification part ]
  [ execution part ]
  [ internal subprogram part ]
END [ PROGRAM [ program-name ] ]
```

The **PROGRAM** statement takes the form:

```
PROGRAM test
```

where *test* is the optional program name.

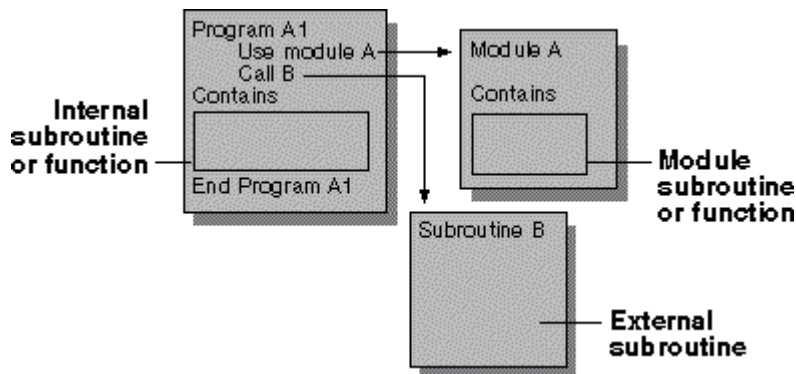
The **END** statement is the only required part of a program. If the **END** statement includes a program name, it must be the same as the one stated in the **PROGRAM** statement.

```
END PROGRAM test
```

The name of a main program is considered a global name, together with names of external procedures and common blocks. Global names must be unique within a program. For more information on global and local names, see [Names](#) and [Scope of Names](#).

The following figure shows the modular structure of a Visual Fortran program. Each shaded box represents a separate source file:

Figure: The Modular Structure of a Visual Fortran Program



The main program A1 contains internal subroutines, calls an external subroutine B, as well as modular procedures contained in A. The functions and subroutines contained in module B are called module procedures.

The specifications part of a program includes the **USE** statement, implicit declarations, parameter statements, format statements, and other declarations of variables and constants. The specifications part of a main program should not include automatic arrays and pointers. For more information on automatic arrays and pointers, see [Arrays and Pointers](#).

The specifications part of a main program cannot contain **OPTIONAL**, **INTENT**, **PUBLIC**, or **PRIVATE** statements, although you can use these statements and their corresponding attributes in external procedures, modules, or interface blocks. **OPTIONAL** and **INTENT** apply only to dummy arguments, which do not appear in a main program. The **PUBLIC** and **PRIVATE** statements are

appropriate only for modules. If you include a **SAVE** statement in a main program, it has no effect. Attributes that apply to procedures, arguments, and function results are discussed in Declaring Procedure Attributes.

Program Execution

The sequence of statements in the execution part of a main program continues from the first executable construct until either a **STOP** or **END PROGRAM** statement is encountered. You cannot include **RETURN** or **ENTRY** statements in the execution part of a main program.

A main program cannot be recursive, although functions and subroutines can call themselves recursively.

The normal execution sequence can be changed by block control statements such as **IF**, **CASE**, or **DO** constructs. For more information on block constructs and controlling the path of program execution, see Execution Control.

Modules

A module is a new feature of Fortran 90, with many uses. Modules can:

- Contain commonly used procedures
- Declare global variables and derived types
- Declare interface blocks for external procedures
- Initialize global data and global allocatable arrays
- Encapsulate data and procedures which work with that data

Modules can serve as a library of routines specific to an operating system, for example, declaring parameters unique to certain processors. Several modules could contain the same named set of functions, which operate differently on different hardware, together with type declarations specific to that hardware.

Variables formerly stored in **COMMON** blocks can be extracted and placed in named modules. Programs which use those variables can be rewritten to use the module instead having to include the **COMMON** block. This saves having multiple declarations in several program units, and ensures that variables and constants are:

- Declared consistently in all program units that use them
- Initialized with the same value

If a program unit uses only some of the variables in the module, or needs to rename some of them, that unit can specify those variables with the **ONLY** option, described in Public and Private Entities.

Using a module is similar to using the **INCLUDE** statement in a program. In each case, material from a separate file provides information necessary for a program to run. The advantages of using modules rather than **INCLUDE** statements are:

- Encapsulation of data and routines which operate on that data. Modules provide a convenient means of packaging related definitions and operations. You can combine information from

several different but related include files into one module, eliminating the need for several **INCLUDE** statements.

- Control over naming of variables, constants, and module procedures. Using modules allows you to temporarily rename constants, variables, and even procedures from program unit to program unit. For more information, see the [The USE Statement](#).
- Implicit interfaces. When you use modules, parameters and return values for module procedures are known to the main program, and matched to the calling procedure.
- Defined data types, including defined operators. Modules allow you to define special operators for derived types, and to extend intrinsic operators to other data types.
- Specifying information at a single place in the program ensures that different program units using that information will translate it consistently. For example, an **IMPLICIT** or **EXPLICIT** statement in a program unit could cause an included file to be interpreted in a different way than another program unit that includes the same file.

For more information on modules, see:

- [Module Format](#)
- [The USE Statement](#)
- [Module Names and File Names](#)
- [Uses of Modules](#)
- [Procedure Libraries](#)

Module Format

The format of a module is similar to that of a main program. A module may begin with specifications, followed by subprograms:

```
MODULE name
    [specification part]
[CONTAINS
    [module subprogram part]...
END [MODULE [name]]
```

Modules contain no executable statements, only data declarations, interface blocks, and procedure definitions. Module procedures, however, can contain executable statements.

The name of a module, like the name of a main program, is global. It must not duplicate the name of another program unit, external procedure or common block, or any local name in the module. The name of a module is limited only by the rules of Fortran and not by the names allowed in the file system. If the end module statement includes the module name, the name should match the one named initially, as in the following example:

```
MODULE DATA_MODULE
  SAVE
  REAL A(10), B, C(20,20)
  COMPLEX D(5,6)
END MODULE DATA_MODULE
```

The specification part of a module cannot contain statement function definitions, **ENTRY** statements, or **FORMAT** statements. Subprograms contained in the module, however, can use these statements in their specification part. The **CONTAINS** statement marks the delimiter between the

specifications part of a module and its subprograms. The specifications part of a module can contain the following statements:

ALLOCATABLE	PRIVATE
COMMON	PUBLIC
DATA	SAVE
DIMENSION	STATIC
EQUIVALENCE	TARGET
EXTERNAL	USE
IMPLICIT	VOLATILE
INTRINSIC	Derived type definition
NAMELIST	Interface blocks
PARAMETER	Type declaration statements
POINTER	

See also [MODULE](#) in the *Reference*.

The USE Statement

A program unit refers to modules with the **USE** statement, which is a *module reference*. A module may not reference itself, either directly or indirectly. You can control access to procedures and variables in a module by using the **PUBLIC** and **PRIVATE** statements, and the **ONLY** option. (See the following topic [Public and Private Entities](#).)

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. This means that if the **USE** statement in the preceding example appears in a subroutine, the name `XTABS` is local to the subroutine only, and will not be understood by the main program. If the **USE** statement is located in the main program, `XTABS` is valid in any of the main program's internal procedures. For more information on scope, see [Scope](#).

Public and Private Entities

The **ONLY** option in a **USE** statement allows a program to specify access to certain functions or variables in a module. When a program includes a **USE** statement, it declares access to those parameters listed. The **ONLY** option also allows public variables to be renamed, as in the following example:

```
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`.

A module can restrict access to functions or variables by using the **PRIVATE** attribute in its declarations. **PUBLIC** and **PRIVATE** may be specified as statements, or as attributes in a data declaration statement. A host program can only access public entities through the **USE** statement.

The **PRIVATE** or **PUBLIC** statement, when used alone, applies to the entire module and not simply to the variables following the statement. 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
```

The **PRIVATE** attribute causes variables and procedures to be internal to the module. A module function might use a local routine and temporary variables to arrive at its result; for example, a sorting function might use a swap routine. While the sort function itself is available to other programs, the internal swap routine and its variables are needed only by the sort function, and are therefore declared **PRIVATE** in the module.

The [example in Keynames](#) demonstrates the use of private variables in a module, and renaming of functions by a host program.

See also in the *Reference*: [PRIVATE](#), [PUBLIC](#), and [USE](#).

Module Names and File Names

The name of a module, as given in the **MODULE** statement, can be different from the name of the source file. Consider the following module, contained in a file `FOO.F90`:

```
MODULE DATA_MODULE
  SAVE
  REAL A(10), B, C(20,20)
  COMPLEX D(5,6)
END MODULE DATA_MODULE
```

A program that uses this module will contain the line:

```
USE DATA_MODULE
```

When it compiles a module, the compiler creates a file whose prefix is the module name and whose suffix is .MOD. In this example, the compiler creates a file called DATA_MODULE.MOD as well as the object file named FOO.OBJ. The .MOD file needs to exist before you can build an application that uses the module. Microsoft Developer Studio does this for you when you set up your project. For information on building a project in the Microsoft Developer Studio environment, see [Building Programs and Libraries](#).

The compiler searches for modules in the same path it searches for include files. If a source file accesses modules, the modules need to be compiled before the programs that use them. For information on setting the search path, see [Compiler and Linker Options](#) and [Building Programs and Libraries](#).

Using Modules

This section discusses how you can use modules. Most of the topics include examples:

- [Common Blocks](#)
- [Global Data](#)
- [Global Allocatable Arrays](#)
- [Procedure Libraries and Operator Extensions](#)
- [Sample Program Keynames](#)

Common Blocks

Modules can contain one or more common blocks which are accessed by a **USE** statement without renaming, as in the following example:

```
USE MY_COMMON
```

This ensures that all references to the common block are identical.

Global Data

You can create modules that contain only data objects, including derived types:

```
MODULE DATA_MODULE
  SAVE
  REAL A(10), B, C(20,20)
  INTEGER :: I = 0
  INTEGER, PARAMETER :: J = 10
  COMPLEX D (J,J)
  TYPE NONZERO
    INTEGER ONE, TWO
    REAL THREE
  END TYPE
END MODULE
```

Data objects declared in a module become global. Programs can use all the variables and constants contained in the module, or they can use some of them by specifying the **ONLY** option with the **USE** statement. To avoid name conflicts, programs can assign a local name to some or all variables, as in the following:

```
USE DATA_MODULE, AMODULE => A, DMODULE => D
```

Global Allocatable Arrays

If several programs need large global allocatable arrays whose size is not known until program execution, the array declarations can be placed in a module, then accessed by any routine which needs access to them. The following example shows the parts of this kind of program:

```
PROGRAM GLOBAL_WORK
  CALL CONFIGURE_ARRAYS
  CALL COMPUTE
END PROGRAM GLOBAL_WORK

MODULE WORK_ARRAYS
  INTEGER N
  REAL, ALLOCATABLE, SAVE :: A (:), B (:,:), C (:, :, :)
END MODULE WORK_ARRAYS

SUBROUTINE CONFIGURE_ARRAYS
  USE WORK_ARRAYS
  READ (*,*) N
  ALLOCATE (A(N), B(N,N), C(N,N,2*N))
END SUBROUTINE CONFIGURE_ARRAYS

SUBROUTINE COMPUTE
  USE WORK_ARRAYS
  . . . ! COMPUTATIONS
END SUBROUTINE COMPUTE
```

Procedure Libraries and Operator Extensions

Modules can contain interface blocks for external procedures in a library. This allows you to use argument keywords, optional arguments, and static checking of the references. You could create several modules, each containing a different version of the interface block for different applications.

Modules can also contain interface blocks which extend the use of intrinsic operator symbols (**INTERFACE OPERATOR**). For example, you can extend the addition operator (+) to specify matrix addition for type `MATRIX`, or interval arithmetic addition for type `INTERVAL`. You can define operators in external functions, either in Fortran or another language, and place only the procedure interface in the module.

For information on interface blocks, see [Procedure Interfaces](#).

Sample Program Keynames

The following example demonstrates the use of modules. The main program requests information for a database key field consisting of name and social security number. A module subroutine called `check_ssn` verifies that the social security number entered is valid.

The module also contains variables used in the main program, which does not need to redeclare them. These same variables could be used by any other program which accesses either the subroutine `check_ssn` or the database by its key. Some variables such as `sort_name` are not used by the program, but are available if needed. The module and main program are stored in two separate files, and the module `name_ssn` is compiled before the main program, `key_names`. The main program renames the routine `check_ssn`, calling it `verify` instead.

```
PROGRAM key_names
  !      input last name, first name, social security number
```

```

!      verify SSN to be a number
!      calculate ssn_key (let's pretend it's the key of a DB)
!      print out our information, get verification from user.

```

```

use name_ssn, verify => check_ssn

print *, 'Enter last name'
read *, in_lastn
print *, 'Enter first name'
read *, in_firstn
msg = 'Social security number OK'
do while (msg .NE. ' ')
  print *, 'Enter social security number'
  read *, in_ssn
  call verify (in_ssn,msg)
end do
full_name = trim(in_firstn)//' '//trim(in_lastn)
print *, full_name
file_key%ssn = in_ssn
file_key%lname = in_lastn
file_key%fname = in_firstn
print *, 'Database key is ',file_key
end program key_names

```

```

module name_ssn
type ssn_key
  character (len = 9)::ssn
  character (len = 20)::lname
  character (len = 20)::fname
end type ssn_key
character (len = 40)::full_name
character (len = 40)::sort_name
type (ssn_key) :: file_key
character (len = 1), private :: SN
character (len = 20) :: in_lastn
character (len = 20) :: in_firstn
character (len = 9) :: in_ssn
character (len = 80)::msg
integer, private :: J,K
!
contains
subroutine check_ssn (ssn, errmsg)
character (len = 9) :: ssn
character (len = 80)::errmsg
  errmsg = ' '
  do j = 1,9
    SN = ssn(j:j)
    K = ichar(SN)
    select case (K)
      case (48:57)
        errmsg = ' '
      case default
        errmsg = 'Invalid social security number'
        exit
    end select
  end do
  if (errmsg .NE. ' ') print *, errmsg
end subroutine check_ssn
end module name_ssn

```

Procedure Libraries

Visual Fortran includes several run-time library modules to assist you in special tasks:

- Run-time functions and subroutines (modules DFLIB and DFLOGM)
- Procedures to aid in porting your programs to or from other systems (module DFPORT)
- Procedures to help write foreign language programs for international markets (module DFNLS)
- Procedures to help write programs that use Component Object Model (COM) and Automation servers (modules DFCOM, DFCOMTY, and DFAUTO)

These functions and subroutines are automatically linked to your program if called, when you include the statement **USE** *module name* in your program.

You can use keywords with these procedures just as you can with the Fortran intrinsic functions. The dummy argument names to use as keywords are shown in the *Reference*.

For more information on these procedures, see [Portability Library](#), [Using National Language Support Routines](#), [Using QuickWin](#), [Using COM and Automation Objects](#), and [Introduction to the Reference in the Reference](#).

Procedures

Procedures can be either subroutines or functions. Fortran 90 includes the following types of procedures:

- *External procedures*, which are defined in an external program unit.
- *Internal procedures*, which are defined within a program unit and accessible only to that unit. The program containing the internal procedure is called the *host*. Internal procedures follow a **CONTAINS** statement.
- *Intrinsic procedures*, which are defined within the compiler, that can be used without any additional declaration or specification. (If you use an intrinsic procedure as an argument to another procedure, you need to declare it with the **INTRINSIC** statement.)
- *Module procedures*, which are defined in a module, accessible to all program units which use that module. The module containing the procedure is called the *host*.

A *procedure reference* is the method of invoking a procedure. A subroutine reference uses a **CALL** statement or sometimes a defined assignment statement. A function reference is either an expression using the name of the function or a defined operator.

Functions are invoked by reference within an expression. A *function result* is the value or set of values returned to the expression by the function. For more information, see [Functions and the RESULT Keyword](#) and [FUNCTION](#) in the *Reference*.

A *subroutine* is a program unit that can be called from other program units with a **CALL** statement. A subroutine does not directly return a value. However, values can be passed back to the calling program unit through arguments or variables known to the calling program. See the sections [Association](#) and [Scope](#) for information on how variables are shared and transferred between program units.

When invoked, a subroutine performs the set of actions defined by its executable statements. The subroutine then returns control to the statement immediately following the one that called it, or to a

statement specified as an alternate return. For more information, see [SUBROUTINE](#) in the *Reference*.

If you have defined special or extended operators, the subroutine or function that defines the operation is invoked each time you use the special operator.

Alternate returns are arguments that direct execution to jump to a labeled statement rather than to the statement immediately following the statement which called the subroutine. These have traditionally been used for error exits, but **IF** blocks and **CASE** blocks can be better alternatives.

Arguments are the variables and constants in the argument list that carry data to or from a procedure. *Actual arguments* are the entities used in the invoking procedure. *Dummy arguments* are dummy names until they are associated with actual arguments. If one of the dummy arguments is a procedure rather than a variable, it is called a dummy procedure.

A *procedure interface* determines the forms of reference through which it can be invoked. A procedure interface consists of:

- The procedure name
- A statement declaring the procedure as a subroutine or a function
- The number and characteristics of the arguments
- The characteristics of the result value, if it is a function
- Any generic identifiers associated with the procedure

Interfaces can be either *explicit* or *implicit*. Internal, module, and external procedures as well as statement functions all have explicit interfaces. External procedures have implicit interfaces by default, unless you supply an interface block for that procedure. Procedure interfaces and interface blocks are discussed in [Procedure Interfaces](#); generic procedures are discussed in [Generic Interfaces and Generic Procedures](#).

A recursive procedure can reference itself directly or indirectly.

A pure procedure is a user-defined procedure that has no side effects. An elemental procedure is a restricted form of pure procedure. It can be passed an array, which is acted upon one element at a time.

A procedure can be declared with attributes. For more information, see [Declaring Procedure Attributes](#).

The RETURN Statement can be used to terminate a procedure, but the **CASE** construct offers a better method of return.

External Procedures

External procedures are functions or subroutines that you write, which are located outside of the main program. They can be stored in a separately compiled source file, or they can be included in the main program's source code after the **END** statement, as in FORTRAN 77. External procedures can themselves contain internal functions or subroutines, as long as the internal procedures fall between a **CONTAINS** statement and the end of the procedure.

An external procedure or module procedure can contain one or more **ENTRY** statements, which

allow you to enter at some point inside the procedure. In effect, this allows you to group several related functions or subroutines that can be conditionally entered. Internal procedures cannot contain **ENTRY** statements.

The **ENTRY** Statement

The **ENTRY** statement allows you to enter a subprogram at a particular executable statement. Its form and use are similar for both a subroutine and a function. The statement includes a name for the entry point, an optional argument list, and, in the case of a function, an optional **RESULT** parameter and result name. A procedure can have one or more **ENTRY** statements.

ENTRY statements can only be used in external procedures or module procedures. Fortran 90 block constructs such as CASE provide a better way of controlling the flow of execution through a procedure.

See also ENTRY in the *Reference*.

Internal Procedures

Internal procedures are functions and subroutines that follow a CONTAINS statement in a program unit. Only the program that contains them can use the procedure. (Procedures located after the **CONTAINS** statement of a module are called module procedures.) Internal procedures are similar to external procedures except:

- Names of internal procedures are local names, not global names.
- Internal procedures cannot have an **ENTRY** statement.
- You cannot use an internal procedure as an actual argument when calling another procedure.
- They have access to host entities by host association, which means that any names declared in a main program are also known to internal functions or subroutines.

Internal procedures cannot be nested within one another. See the figure The Modular Structure of a Visual Fortran Program for a diagram of program units.

The **CONTAINS** Statement

The **CONTAINS** statement marks the delimiter between the executable portion of a program and any internal subprograms it may contain. It separates internal procedures from the host procedure, and it separates the specification part of a module from the module procedures.

If a program contains internal subprograms, they must follow a **CONTAINS** statement. Any number of internal functions or subroutines can follow the **CONTAINS** statement, but the internal procedures themselves may not contain a **CONTAINS** statement.

In FORTRAN 77, one source file could contain a program followed by several function or subroutine definitions. When it was compiled, the file made up one entire program with external functions and subroutines. You can still do this in Visual Fortran. By using the **CONTAINS** statement, you can include within a main program any procedure whose use is limited to that program. The procedure then becomes internal to the program rather than external.

The following sample 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. Sharing of data and variables between a host program and its internal procedures is described in [Association](#).

```
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
```

See also [CONTAINS](#) in the *Reference*.

Intrinsic Procedures

Intrinsic procedures are predefined by the Fortran 90 language and Visual Fortran extensions. Intrinsic procedures are automatically linked to your program.

Among other things, intrinsic procedures carry out data type conversions and return information about data types, perform operations on both numeric and character data, test for the end of the file, return addresses, and perform bit manipulation.

There are four classes of intrinsic procedures, as follows:

- Elemental procedures
- Inquiry functions
- Transformational functions
- Nonelemental procedures

In the *Reference*, the section [Introduction to the Reference](#) lists intrinsic procedures by their function. Each reference entry indicates whether the procedure is inquiry, elemental, transformational, or nonelemental, and whether it is a function or a subroutine.

An inquiry function returns a result that is dependent on the properties of its argument. An elemental procedure is specified for scalar arguments, but can be applied to array arguments. Almost all transformational functions have one or more array-valued arguments or an array-valued result. Nonelemental procedures must be called with only scalar arguments; they return scalar results. All subroutines (except **MVBITS**) are nonelemental.

Some intrinsic functions have two names, a generic name and a specific name. Generic functions accept arguments of more than one type, and the result type is the same as that of the arguments. Specific functions accept only specified argument types and return specified result types.

If you write a routine or create a variable whose name duplicates that of an intrinsic procedure, any reference to that name will invoke your routine (or your variable), not the intrinsic procedure.

This topic also discusses [using intrinsic functions](#) and [elemental intrinsic procedures](#).

The **INTRINSIC** statement is discussed in [The INTRINSIC Attribute and Statement](#).

Using Intrinsic Functions

The data type of an intrinsic function is the same as the data type of its return value. For example, the **CEILING** intrinsic function returns an integer; it is therefore an integer function. Some intrinsic functions, such as **INT**, can take arguments of more than one type, but return a particular type. Others, such as **ABS**, can return a value that has the same type as the argument.

An **IMPLICIT** statement cannot change the type of an intrinsic function. An intrinsic function name can appear in a type statement, but only if the type is the same as the standard type for that intrinsic function. If you use an intrinsic function name in a type statement that is not the same type as the standard for that function, you are declaring a new local procedure or variable. For more information on names, name duplication, and scope of names, see [Scope](#) and [Duplicating Names of Intrinsic Procedures](#).

If the arguments are of different types and kinds, or not of the type and kind specified, Visual Fortran first attempts to convert the arguments to the correct type and kind. For example, if *I* and *J* in **IDIM** (*I*, *J*) are real numbers, they are first converted to **INTEGER** type, and then the operation is performed. Or, if *I* is of type **INTEGER(2)** and *J* is of type **INTEGER(4)**, *I* is first converted to **INTEGER(4)**, and then the operation is performed. For more information on type conversion, see [Type Conversion of Numeric Operands](#).

All intrinsic procedures that take two or more integer arguments allow a mix of integer kinds. In this case, all integer arguments are promoted to the largest integer kind of all arguments, and the result is based on the larger kind values. For example:

```
integer(1) a1, b1
integer(2) a2, b2
integer(4) a4, b4
mod(a1,b4)      !a1 promoted to integer(4), result is integer(4)
mod(a2,b1)      !b1 promoted to integer(2), result is integer(2)
mod(a4,b2)      !b2 promoted to integer(4), result is integer(4)
```

When logarithmic and trigonometric intrinsic functions act on a complex argument, they return the *principal value*. The principal value of a complex number is the number whose argument (angle in radians) is less than or equal to pi and greater than -pi.

Elemental Intrinsic Procedures

An elemental intrinsic procedure operates on an array element-by-element when you pass to it the name of a whole array. Elemental procedures can simplify the way a program handles arrays. Many algorithms involving arrays can now be written conveniently as a series of computations with whole arrays. For example, if *a*, *b*, *c*, and *s* are all arrays of similar shape, then statements such as:

```
a = b + c
```

or

```
s = sum(a)
```

can replace entire **DO** loops. In the next example, since the **SIN(X)** function is an elemental

procedure, it operates element-by-element on the array `x` when you pass it the name of the whole array:

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

Many mathematical formulas can be translated directly into Fortran by use of array-processing elemental procedures.

Module Procedures

A module procedure is a procedure declared and defined in a module, between its **CONTAINS** and **END** statements. Any program that accesses the module by using the **USE** statement has access by use association to public procedures in the module.

The **MODULE PROCEDURE** statement is an optional part of an interface block. It names module procedures which can be identified by a generic name given in the **INTERFACE** statement. Generic procedures are further discussed in [Generic Interfaces and Generic Procedures](#).

See also [MODULE PROCEDURE](#) in the *Reference*.

Procedure Interfaces

Every subroutine or function has an interface, which consists of the characteristics of the procedure, its name, the nature of each dummy argument, and the procedure's generic identifiers, if any. A procedure interface can be either *explicit* or *implicit*.

In FORTRAN 77, the interface to an external function or subroutine is deduced from the form of reference to the procedure, as well as any declarations of the procedure name. Fortran 90 provides a method of making information about a procedure available through explicit interfaces. Procedures whose interfaces must be deduced, as in FORTRAN 77 are said to have *implicit interfaces*. Procedures whose interface is fully known are said to have *explicit interfaces*.

Internal, module, and intrinsic procedures as well as statement functions all have explicit interfaces. External procedures have implicit interfaces by default, unless you supply an interface block for that procedure. Specifying an interface block for a procedure is equivalent to declaring it external.

An interface block allows the compiler to confirm the correctness of subprogram calls. The compiler verifies that the number, types, and attributes of arguments in a subprogram call are consistent with those in the interface.

This topic also discusses the [INTERFACE statement](#) and [interface operator and interface assignment](#).

The INTERFACE Statement

An **INTERFACE** statement marks the beginning of an interface block. You must provide an explicit **INTERFACE** statement in the following cases:

- If the procedure has any of the following:

- An optional dummy argument
- A result that is array-valued or a pointer (functions only)
- A dummy argument that is an assumed-shape array, a pointer, or a target
- 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 defined assignment (subroutines only)
 - In an expression as a defined operator (functions only)
 - As a reference by its generic name
 - [In a context that requires it to be pure](#)
- [If the procedure is elemental](#)

An example of an interface block is:

```

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

```

See also [INTERFACE](#) in the *Reference*.

Interface Operator and Interface Assignment

The **INTERFACE** statement can also be used to define new operators, and to extend intrinsic operators as well as the assignment operator (=). For a full discussion, see [Defined Operators and Expressions](#) in *Declaring and Using Data*.

See also in the *Reference*: [INTERFACE](#) and [Assignment\(=\) -- Defined Assignment](#).

Generic Interfaces and Generic Procedures

An interface defines a generic procedure if a name follows the **INTERFACE** statement, and the interface block contains more than one procedure. For example:

```

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
  END SUBROUTINE INT_LINE_EQ
END INTERFACE

```

```

    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`.

Procedures identified with a generic name can be a mix of functions and subroutines, provided you use the name consistently. In the preceding example, if `REAL_LINE_EQ` were a function instead of a subroutine, you could call `LINE_EQUATION` as a function with real arguments, or as a subroutine with integer arguments. A compile-time error would be generated if you used `LINE_EQUATION` as a function with integer arguments.

When you are designing a set of procedures that can be referred to by one generic name, you should make sure that the generic reference is unambiguous. You can do this by making sure one of the following is true:

- At least one of the arguments is of a different type in each procedure
- The number of nonoptional arguments or their names is different for each procedure
- At least one of the arguments has a different kind type parameter for each procedure
- If the arguments are arrays, they are of a different rank (number of dimensions) in each procedure

[Resolving Procedure References](#) discusses the issue of ambiguous name references.

Dummy Procedures

If a procedure definition contains an interface block, and a subroutine or function declared in the interface block is also named as an argument to the procedure being defined, then the subroutine or function in the interface block is called a *dummy procedure*.

The following example shows a dummy procedure used as a dummy argument:

```

FUNCTION fnsam(fred,x)
INTERFACE
  SUBROUTINE fred(a,b,c)
  END SUBROUTINE
END INTERFACE
REAL x, a1, b1
!
CALL fred(a1,b1,x)
fnsam = a1
END FUNCTION

```

Subroutine `fred` can be defined in any other program unit; it is treated as an external subroutine. Since it is a dummy argument to `fnsam`, it is a dummy procedure. The rules of host and use association apply to dummy procedures, as described in [Association](#) and [Scope](#) in this chapter.

Recursive Procedures

A recursive procedure is a function or subroutine that references itself, either directly or indirectly. You must specify **RECURSIVE** in the **SUBROUTINE** or **FUNCTION** statement to use this feature.

Objects in recursive subprograms have, by default, the **AUTOMATIC attribute**. You can override this by using the **SAVE** statement or the **STATIC attribute**, either on an object-by-object basis, or for the entire subprogram (by specifying the **SAVE** statement with no object name). For more information, see [AUTOMATIC](#), [STATIC](#), and [SAVE](#) in the *Reference*.

Local variables initialized with a **DATA** statement are defined only once, regardless of the number of layers of recursion. This is equivalent to specifying **SAVE** for these variables, and so is referred to as **DATA-implied SAVE**.

An example of recursion is shown in the following code:

```
!   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
```

This example can be found in in the /DF/SAMPLES/TUTORIAL subdirectory as RECURS.F90.

See also [RECURSIVE](#) in the *Reference*.

Pure Procedures

A pure procedure is a user-defined procedure that is specified by using the prefix **PURE** (or **ELEMENTAL**) in a **FUNCTION** or **SUBROUTINE** statement. Pure procedures are a feature of Fortran 95.

A pure procedure has no side effects. It has no effect on the state of the program, except for the following:

- For functions: It returns a value.
- For subroutines: It modifies **INTENT(OUT)** and **INTENT(INOUT)** parameters.

The following intrinsic and library procedures are implicitly pure:

- All intrinsic functions
- The elemental intrinsic subroutine **MVBITS**

- The library routines in the `HPF_LIBRARY` (DIGITAL UNIX only)

A statement function is pure only if all functions that it references are pure.

For more information, see [PURE](#) in the *Reference*.

Elemental Procedures

An elemental procedure is a user-defined procedure that is a restricted form of pure procedure. An elemental procedure can be passed an array, which is acted on one element at a time. Elemental procedures are a feature of Fortran 95.

To specify an elemental procedure, use the prefix `ELEMENTAL` in a `FUNCTION` or `SUBROUTINE` statement.

For more information, see [Pure procedures](#) and [ELEMENTAL](#) in the *Reference*.

Functions and the RESULT Keyword

A function is a subprogram that has a `FUNCTION` statement as its first statement and includes the name of the function. The first statement can include an argument list, a type specification, the keyword `RECURSIVE`, and the keyword `RESULT` and result name. The last statement is either `END` or `END FUNCTION`.

A function result is usually the same name as the function itself, but you can specify a different name by using the keyword `RESULT` and result name. This is particularly useful for recursive functions. The function result variable has the function name unless a different result name is specified.

If `RESULT` is specified:

- The result of the function has the specified result name in place of the function name
- The result name cannot be the same as the function name
- The function name cannot appear in a specification statement within the function, but the function type can be specified in the header

The type specification of the function result can be either in the `FUNCTION` statement or in a type statement within the function using the result name, but both specifications cannot be used together. If you do not specify a type, the implicit typing rules apply.

If the function result is array-valued or a pointer, you must specify the name of the result variable within the function. If the function result is a pointer, the function either associates a target with the variable or its association status is disassociated.

See also [FUNCTION](#) and [RESULT](#) in the *Reference*.

The RETURN Statement

The `RETURN` statement terminates execution of a procedure and transfers control back to the calling program. It can be placed within the body of the procedure, such as in `IF` blocks for

conditional returns, and can be used more than once to provide alternate points for leaving the procedure. The statement is often placed immediately before the **END** statement, but this is not necessary because **END** performs the same function.

In a subroutine, the **RETURN** statement can include a scalar integer expression for use with alternate returns. Its value must be less than or equal to the number of asterisks (*) in the argument list. Fortran 90 provides the CASE construct, which is a better method of directing the return.

See also RETURN in the *Reference*.

Declaring Procedure Attributes

Procedures as well as data objects have attributes. Attributes that specify properties of procedures and their arguments are:

- OPTIONAL
- INTENT
- EXTERNAL
- INTRINSIC

SAVE affects variable values in procedures.

EXTERNAL and **INTRINSIC** are mutually exclusive, since a procedure is either an external or intrinsic procedure, but not both.

All attributes can be expressed in two ways: either as a statement or as an attribute in a type declaration statement. The sections which follow describe both ways of declaring the attribute.

Variable Values in Procedures

To make sure that variables you use in procedures retain their association status, allocation status, definition status, and value following execution of a **RETURN** or **END** statement, you should use the **SAVE** attribute in the module, subroutine, or function. The **SAVE** attribute has no effect when it appears in a main program. For information on what happens to variable values after exiting, see Defining Variables.

For more information on **SAVE**, see The SAVE Attribute and Statement.

The OPTIONAL Attribute and Statement

The **OPTIONAL** attribute specifies that a dummy argument need not be associated with an actual argument in a reference to a procedure. You can use the PRESENT intrinsic function to determine if an actual argument has been passed to the procedure. You can use the **OPTIONAL** attribute only in a subprogram or an interface block, and only for dummy arguments.

The **OPTIONAL** attribute can be assigned by a type declaration statement or by an **OPTIONAL** statement. The **OPTIONAL** statement has the form:

OPTIONAL [::] *dummy-argument-name-list*

An example of an **OPTIONAL** statement is:

```
SUBROUTINE EX (a, b)
OPTIONAL :: a
```

The form of the **OPTIONAL** attribute specification is:

type-spec, **OPTIONAL** [[, *attribute-spec*] ::] *dummy-argument-name-list*

An example of an **OPTIONAL** attribute specification is:

```
SUBROUTINE EX (a, b, c)
REAL, OPTIONAL :: b,c
```

This subroutine could be called with any of these 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)     !Malformed statement.
```

This results in a compile-time error.

For more information on how to call procedures with optional arguments, see [Argument Keywords](#) and [Optional Arguments](#).

See also [OPTIONAL](#) in the *Reference*.

The **INTENT** Attribute and Statement

You can specify the intended use of a dummy argument with the **INTENT** attribute to protect against unintended actions. You cannot use the attribute for a dummy procedure. Dummy procedures are discussed in [Procedure Interfaces](#).

If you do not specify the **INTENT** attribute, the argument is subject to the limitations of the associated actual argument.

An **INTENT** statement can appear only in the specification part of a subprogram or an interface body. The form of the **INTENT** statement is:

INTENT (IN | OUT | INOUT) [::] *dummy-argument-name-list*

The **INTENT** (IN) attribute specifies that a dummy argument must not be redefined or become undefined during execution of the procedure. If you attempt to redefine the dummy argument in the procedure, a compile-time error results.

The **INTENT** (OUT) attribute specifies that a dummy argument must be defined before it is used in the procedure. Any actual argument that becomes associated with it must be definable. On invocation

of the procedure, such a dummy argument becomes undefined.

The **INTENT (INOUT)** attribute specifies that a dummy argument can receive data from and return data to the invoking program unit. Any actual argument that becomes associated with the dummy argument must be definable.

An example of an **INTENT** statement is:

```
SUBROUTINE EX (a, b)
  INTENT (INOUT) :: a, b
```

The form of the **INTENT** attribute specification is:

type-spec, **INTENT** (IN | OUT | INOUT) [, *attribute-list*] :: *entity-list*

An example of an **INTENT** attribute specification is:

```
SUBROUTINE MOVE (from, to)
  USE PERSON_MODULE
  TYPE (member), INTENT (IN) :: from
  TYPE (member), INTENT (OUT) :: to
```

See also **INTENT** in the *Reference*.

The **EXTERNAL** Attribute and Statement

The **EXTERNAL** attribute specifies that a procedure is an external function or subroutine, and permits the name to be used as an actual argument. This attribute can be declared either as an attribute parameter in a type declaration or as a statement.

The form of the **EXTERNAL** statement is:

EXTERNAL *entity-list*

An example of an **EXTERNAL** statement is:

```
EXTERNAL a, b
```

The form of the **EXTERNAL** attribute specification is:

type-spec, **EXTERNAL** [[, *attribute-spec*] ::] *entity-list*

An example of an **EXTERNAL** attribute specification is:

```
REAL, EXTERNAL :: a, b
```

If you include an interface block for an external procedure, you do not need to specify the **EXTERNAL** attribute for it. For information on interface blocks, see [Procedure Interfaces](#).

See also **EXTERNAL** in the *Reference*.

The **INTRINSIC** Attribute and Statement

The **INTRINSIC** attribute specifies that an object is the name of an intrinsic function, and lets you

use the name of an intrinsic function as an actual argument in another procedure.

Only intrinsic functions with specific names can appear with the **INTRINSIC** attribute and be used as actual arguments in calls to a procedure. The reference must be to the specific intrinsic name, not the generic name.

The object cannot have been given the **EXTERNAL** attribute previously. This attribute can be declared either as an attribute parameter in a type declaration or as a statement.

The form of the **INTRINSIC** statement is:

INTRINSIC *entity-list*

An example of an **INTRINSIC** statement is:

```
INTRINSIC sin, cosine
```

The form of the **INTRINSIC** attribute specification is:

type-spec, **INTRINSIC** [[, *attribute-spec*] ::] *entity-list*

An example of an **INTRINSIC** attribute specification is:

```
REAL, INTRINSIC :: sin, cosine
```

This declaration allows the `sin` and `cosine` intrinsic functions to be passed as an argument to another procedure.

Certain specific function names cannot be used; these are indicated in [Functions Not Allowed as Actual Arguments](#) in the *Reference*.

See also [Intrinsic Procedures](#) and [INTRINSIC](#) in the *Reference*.

Arguments

An *argument* is a means of passing data back and forth between the program unit invoking a procedure, and the procedure itself.

This section also discusses [argument keywords](#) and [optional arguments](#).

Host and use association features let you use variables in a procedure without having to pass them as arguments. For more information, see [Association](#) and [Scope](#).

You can also use modules to avoid having to pass arguments to a procedure. For more information, see the example in [Sample Program Keynames](#).

Argument Keywords

The concept of *argument keywords* is new with Fortran 90. An argument keyword is simply the name of the dummy argument in the procedure argument list. Keywords can be used with dummy arguments from the called procedure to explicitly associate a dummy argument with an actual argument. They can be used to associate dummy and actual arguments that are not listed in the same

sequence, or when optional arguments have been omitted from the argument list.

All procedures can be invoked with either positional arguments (arguments arranged in positional sequence) or argument keywords. A keyword is required for an argument *only* if a preceding optional argument is omitted or if the argument sequence is changed.

When you use a keyword in a procedure call, use an equal sign (=) between it and the constant or variable actual argument. It has the same form as a simple assignment statement, as shown in the next example.

All of the following references to the intrinsic function **CMPLX** are valid:

```
! Positional arguments.
CMPLX (first, second, m)
! Keywords specified.
CMPLX (y = second, kind = m, x = first)
! An optional argument is skipped.
CMPLX (first, kind = m)
```

In the intrinsic function CMPLX, the dummy arguments are *x*, *y*, and *kind*, as shown in the *Reference*. The arguments *y* and *kind* are optional for this function.

You can use the PRESENT intrinsic function to determine if an optional argument has been passed to a procedure.

Entries in the *Reference* for intrinsic functions show the dummy variable names in the argument list. Those names can be used as keywords.

Optional Arguments

Some dummy arguments can be optional. Every actual argument in the calling procedure must have a corresponding dummy argument in the called procedure. However, you can include dummy arguments in the dummy argument list that do not have corresponding actual arguments. If any dummy arguments are optional, you must specify the **OPTIONAL** attribute when declaring the argument. For more information, see The OPTIONAL Attribute and Statement.

Within a procedure, you can use the PRESENT intrinsic function to test whether or not an optional argument has been specified.

An example of how to use optional arguments with a keyword follows:

```
CALL SOLVE (fun, sol, print = 6)

SUBROUTINE Solve (Funct, solution, method, strategy, print)
INTERFACE
  FUNCTION Funct (x)
    REAL Funct, x
  END FUNCTION Funct
END INTERFACE
REAL solution
INTEGER, OPTIONAL :: method, strategy, print
. . .
```

In this example, when the interface is specified by an interface block, the name of the last argument must be **PRINT**. (Note also that *Funct* in this example is a dummy procedure.)

Block Data Program Units

A block-data subprogram is a program unit that defines initial values for variables in named common blocks. It contains no executable statements, only data specifications and initial values. The format of a block data program unit is:

```
BLOCK DATA [name]
    [specifications]
END [BLOCK DATA [name]]
```

Variables are normally initialized with **DATA** statements. Variables in named common blocks can be initialized once in the block-data subprogram or in one routine, or they can be initialized exactly the same way in all routines. Variables in the blank (unnamed) common block cannot be initialized in block-data subprograms. A better programming practice is to use modules rather than block data program units to declare variables and their initial values.

A block data module can contain the following statements:

COMMON	PARAMETER	USE
DATA	POINTER	Derived-type definitions
DIMENSION	RECORD	Record structure declarations
EQUIVALENCE	SAVE	Type declaration statements
IMPLICIT	STATIC	
INTRINSIC	TARGET	

Type declaration statements in a block data program unit must *not* contain the following attribute specifiers:

- ALLOCATABLE
- **AUTOMATIC**
- EXTERNAL
- INTENT
- OPTIONAL
- PRIVATE
- PUBLIC
- **VOLATILE**

Even if you do not initially define all objects in a named common block, you must specify them if you are defining at least one object in that common block. You can initialize objects in more than one common block in one block data program unit.

A block data program unit can initially define only nonpointer objects. Any object associated with an object in a common block is considered to be in that common block.

An example of a block data program unit is:

```
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
```

See also BLOCK DATA in the *Reference*.

Association

Association allows different program units to access the same value through different names. There are three kinds of association:

- Name association
- Pointer association
- Storage association

Pointer association is discussed in Dynamic Association of Arrays and Pointers. Storage association is discussed in Storage Association.

This section discusses name association. Names can be associated by argument, use, or host association.

Argument Association

Arguments are the values passed to and from functions and subroutines through calling argument lists. An actual argument is the specific variable, expression, array, function name, or other item passed to a subroutine or function when it is called. Within the called procedure, the argument is a dummy argument.

The reference to a procedure establishes an association between a dummy argument and an actual argument with a different name. If a subroutine has three dummy arguments X, Y, and Z, it can be called in several ways:

```
! Conventional method.
CALL EXT1 (A, B, C)
! One dummy name is specified.
CALL EXT1 (A, B, Z=C)

! 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
```

Argument A is associated with dummy argument X either by its location within the parentheses, or by explicit assignment, as in the fifth line of the example. 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.

If an actual argument is a constant, a function reference, or an expression other than a single variable, you cannot assign a value to the corresponding dummy argument. If you do, the result is unpredictable.

If an actual argument is an expression, it is evaluated first, before association of the dummy and actual arguments. If an actual argument is an array element, its subscript expressions are evaluated before the association. The subscript expressions remain constant throughout the execution of the subroutine or function, even if they contain variables that receive new values during the execution.

The following table shows how actual and dummy arguments can be associated:

When the actual argument is	The dummy argument is
A variable, an array element, a derived-type component, or an expression	A variable name
An array or an array element	An array name
An array	A variable
An alternate-return specifier (<i>*label</i>) in the CALL statement	An asterisk (*)
The name of an external or intrinsic procedure	Any unique name that is used in a subroutine call or function reference within the procedure

When the actual argument is an array or array element, the number and size of its dimensions can be different from those of the dummy argument, but any reference to the dummy array must be within the limits of the memory sequence in the actual array. A reference to an out-of-bounds element is not detected as an error, and has unpredictable results.

When the argument is an array, each element is passed to the procedure one element at a time, and the procedure is executed once for each element. The procedure must be declared in an **INTERFACE** statement, or it must be an intrinsic function, if arrays will be passed to a scalar dummy argument.

When the argument is an external procedure or an intrinsic one, the actual argument must be declared with the **EXTERNAL** statement, or it must be an intrinsic function declared with an **INTRINSIC** statement. (Certain intrinsic functions cannot be used this way; see [Functions Not Allowed as Actual Arguments](#) in the *Reference*.)

Use Association

The sharing of named entities between a module and the program using it is called *use association*. Through use association, procedures and data object that are public in the module are known and definable in the program using the module.

The **USE** statement allows a program to access entities defined in modules, such as the following:

- Named data objects
- Derived types
- Interface blocks
- Procedures
- Generic identifiers
- Namelist groups

Because use association expands the constants, variables, and procedures available to a using program, it is possible to encounter duplication of names. If a program uses several modules, and

entities in some modules duplicate names of entities in another module, then the using program must do one of the following:

- Not refer to either entity which shares a duplicate name
- Rename one or both entities so there is no longer any name duplication

The local name of a module entity (such as a variable or procedure name) used in a program must not duplicate another local name accessible to the program. If a function or subroutine in a module duplicates the name of an intrinsic Fortran procedure, then the intrinsic procedure is not available.

For more information on local and global names, see [Scope](#). For information on how to rename module entities, see [The USE Statement](#).

Host Association

Any program unit that contains an internal procedure is said to be the *host* for that internal procedure. Host association allows a host program, its internal procedures, module subprograms, and derived-type definitions to access the same entities. Host association remains in effect throughout the execution of the program.

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

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

In this example, the variables `a`, `b`, and `c` are available to the internal subroutine `find` through host association. They do not have to be passed as arguments to the internal procedure. In fact, if they are, they become local variables to the subroutine and hide the variables declared in the host program.

Conversely, the host program knows the value of `c`, when it returns from the internal subroutine that has defined `c`.

Scope

The Fortran 90 standard defines names in terms of *scoping units*. A scoping unit is a program or part of a program in which a name is defined and known. It can be an entire program, a program unit, a single statement, or a part of a statement. Scope defines the extent to which a name is recognized, whether the name is a constant, a variable, a procedure, an operator, or any other kind of name.

If a variable is used outside of its scope, it is undefined. See [Undefined Variables](#) for other instances of when variables become undefined.

The following sections discuss the concept of scope as it applies to:

- [Global, local, and statement names](#)
- [Resolving references to names](#)

Scope of Names

Program Structure, Characters, and Source Forms introduced the concept of [names](#). A name can be applied to a program, subprogram, variable, array, dummy argument, named constant, derived type, or block construct. There are three kinds of names: [global](#), [local](#), and [statement](#).

This section also discusses [duplicating names of intrinsic procedures](#), [common block names](#), [function result names](#), [derived-type component names](#), the [scope of argument keywords](#), and the [scope of other entities](#): labels, I/O units, operators and assignment symbols.

Global Names

Global names identify program units, common blocks, and external procedures. They are recognized anywhere in a program, so they can be defined only once within a program. For example, if you use a subroutine named Sort in one program, you cannot also have a common block or a function named Sort in that program.

Local Names

Local names identify variables (both scalar and array), constants, named constructs, statement functions, internal procedures, module procedures, dummy procedures, intrinsic procedures, generic identifiers, derived types, and namelist group names. Components of a derived data type and argument keywords (dummy arguments) are also examples of local names. Local names hide global names and other local names in the same program unit. (Exceptions are [argument keywords](#), [generic names](#), and [common block names](#).) The following example shows local names in a module:

```
MODULE test1
CHARACTER (len=1), private :: S
CHARACTER (len=9) SSN

CONTAINS

SUBROUTINE print_part
DO j = 1,9
  S = ssn(j:j)
  print *, S
END DO
END SUBROUTINE

END MODULE
```

The following line declares S and SSN as dummy arguments in:

```
print_part:

SUBROUTINE print_part (S, SSN)
```

In this case, S and SSN become variables local to the subroutine, and hide the variables S and SSN declared in the rest of the module. In order to recognize the variables of the declarations part of the module, they cannot be stated as arguments to the subroutine.

If a name is local to one program unit, the same name can be used as either a local or a global entity

in other program units. Resolving Procedure References explains how the compiler resolves duplicate or ambiguous name references.

Duplicating Names of Intrinsic Procedures

Since Fortran does not reserve keywords as in other languages, you can create variables, constants, or procedures that have the same name as Fortran intrinsic procedures. Once you do this, however, the original intrinsic function is no longer accessible. For example, the following code defines a new function `sin`:

```
SUBROUTINE sub
  . . .
CONTAINS
  FUNCTION sin(x)
  . . .
  END FUNCTION sin
END SUBROUTINE sub
```

Any references to `sin` in subroutine `sub` invokes the internal function, not the intrinsic function of the same name.

Similarly, any type declaration that names an intrinsic procedure, giving it a different type than the intrinsic procedure's standard type, creates a new local name. The following example declares a variable called `sin`:

```
CHARACTER (len = 10) sin
```

Any program or internal procedure that has access to this character variable can no longer use the intrinsic function of the same name. If this variable is declared in a module with the `PRIVATE` attribute, however, then any program unit outside the module still can use the intrinsic `SIN` function.

Statement Names

A statement name is a name whose scope is one statement. Statement names can appear in statement function statements, or in an implied-`DO` of a `DATA` statement or an array constructor. The name of a variable that appears as a dummy argument in a statement function statement has a scope of the statement in which it appears. The scope of the `DO` variable, which must be of integer type, is the implied-`DO` list.

```
DIMENSION x(10)
Add (a,b) = a + b
DO n = 1,10
  x(n) = add(y,z)
END DO
```

In this example, the scope of `a` and `b` is limited to the statement function itself. The scope of `n` is the `DO` loop.

Common Block Names

A common block name is a global name. Since local names may duplicate global names, any reference to that name identifies the local entity, except when the reference occurs in a `COMMON` or `SAVE` statement. When a common block is named in a `SAVE` statement, it should be enclosed in slashes (`/`) to distinguish it from any local variables with the same name, as in the following

example:

```
COMMON /ralph/ fred, ethel, lucy
character(20) ralph
SAVE /ralph/      !Common block is saved, not the variable
```

The name may not be used for named constants, intrinsic procedures, or a local variable that is also an external function in a function subprogram. If two program units do not reference one another, the same name can be used for a common block in one, and an intrinsic procedure name in the other.

Function Result Names

The other instance when duplicate names are allowed occurs with function results. For each **FUNCTION** statement or **ENTRY** statement in a function subprogram, there is a result variable. If there is no **RESULT** clause specifying a different variable name, the result variable has the same name as the function being defined.

Derived-Type Component Names

If a variable is a component of another name, it has the same scope as its parent variable. A derived type declared in a module, for example, has the scope of the module itself and any program units that use it. The components of the derived type are recognized in the same program units the entire derived type is. The same is true for a component of an array; it has a valid, recognized value only in the same program units as the original array of which it is a part.

Components of a derived type can duplicate the name of other local variables, since the type components never appear without a qualifier. For example:

```
TYPE FOO
  INTEGER IJK
  CHARACTER L
END TYPE FOO
REAL IJK
TYPE (FOO) :: SAMPLE
```

In this example, the integer component of `SAMPLE` is referred to as `SAMPLE%IJK`. Any references to `IJK` alone, outside of the **TYPE...END TYPE** declaration, refer to the real variable, not the derived type, and there is no ambiguity.

Scope of Argument Keywords

Argument keywords are local entities. A dummy argument name is known only to its host, which can be an internal procedure, module procedure, or procedure interface block that defines the argument. If the procedure or procedure interface block becomes accessible to another program unit through the **USE** or **CONTAINS** statements, the argument keyword is also accessible to the same program unit.

Dummy arguments of intrinsic procedures are local keywords in the program units that make reference to the procedure. Dummy arguments of internal procedures, module procedures, or procedure interface blocks have a scope of the host program unit, or other program units accessible by use or host association. As a keyword, the dummy argument name can only appear in the procedure reference.

For information on how to invoke procedures using keywords, see [Argument Keywords](#).

Other Special Cases

Other entities such as labels, I/O units, operators and assignment symbols have scope as well. The following rules apply:

- Labels are always considered to be local. No two statements in the same scoping unit can have the same label.
- Intrinsic operators (such as +, -, *, **, or /) are global, but user-defined operators are local entities. The scope of the special operator is the same as the scope of the procedure that defines it. You can make a user-defined operator either global or generic by using a procedure interface block. For information on creating procedure interface blocks, see [Defined Operators and Expressions](#).
- The assignment symbol (=) is a global entity. You can identify additional generic assignment operations in an interface block, as described in [Defined Operators and Expressions](#).

Resolving Procedure References

Because Fortran allows duplication of names, you must make sure that references to names are unambiguous. The compiler takes the following steps to determine the correct reference for a name:

1. It looks for an interface block. If there is one, it looks for a generic name for the interface block. If the generic name matches, the procedure is generic (see below, [Resolving Generic References](#)).
2. If the generic name does not match, or if the interface block does not have one, the compiler looks for procedures defined in the interface block. If it finds a matching procedure, it is specific, not generic.
3. Does the procedure have the INTRINSIC attribute? If so, then the intrinsic procedure definition tells whether the reference is generic or specific.
4. Is the name declared as an EXTERNAL procedure? If so, the procedure is specific (see below, [Resolving Specific References](#)).

If a match has not yet been found, steps 1 through 4 are repeated for any module the program uses, and for any containing host program. If a match still has not been found, the compiler continues with the following steps:

- It looks for a dummy argument to match the name of the procedure
- It looks for an intrinsic procedure with that name

If the procedure still has not been identified, the compiler assumes it is an external procedure.

Resolving Generic References

Once a procedure is determined to be generic, the compiler checks the interface block to find the specific one that matches by checking:

- Whether it is a subroutine or a function
- The number and characteristics of the arguments
- The characteristics of the result value, if it is a function

To insure that a generic reference is unambiguous, you need to make sure that every procedure which uses that generic name satisfies at least one of the following:

- One procedure is a subroutine, and the other is a function, making the calling method different
- There is a different number of non-optional arguments for each procedure
- Corresponding dummy arguments for each procedure are of different type, have different kind type parameters, or ranks (if they are arrays)

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`.

Resolving Specific References

Only a generic name can refer to more than one procedure. If two separate non-generic procedures have the same name, then they must have a different scope. For more information, see [Scope of Names](#).

Files, Devices, and I/O Hardware

This chapter discusses Visual Fortran files and devices, and using your input/output (I/O) hardware. Together with the sections on I/O statements and I/O editing, these sections explain where and how Fortran data is input and output. Files and devices are where data is stored and retrieved, I/O editing determines how the data is organized when it is read or written, and I/O statements determine what input/output operations are performed on the data. This section is organized as follows:

- [Devices and Files](#)
- [I/O Hardware](#)

Devices and Files

In Fortran's I/O system, data is stored and transferred among files. All I/O data sources and destinations are considered files. Devices such as the screen, keyboard and printer are external files, as are data files stored on a device such as a disk.

Variables in memory can also act as a file on a disk, and are typically used to convert ASCII representations of numbers to binary form. When variables are used in this way, they are called internal files.

The discussion of I/O files is divided into two sections:

- [Logical Devices](#)
- [Files](#)

Logical Devices

Every file, internal or external, is associated with a logical device. You identify the logical device associated with a file by a unit specifier (**UNIT=**). The unit specifier for an internal file is the name of the character variable associated with it. The unit specifier for an external file is either a number you assign with the **OPEN** statement, a number preconnected as a unit specifier to a device, or an asterisk (*).

External unit specifiers that are preconnected to certain devices do not have to be opened. External units that you connect are disconnected when program execution terminates or when the unit is closed by a **CLOSE** statement.

A unit must not be connected to more than one file at a time, and a file must not be connected to more than one unit at a time. You can **OPEN** an already opened file but only to change some of the I/O options for the connection, not to connect an already opened file or unit to a different unit or file. See [Changing I/O Specifications with OPEN](#) for a description and example.

You must use a unit specifier for all I/O statements except in the following three cases:

- **PRINT**, which always writes to standard output (**UNIT= 6**)
- **READ** statements that contain only an I/O list and format specifier, which read from standard input (**UNIT= 5**)
- **INQUIRE** by file, which specifies the filename, rather than the unit with which the file is

associated

External Files

A unit specifier associated with an external file must be either an integer expression or an asterisk (*). The integer expression must be in the range 0 (zero) to a maximum value of 2,147,483,640. The following example connects the external file `UNDAMP.DAT` to unit 10 and writes to it:

```
OPEN (UNIT = 10, FILE = 'undamp.dat')
WRITE (10, '(A18,\)') ' Undamped Motion:'
```

The asterisk (*) unit specifier specifies the keyboard when reading and the screen when writing. The following example uses the asterisk specifier to write to the screen:

```
WRITE (*, '(1X, A30,\)') ' Write this to the screen.'
```

Visual Fortran has four units preconnected to external files (devices), as shown in the following table.

External Unit Specifier	Description
Asterisk (*)	Always represents the keyboard and screen
0	Initially represents the keyboard and screen
5	Initially represents the keyboard
6	Initially represents the screen

The asterisk (*) specifier is the only unit specifier that cannot be reconnected to another file, and attempting to close this unit causes a compile-time error. Units 0, 5, and 6, however, can be connected to any file with the **OPEN** statement. If you close unit 0, 5, or 6, it is automatically reconnected to its respective device the next time an I/O statement attempts to use that unit.

The following example writes to the preconnected unit 6 (the screen), then reconnects unit 6 to an external file and writes to it, and finally reconnects unit 6 to the screen and writes to it:

```
      REAL a, b
!   Write to the screen (preconnected unit 6).
      WRITE(6, '(\'\' This is unit 6\'')')
!   Use the OPEN statement to connect unit 6
!   to an external file named 'COSINES'.
      OPEN (UNIT = 6, FILE = 'COSINES', STATUS = 'NEW')
      DO a = 0.1, 6.3, 0.1
         b = COS (a)
!   Write to the file 'COSINES'.
         WRITE (6, 100) a, b
100      FORMAT (F3.1, F5.2)
      END DO
!   Close it.
      CLOSE (6)
!   Reconnect unit 6 to the screen, by writing to it.
      WRITE(6, '(\'\' Cosines completed\'')')
      END
```

Internal Files

The unit specifier associated with an internal file is a character string or character array. There are two types of internal files:

- An internal file that is a character variable, character array element, or noncharacter array element that has exactly one record, which is the same length as the variable, array element, or noncharacter array element.
- An internal file that is a character array, a character derived type, or a noncharacter array that is a sequence of elements, each of which is a record. The order of records is the same as the order of array elements or type elements, and the record length is the length of one array element or the length of the derived-type element.

Follow these rules when using internal files:

- Use only formatted I/O, including I/O formatted with a format specification and list-directed I/O. (List-directed I/O is treated as sequential formatted I/O.) Namelist formatting is not allowed.
- If the character variable is an allocatable array or array part of an allocatable array, the array must be allocated before use as an internal file. If the character variable is a pointer, it must be associated with a target.
- Use only **READ** and **WRITE** statements. You cannot use file connection (**OPEN**, **CLOSE**), file positioning (**REWIND**, **BACKSPACE**) or file inquiry (**INQUIRE**) statements with internal files.

You can read and write internal files with **FORMAT** I/O statements or list-directed I/O statements exactly as you can external files. Before an I/O statement is executed, internal files are positioned at the beginning, before the first record.

With internal files, you can use the formatting capabilities of the I/O system to convert values between external character representations and Fortran internal memory representations. That is, reading from an internal file converts the ASCII representations into numeric, logical, or character representations, and writing to an internal file converts these representations into their ASCII representations.

This feature makes it possible to read a string of characters without knowing its exact format, examine the string, and interpret its contents. It also makes it possible, as in dialog boxes, for the user to enter a string and for your application to interpret it as a number.

If less than an entire record is written to an internal file, the rest of the record is filled with blanks.

In the following example, `x` and `fname` specify internal files:

```

CHARACTER(10) str
INTEGER n1, n2, n3
CHARACTER(14) fname
INTEGER i

str = " 1  2  3"
! List-directed READ sets n1 = 1, n2 = 2, n3 = 3.
  READ(str, *) n1, n2, n3
  i = 4
! Formatted WRITE sets fname = 'FM004.DAT'.
  WRITE (fname, 200) i
200  FORMAT ('FM', I3.3, '.DAT')
```

Files

Fortran supports two methods of file access (sequential and direct) and three kinds of file structure (formatted, unformatted, and [binary](#)). Sequential-access and direct-access files can have any of the three file structures. The following kinds of files are possible:

- [Formatted Sequential](#)
- [Formatted Direct](#)
- [Unformatted Sequential](#)
- [Unformatted Direct](#)
- [Binary Sequential](#)
- [Binary Direct](#)

Each kind of file has advantages and the best choice depends on the application you are developing:

- **Formatted Files**

You create a formatted file by opening it with the **FORM='FORMATTED'** option, or by omitting the FORM parameter when creating a sequential file. The records of a formatted file are stored as ASCII characters; numbers that would otherwise be stored in binary form are converted to ASCII format. Each record ends with the ASCII carriage return (CR) and line feed (LF) characters.

If you need to view a data file's contents, use a formatted file. You can load a formatted file into a text editor and read its contents directly, that is, the numbers would look like numbers and the strings like character strings, whereas an unformatted or binary file looks like a set of hexadecimal characters.

- **Unformatted Files**

You create an unformatted file by opening it with the **FORM='UNFORMATTED'** option, or by omitting the FORM parameter when creating a direct-access file. An unformatted file is a series of records composed of physical blocks. Each record contains a sequence of values stored in a representation that is close to that used in program memory. Little conversion is required during input/output.

The lack of formatting makes these files quicker to access and more compact than files that store the same information in a formatted form. However, if the files contain numbers, you will not be able to read them with a text editor.

- **Binary Files**

You create a binary file by specifying **FORM='BINARY'**. Binary files are the most compact, and good for storing large amounts of data.

- **Sequential-Access Files**

Data in sequential files must be accessed in order, one record after the other (unless you change your position in the file with the **REWIND** or **BACKSPACE** statements). Some methods of I/O are possible only with sequential files, including nonadvancing I/O, list-directed I/O, and namelist I/O. Internal files also must be sequential files. You must use sequential access for files associated with sequential devices.

A sequential device is a physical storage device that does not allow explicit motion (other than reading or writing). The keyboard, screen, and printer are all sequential devices.

- Direct-Access Files

Data in direct-access files can be read or written to in any order. Records are numbered sequentially, starting with record number 1. All records have the length specified by the **RECL=** option in the **OPEN** statement. Data in direct files is accessed by specifying the record you want within the file. If you need random access I/O, use direct-access files. A common example of a random-access application is a database.

All files are composed of records. Each record is one entry in the file. It can be a line from a terminal or a logical record on a magnetic tape or disk file. All records within one file are of the same type.

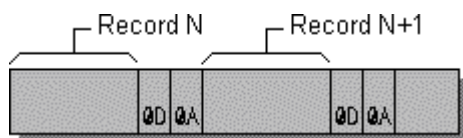
In Fortran, the number of bytes written to a record must be less than or equal to the record length. One record is written for each unformatted **READ** or **WRITE** statement. A formatted **READ** or **WRITE** statement can transfer more than one record using the slash (/) edit descriptor.

For binary files, a single **READ** or **WRITE** statement reads or writes as many records as needed to accommodate the number of bytes being transferred. On output, incomplete formatted records are padded with spaces. Incomplete unformatted and binary records are padded with undefined bytes (zeros).

Formatted Sequential Files

A formatted sequential file is a series of formatted records written sequentially and read in the order in which they appear in the file. Records can vary in length and can be empty. They are separated by carriage return (0D) and line feed (0A) characters as shown in the following figure.

Figure: Formatted Records in a Formatted Sequential File



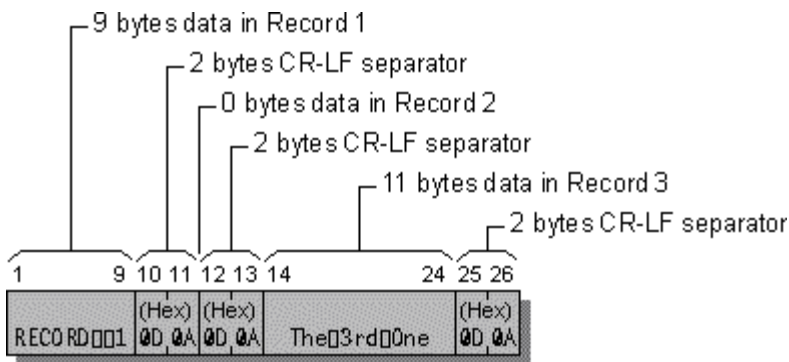
An example of a program writing three records to a formatted sequential file is given below. The resulting file is shown in the following figure.

```

OPEN (3, FILE='FSEQ')
! FSEQ is a formatted sequential file by default.
WRITE (3, '(A, I3)') 'RECORD', 1
WRITE (3, '()')
WRITE (3, '(A11)') 'The 3rd One'
CLOSE (3)
END

```

Figure: Formatted Sequential File



Formatted Direct Files

In a formatted direct file, all of the records are the same length and can be written or read in any order. The record size is specified with the **RECL=** option in an **OPEN** statement and should be equal to or greater than the number of bytes in the longest record.

The carriage return (CR) and line feed (LF) characters are record separators and are not included in the **RECL=** value. Once a direct-access record has been written, you cannot delete it, but you can rewrite it.

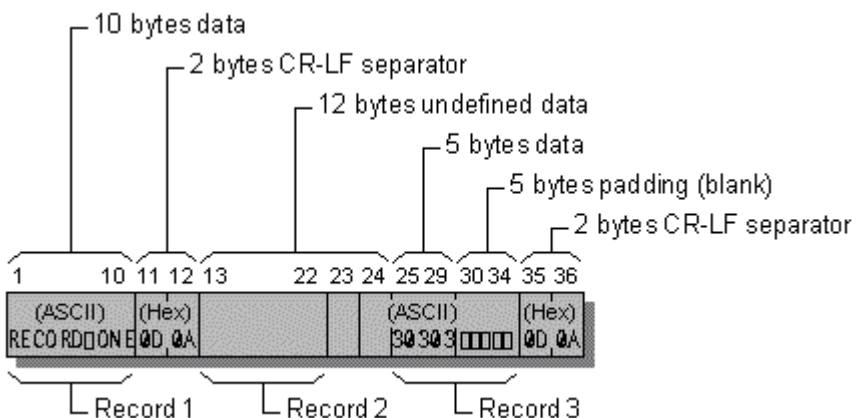
During output to a formatted direct file, if data does not completely fill a record, the compiler pads the remaining portion of the record with blank spaces. The blanks ensure that the file contains only completely filled records, all of the same length. During input, the compiler by default also pads the input if the input list and format require more data than the record contains.

You can override the default blank padding on input by setting **PAD='NO'** in the **OPEN** statement for the file. If **PAD='NO'**, the input record must contain the amount of data indicated by the input list and format specification. Otherwise, an error occurs. **PAD='NO'** has no effect on output.

An example of a program writing two records, record one and record three, to a formatted direct file is given below. The result is shown in the following figure.

```
OPEN (3, FILE='FDIR', FORM='FORMATTED', ACCESS='DIRECT', RECL=10)
WRITE (3, '(A10)', REC=1) 'RECORD ONE'
WRITE (3, '(I5)', REC=3) 30303
CLOSE (3)
END
```

Figure: Formatted Direct File



Unformatted Sequential Files

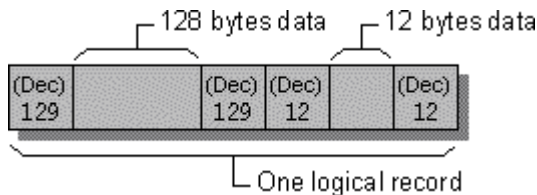
Unformatted sequential files are organized slightly differently on different platforms. This section describes unformatted sequential files created by Visual Fortran. If you are accessing files from another platform that organizes them differently, see [Converting Unformatted Numeric Data](#) or you can use the conversion utility in the \DF\SAMPLES\TUTORIAL subdirectory called UNFSEQ.F90.

The records in an unformatted sequential file can vary in length. Unformatted sequential files are organized in chunks of 130 bytes or less called *physical blocks*. Each physical block consists of the data you send to the file (up to 128 bytes) plus two 1-byte "length bytes" inserted by the compiler. The length bytes indicate where each record begins and ends.

A *logical record* refers to an unformatted record that contains one or more physical blocks. (See the following figure.) Logical records can be as big as you want; the compiler will use as many physical blocks as necessary.

When you create a logical record consisting of more than one physical block, the compiler sets the length byte to 129 to indicate that the data in the current physical block continues on into the next physical block. For example, if you write 140 bytes of data, the logical record has the structure shown in the following figure.

Figure: Logical Record in Unformatted Sequential File



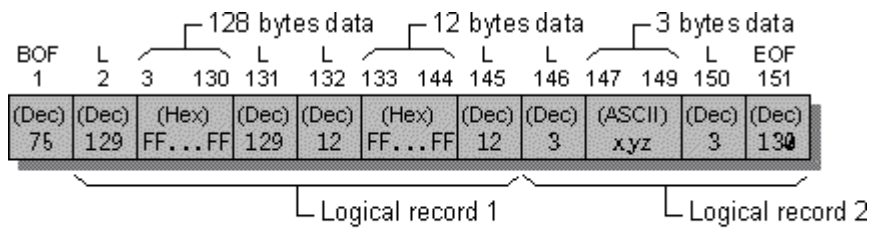
The first and last bytes in an unformatted sequential file are reserved; the first contains a value of 75, and the last holds a value of 130. Fortran uses these bytes for error checking and end-of-file references.

The following program creates the unformatted sequential file shown in the following figure:

```
! Note: The file is sequential by default
!       -1 is FF FF FF FF hexadecimal.
!
CHARACTER xyz(3)
INTEGER(4) idata(35)
DATA      idata /35 * -1/, xyz /'x', 'y', 'z'/

!
! Open the file and write out a 140-byte record:
! 128 bytes (block) + 12 bytes = 140 for IDATA, then 3 bytes for XYZ.
OPEN (3, FILE='UFSEQ',FORM='UNFORMATTED')
WRITE (3) idata
WRITE (3) xyz
CLOSE (3)
END
```

Figure: Unformatted Sequential File



BOF Beginning-of-file byte (75 decimal)
 L Physical-block-length byte (0 ≤ L ≤ 129)
 EOF End-of-file byte (130 decimal)

Unformatted Direct Files

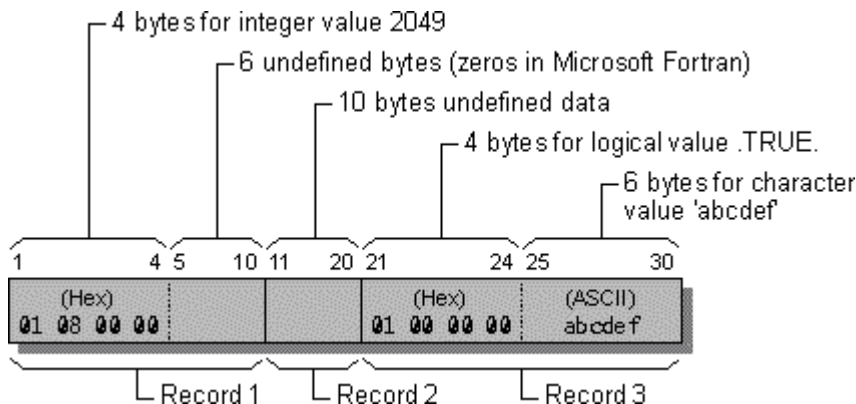
An unformatted direct file is a series of unformatted records. You can write or read the records in any order you choose. All records have the same length, given by the **RECL=** specifier in an **OPEN** statement. No delimiting bytes separate records or otherwise indicate record structure.

You can write a partial record to an unformatted direct file. Visual Fortran pads these records to the fixed record length with ASCII NULL characters. Unwritten records in the file contain undefined data.

The following program creates the sample unformatted direct file shown in the following figure:

```
OPEN (3, FILE='UFDIR', RECL=10,&
      & FORM = 'UNFORMATTED', ACCESS = 'DIRECT')
WRITE (3, REC=3) .TRUE., 'abcdef'
WRITE (3, REC=1) 2049
CLOSE (3)
END
```

Figure: Unformatted Direct File



Binary Sequential Files

A binary sequential file is a series of values written and read in the same order and stored as binary numbers. No record boundaries exist, and no special bytes indicate file structure. Data is read and written without changes in form or length. For any I/O data item, the sequence of bytes in memory is the sequence of bytes in the file.

The next program creates the binary sequential file shown in the following figure:

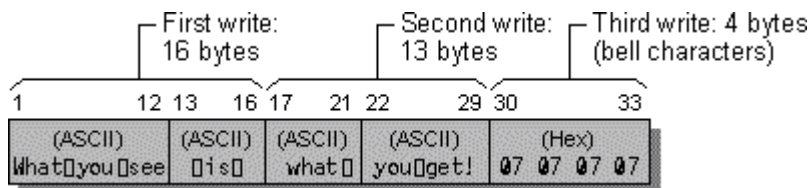
```
! NOTE: 07 is the bell character
! Sequential is assumed by default.
```

```

!
INTEGER(1) bells(4)
CHARACTER(4) wys(3)
CHARACTER(4) cvar
DATA bells /4*7/
DATA cvar /' is ',wys /'What',' you',' see'/

OPEN (3, FILE='BSEQ',FORM='BINARY')
WRITE (3) wys, cvar
WRITE (3) 'what ', 'you get!'
WRITE (3) bells
CLOSE (3)
END

```

Figure: Binary Sequential File

Binary Direct Files

A binary direct file stores records as a series of binary numbers, accessible in any order. Each record in the file has the same length, as specified by the **RECL=** argument to the **OPEN** statement. You can write partial records to binary direct files; any unused portion of the record will contain undefined data.

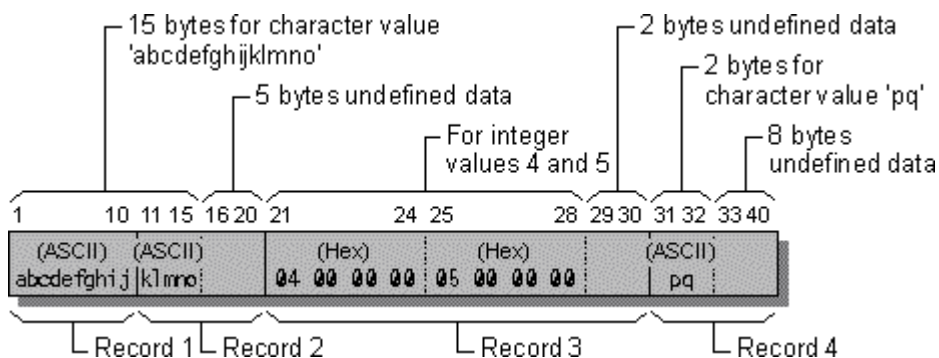
A single read or write operation can transfer more data than a record contains by continuing the operation into the next records of the file. Performing such an operation on an unformatted direct file would cause an error. Valid I/O operations for unformatted direct files produce identical results when they are performed on binary direct files, provided the operations do not depend on zero padding in partial records.

The following program creates the binary direct file shown in the following figure:

```

OPEN (3, FILE='BDIR',RECL=10,FORM='BINARY',ACCESS='DIRECT')
WRITE (3, REC=1) 'abcdefghijklmno'
WRITE (3) 4,5
WRITE (3, REC=4) 'pq'
CLOSE (3)
END

```

Figure: Binary Direct File

I/O Hardware

Most of your hardware configuration and setup is done through your computer's operating system. To connect and communicate with your printer, for example, you should read your system and printer manuals. This section describes how Visual Fortran refers to physical devices and shortcuts for printing text and graphics from Microsoft Developer Studio.

For more information, see:

- [Printing](#)
- [Physical Devices](#)

Printing

The simplest way to print a file while you are in Microsoft Developer Studio is to choose File/Print from the file menu. You are prompted for the file name and the file is printed on the printer connected to your computer.

You can also print files with the extension .F90, .FOR, .FD, .FI, or .RC by dragging the file from Windows Explorer and dropping it on the Print Manager icon.

If you have drawn graphics on the screen and want to print the screen, the simplest way is to press the key combination ALT+PRINT SCREEN. This copies the active window (the one with graphical focus) onto the Clipboard. (If you only press PRINT SCREEN, it prints your entire screen including any background applications.)

Once you have copied your screen onto the Clipboard, open Paintbrush and select Edit/Paste to paste the image from the Clipboard into Paintbrush, then select File/Print to print it to the printer connected to your computer. You can also save the image as a bitmap (.BMP) file.

Physical Devices

I/O statements that do not refer to a specific file or I/O device read from standard input and write to standard output. Standard input is the keyboard, and standard output is the screen (console). To perform input and output on a physical device other than the keyboard or screen, you specify the device name as the filename to be read from or written to.

Some physical device names are determined by the host operating system; others are recognized by Visual Fortran. Extensions on most device names are ignored.

Table: Filenames for Device I/O

Device	Description
CON	Console (standard output)
PRN	Printer
COM1	Serial port #1
COM2	Serial port #2
COM3	Serial port #3

COM4	Serial port #4
LPT1	Parallel Port #1
LPT2	Parallel Port #2
LPT3	Parallel Port #3
LPT4	Parallel Port #4
NUL	NULL device. Discards all output; contains no input
AUX	Serial port #1
LINE ¹	Serial port #1
USER ¹	Standard output
ERR ¹	Standard error
CONOUT\$	Standard output
CONIN\$	Standard input

¹ If you use one of these names with an extension -- for example, LINE.TXT -- Fortran will write to a file rather than to the device.

Examples of opening physical devices as units are:

```
OPEN (UNIT = 4, FILE = 'PRN')  
OPEN (UNIT = 7, FILE = 'COM2', ERR = 100)
```

Input/Output Editing

This chapter discusses the following input/output (I/O) editing topics:

- [I/O Lists](#), which provide information about data to be transferred.
- [Methods of I/O Editing](#), which tells the I/O system how to transfer data between variables in memory and external devices or internal files.
- [Formatted I/O](#), which provides formatting information.
- [List-Directed I/O](#), which lets you read and write data in an I/O list without using a **FORMAT** statement; the I/O is controlled by using the number and type of data items in the list.
- [Namelist I/O](#), which lets you specify one or more data items in a namelist group, so that the values can be read or written with a single I/O statement.

This chapter, as well as [Input/Output Statements](#) and [Files, Devices, and I/O Hardware](#) explain where and how Fortran data is input and output. Files and devices are where data is stored and retrieved, I/O editing determines how the data is organized when it is read or written, and I/O statements determine what input/output operations are performed on the data.

Input/Output Lists

Data transfer statements (**READ**, **WRITE** and **PRINT**) need information about how to transfer data and which data to transfer. Specifying how data is transferred is described in [Methods of I/O Editing](#). The data to be transferred is specified by listing the items to be read or written in an I/O list (*iolist*). You can specify an *iolist* by using the following:

- No entry

An *iolist* can be empty. The resulting record is either of zero length or contains only padding characters. This is useful if you need to write a record as a placeholder. If you use a format that contains a string but has no *iolist*, the resulting record will contain the string. For instance:

```
WRITE (UNIT=7, FMT='(2I8)')
WRITE (4, "('string')")
```

- A variable name, an array-element name, a derived type name, a derived-type element name, or a character-substring name.

```
! 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

! 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 )

```

- An array name or array section specification.

An unsubscripted array name specifies in column-major order all the elements of the array. Assumed-size dummy arrays that do not reference a specific array element cannot appear in the list of an input/output statement, but allocatable arrays and assumed-size dummy arrays can appear in an *iolist*.

```

! 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)

```

- Any expression.

Output lists in **WRITE** and **PRINT** statements can contain any expression, either numeric, logical, character or derived-type (operators can be defined for derived types).

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

- A namelist.

By specifying a namelist, you can read or write all the variables in it with one I/O statement.

```

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

```

For detailed information about namelists, see [Namelist I/O](#).

- An implied-**DO** list.

An implied-**DO** list is like an ordinary **DO** loop. The *start*, *stop*, and *inc* parameters determine the number of iterations, and *dovar* (where appropriate) can be used as an array element specifier.

In the following example, the *iolist* tells Visual Fortran 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 on implied-**DO** lists, see [DATA](#) in the *Reference*.)

```

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

```

Methods of I/O Editing

I/O editing tells the Visual Fortran I/O system how to transfer data between variables in memory and external devices or internal files. There are three methods of specifying how data is transferred with **READ**, **WRITE**, and **PRINT** statements: explicit formatting, list-directed I/O, and namelist I/O.

- Formatted I/O

In explicitly formatted I/O, you specify exactly how you want the data organized. The format specifier used by the I/O statement is either a character expression of the desired format or a label associated with a **FORMAT** statement containing the format. For example:

```

      WRITE (*, '(I3)') int1
      WRITE (*, 9000) int2
9000  FORMAT ( I5)

```

- List-directed I/O

In list-directed I/O, you read and write data items in an I/O list without an explicit format or reference to a **FORMAT** statement. Instead, the format specifier in the I/O statement is an asterisk (*). For example:

```

      WRITE (6,*) int1

```

- Namelist I/O

In namelist I/O, you read and write data by specifying one or more variables in a namelist group you create with the **NAMELIST** statement. The format specifier is the namelist name. For example:

```

      NAMELIST /example/ int1, int2
      WRITE (*, example)

```

Formatted I/O

Format information is contained in a format list and used by **PRINT**, **READ**, and **WRITE** statements. These statements can contain a format list themselves, or contain the label of a

FORMAT statement with a format list, or contain a variable whose value is set to an edit list or statement label.

A format list is a series of formatting descriptors, separated by commas (,), which describes the data to be transferred, such as the number, data types and lengths of variables to be read or written. The following are examples of a format list in a **FORMAT** and a **WRITE** statement:

```
! The format list in this FORMAT statement is below the
! hyphens: -----
100  FORMAT (' A = ', I5, ' B = ', F7.2)
!
! The format list in this WRITE statement is below the
! hyphens: -----
      WRITE(*, '(F8.5, 2I3, A20)') REAL1, INT1, INT2, "format list example"
```

The format list (including the outer parentheses) is a character constant, and is enclosed in single or double quotation marks when it appears in a **READ** or **WRITE** statement. There are no quotation marks around the entire format list when it appears in a **FORMAT** statement. An edit list can also contain another format list. Up to 8 levels of nested parentheses are permitted within the outermost level of parentheses in a format list.

Format lists are made up of repeatable and nonrepeatable edit descriptors. Repeatable edit descriptors describe data items. For example, 2I3 specifies that the edit descriptor I3 is repeated twice, so two three-digit integer numbers are written. Nonrepeatable edit descriptors modify the data format. For example, SP causes a plus sign (+) to be output with positive numbers. For more information, see [Repeatable Edit Descriptors](#) and [Nonrepeatable Edit Descriptors](#).

You can designate the format containing a format list by using any of the following:

- A **FORMAT** statement label

If you specify the label of a **FORMAT** statement in your I/O, the format list in the **FORMAT** statement formats the data.

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

- An integer-variable name

You can use an **ASSIGN** statement to associate an integer variable with the label of a **FORMAT** statement, and then use the variable to refer to the **FORMAT** statement. In this example, the integer-variable name MYFMT refers to the **FORMAT** statement 9000, as assigned just before the **FORMAT** statement. (For more information, see [ASSIGN](#) in the *Reference*.)

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

- A character expression or variable

You can write a format list as a character expression and use it in a **READ**, **WRITE**, or **PRINT** statement. For example:

```

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

```

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

```

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

```

- An array or array element

If you write a format list as a character expression and assign it to an array, you can use the array as the format specifier. The array is interpreted as all the elements of the array concatenated in column-major order. For instance, consider the following two-dimensional array:

```

  1  2  3
  4  5  6

```

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

```

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

```

If you write a format list as a character expression and assign it to a character array element, you can use the character array element as the format specifier. In the following example, the **WRITE** statement uses the character array element array(2) as the format specifier for data transfer:

```

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

```

[A noncharacter array can also be specified where the elements of the array are treated as equivalent character variables of the same length.](#)

You use repeatable and nonrepeatable edit descriptors to tell the Visual Fortran I/O system exactly how to organize your data in formatted I/O. Repeatable edit descriptors tell the I/O system how to interpret a given data item. Nonrepeatable edit descriptors tell the Fortran I/O system how to perform the I/O statement, for example, position within a record, interpretation of blanks, and sign suppression.

This section also discusses [variable format expressions](#) and the [interaction between format specifications and I/O lists](#).

Repeatable Edit Descriptors

To transfer data with input and output operations, you must list the data items to be transferred and specify how they are transferred. The data items are specified with an input/output list, or *iolist*, and their transfer is specified with an I/O format. Repeatable edit descriptors tell the Visual Fortran I/O system how to interpret data items with the **FORMAT** statement.

Repeatable edit descriptors are used as many times as needed to describe all data items in an *iolist*.

You can also indicate that a given data format is repeated some number of times. For example, 5I3 repeats a three-digit integer edit descriptor five times.

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

The following output is produced:

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

A repeat specification is a nonzero, unsigned integer constant or an integer expression enclosed by angle brackets (< and >) that tells the Visual Fortran I/O system how many of the repeatable items there are. For example, 2I5 says that there are two five-digit integer data items, and <J+K>I5 causes the expression J+K to be evaluated as the number of I5 data items.

Repeatable edit descriptors are:

- Integer Editing (I)
- Binary (B), Octal (O), and Hexadecimal (Z) Editing
- Real Editing Without Exponents (F)
- Real Editing With Exponents (E)
- Double-Precision Real Editing (D)
- Engineering-Notation Editing (EN)
- Scientific-Notation Editing (ES)
- Logical Editing (L)
- Character Editing (A)
- Generalized Editing (G)

The **I** (integer), **B** (binary), **O** (octal) **Z** (hexadecimal), **F** (single-precision real), **E** (real with exponent), **EN** (engineering notation real), **ES** (scientific notation real), **G** (general [integer, logical or real] with optional exponent), and **D** (double-precision real) edit descriptors are used for I/O of numeric data. The following rules apply to numeric edit descriptors:

- On input, fields that are all blanks are always interpreted as zero. The interpretation of trailing and interspersed blanks is controlled by the **BN** and **BZ** edit descriptors and by the **BLANK=** option in an **OPEN** statement. Plus signs (+) are optional, except in exponents. The blanks supplied by the file system to pad a record to the required size are not significant.
- On input with **F**, **E**, **EN**, **ES**, **G**, and **D** editing, an explicit decimal point in the input field overrides any edit-descriptor specification of the decimal-point position. On output, the decimal position given in the edit descriptor is followed. Consider the following:

```
      READ (*, 100) x
100   FORMAT(F4.2)
```

If $x = 123.4$ is specified, the decimal in the input overrides the specification in the **F** edit descriptor in the **FORMAT** statement. So, the number is read correctly as 123.4, not 23.40 as specified by the descriptor. Consider the following:


```

      WRITE(*, 200) x
200   FORMAT (F6.2)

```

In this case, if $x = 123.4$ is specified, the decimal in x does not override the **FORMAT** statement. So, the program outputs 123.40.

- On output, if the number of characters produced exceeds the field width or if the exponent exceeds its specified width, the entire field is filled with asterisks (*). If a real number contains more digits after the decimal point than are allowed in the field, the number is rounded. Consider the following:

```

      WRITE(*, 200) x
200   FORMAT (F4.1)

```

If $x = 123.4$ is specified, the program will write ***** because there are five characters in 123.4.

- On output, the characters generated are right-justified in the field and padded by leading blanks, if necessary.
- When reading with **I**, **B**, **O**, **Z**, **F**, **E**, **EN**, **ES**, **G**, **D**, or **L** (logic) edit descriptors, the input field may contain a comma (,), that terminates the field. The next field starts at the character following the comma. The missing characters are not significant. Do not use this feature when you use explicit positional editing (the **T**, **TL**, **TR**, or **nX** edit descriptors) because it changes the character positions of the data.
- Two successively interpreted edit descriptors of the types **F**, **E**, **G**, and **D** are required to format complex numbers. Two different descriptors can be used. The first edit descriptor specifies the real part of the complex number, and the second specifies the imaginary part.
- Nonrepeatable edit descriptors can appear between repeatable edit descriptors.

Integer Editing (I)

Syntax

I w [. m]

The field of the integer edit descriptor is w characters wide, including a sign if one is present. For example, an integer identified as **I5** is five characters wide, including the sign. Optionally, m specifies the minimum number of digits in the output. For example, a number identified as **I5.3** has a field width of five characters on input, and on output a maximum width of five characters with a minimum of three digits. Numbers associated with the **I** descriptor must be integers; they cannot have a decimal point or an exponent. If a noninteger number is input or output with the **I** descriptor, a run-time error occurs.

The m parameter has no effect on input and must be less than or equal to w . If m is not given, the minimum number of output digits defaults to 1. If the input or output is less than w characters wide, it is padded with leading blanks. If the output is less than m digits, leading zeroes are added up to the width m . For example, consider the statement:

```
WRITE (*, '(I5, I5.3, I5.3)') 4, -4, 4567
```

The following output is produced:

4 -004 4567

Binary (B), Octal (O), and Hexadecimal (Z) Editing

Syntax

B $w[.m]$, **O** $w[.m]$, **Z** $w[.m]$

Binary (**B**), Octal (**O**), and Hexadecimal (**Z**) edit descriptors cannot contain a decimal point or a sign (plus or minus). Data corresponding to these edit descriptors consist solely of blanks and digits in the binary, octal, or hexadecimal base: digits 0 and 1 for the **B** descriptor; digits 0-7 for the **O** descriptor; digits 0-9 and letters A-F (uppercase or lowercase) for the **Z** descriptor. The **B**, **O**, and **Z** edit descriptor data can be of type integer, [character](#), [real](#), or [logical](#).

Because there is no minus sign (-), negative **B**, **O**, and **Z** values are represented according to the coding convention used (for instance, two's complement). Encoding of binary, octal, and hexadecimal numbers, especially negative numbers, is processor-dependent. Programs that use the **B**, **O**, and **Z** edit descriptors and **B**, **O**, and **Z** stored data may not port directly to other computers.

The field is w characters wide, and m specifies the minimum number of digits in the output. The m parameter must be less than or equal to w . If m is not given, the minimum defaults to 1. If the output is less than w characters wide, it is padded with leading blanks. If the output is less than m digits, leading zeros are added up to the width m . Binary numbers in particular are often easier to read with leading zeros. For example, 00010101 shows all eight bits unlike 10101. You can force leading zeros with B8.8, B16.16, B32.32 and so on.

With **B**, **O**, and **Z** editing, you can convert between external data in binary, octal, or hexadecimal form and the internal numeric representation. Each byte of internal data corresponds to eight (1-bit) binary characters, three octal characters, and two (4-bit) hexadecimal characters. For example, the number 255 is output as the binary characters 11111111, the octal characters 377, and the hexadecimal characters FF. Similarly, an INTEGER(4) value is output as the thirty-two binary characters, twelve octal characters or eight hexadecimal characters in which the number was stored.

The field width, w , specifies the number of characters to be read or written. If w is omitted, the field width defaults to $8*n$ binary characters, $3*n$ octal characters, or $2*n$ hexadecimal characters, where n is the length in bytes of the item in the I/O list. For example, an INTEGER(2) value is represented internally by four hexadecimal characters.

For numeric and logical types, bytes are output in order of significance, from the most significant on the left to the least significant on the right. The INTEGER(2) value 10, for example, will be output in hexadecimal format as three blanks followed by A (assuming m is not specified).

The following rules of truncation and padding apply. In this table, the value n is the length of the *iolist* variable in bytes:

Operation	Rule
Output	If $w > 8*n$ (binary), $3*n$ (octal), or $2*n$ (hexadecimal), the characters are right-justified and leading blanks are added to make the external field width equal to w .
	If $w \leq 8*n$ (binary), $3*n$ (octal), or $2*n$ (hexadecimal), the field is filled with asterisks

	(*). If $m > 8*n$ (binary), $3*n$ (octal), or $2*n$ (hexadecimal), the characters are right-justified and leading zeroes are added to make the number of digits equal to m . If $m < 8*n$ (binary), $3*n$ (octal), or $2*n$ (hexadecimal), m has no effect.
Input	If $w \geq 8*n$ (binary), $3*n$ (octal), or $2*n$ (hexadecimal), the rightmost $8*n$ (binary), $3*n$ (octal), or $2*n$ (hexadecimal) characters are taken from the input field. If $w < 8*n$ (binary), $3*n$ (octal), or $2*n$ (hexadecimal), the first w characters are read from the input field. Enough leading blanks are added to make the width equal to $8*n$ (binary), $3*n$ (octal), or $2*n$ (hexadecimal). The m parameter has no effect on input.

Leading blanks are never significant. Blanks embedded in an input field are ignored unless the **BLANK= 'ZERO'** specifier is set during an **OPEN** statement, or the **BZ** edit descriptor is in effect. (A description of the **BZ** and **BN** edit descriptors is given in [Blank Interpretation \(BN, BZ\)](#) within the [Nonrepeatable Edit Descriptors](#) topic.)

To edit complex numbers, two **Z** edit descriptors must be used. The first edit descriptor specifies the real part of the complex number, and the second specifies the imaginary part.

The following example demonstrates hexadecimal editing for output:

```
CHARACTER(2) alpha
INTEGER(2) num

alpha = 'YZ'
num   = 3035

WRITE (*, '(Z4.4, 1X, Z2, 1X, Z6)') alpha, alpha, alpha
WRITE (*, '(Z4.4, 1X, Z2, 1X, Z6)') num, num, num
WRITE (*, '(B16.16, 1X, B2, 1X, B6)') num, num, num
WRITE (*, '(O5.5, 1X, O2, 1X, O6)') num, num, num
```

This example produces the following output:

```
5A59 **      5A59
0BDB **      BDB
0000101111011011 ** *****
05733 **     5733
```

As an example of input, suppose the input record is 5A59 (hexadecimal for "YZ"), and the *iolist* variable has been declared as a CHARACTER(2) type. The record would be read as follows:

Edit descriptor	Value read
Z	YZ
Z2	Z
Z6	YZ

Real Editing Without Exponents (F)

Syntax

F*w.d*

The **F** edit descriptor tells Fortran to treat a number as a simple decimal floating-point value. On output, the I/O list item associated with an **F** edit descriptor must be a single- or double-precision real, or the real or imaginary part of a complex number; otherwise, a run-time error occurs.

On input, the number entered may have any real or complex form as long as its value is within the range of the associated variable. Each complex number requires two **F** edit descriptors. For example:

```

      COMPLEX c1
      READ(10, 100) c1
100   FORMAT(2F8.3)

```

The field is *w* characters wide, with a fractional part *d* decimal digits wide. The input field begins with an optional sign followed by a string of digits that may contain an optional decimal point. If the decimal point is present in the input, it overrides the *d* specified in the edit descriptor; otherwise, the rightmost *d* digits of the string are interpreted as following the decimal point (with leading blanks converted to zeros, if necessary).

Values input with the **F** descriptor can also be expressed in exponent form. In this case, the field continues with either a plus (+) or minus (-) sign followed by an integer, or the **E** or **D** exponent designator followed by zero or more blanks, followed by an optional sign, followed by an integer, e.g. F8.3E-03.

An example is the following **READ** statement:

```
READ (*, '(F8.3)') xnum
```

This statement reads a given input record as follows:

Input	Number read
5	.005
2468	2.468
-24680	-24.680
-246801	-246.801
5678	5.678
-28E2	-2.800

The output field occupies *w* characters. One character is a decimal point, leaving *w*-1 characters available for digits. If the sign is negative, it must be included, leaving only *w*-2 characters available. Out of these *w*-1 or *w*-2 characters, *d* characters will be used for digits to the right of the decimal point. The remaining characters will be blanks or digits, as needed, to represent the digits to the left of the decimal point.

The value of the number being output is controlled both by the *iolist* item and the current scale factor. The number being output using **F** editing is rounded rather than truncated.

```

      REAL(4) g, h, e, r, k, i, n
      DATA g /12345.678/, h /12345678./, e /-4.56E+1/, r /-365/
100   WRITE (*, 100) g, h, e, r
      FORMAT (F8.2)
200   WRITE (*, 200) g, h, e, r
      FORMAT (4F10.1)

```

The preceding program produces the following output:

```

12345.68
*****
-45.60
-365.00
12345.712345680.0    -45.6    -365.0

```

Real Editing With Exponents (E)

Syntax

$Ew.d [Ee]$

An **E** edit descriptor tells Visual Fortran that there is an exponent in the syntax of the value. The I/O list item associated with the **E** edit descriptor for an output item must be a single- or double-precision real, or the real or imaginary part of a complex number; otherwise, a run-time error occurs.

A number input to a variable described with an **E** edit descriptor can have any real or complex form, as long as its value is within the range of the associated variable. Note that each complex number requires two **E** edit descriptors.

The field is w characters wide. The e parameter is ignored in input statements. The input field for the **E** edit descriptor is identical to that described by an **F** edit descriptor with the same w and d .

The form of the output field depends on the scale factor (set by the **P** edit descriptor) in effect. For a scale factor of 0, the output field is a minus sign (if necessary), followed by a decimal point, followed by a string of digits, followed by an exponent field for exponent e , having one of the forms shown in the following table:

Table: Forms of Exponents for the E Edit Descriptor		
Edit descriptor	Absolute value of exponent	Form of exponent
$Ew.d$	$ exp \leq 99$	E followed by plus or minus, followed by the 2-digit exponent. Example: E8.3E-22
$Ew.d$	$99 < exp \leq 999$	Plus or minus, followed by the 3-digit exponent. Example: E8.3+102
$Ew.d Ee$	$ exp \leq (10e) - 1$	E followed by plus or minus, followed by e digits, which are the exponent (with possible leading zeros). Example: E8.3E-22

The scale factor controls the decimal normalization of the printed **E** field. If the scale factor k is greater than $-d$ and less than or equal to 0, then the output field contains exactly k leading zeros after the decimal point and $d+k$ significant digits after this. If $(0 < k < d+2)$, the output field contains exactly k significant digits to the left of the decimal point and $(d-k-1)$ places after the decimal point. Other values of k are errors.

The $Ew.d [De]$ descriptor is equivalent to $Ew.d [Ee]$ except that a 'D' is printed instead of 'E' on output.

Double-Precision Real Editing (D)

Syntax

D*w.d*

On output, the I/O list item associated with a **D** edit descriptor must be a single- or double-precision real, or the real or imaginary part of a complex number; otherwise, a run-time error occurs.

On input, the number entered can have any real or complex form, as long as its value is within the range of the associated variable. All parameters and rules for the **E** descriptor apply to the **D** descriptor.

The field is *w* characters wide. The input field for the **D** edit descriptor is identical to that described by an **F** edit descriptor with the same *w* and *d*.

The form of the output field depends on the scale factor (set by the **P** edit descriptor) in effect. For a scale factor of 0, the output field is a minus sign (if necessary), followed by a decimal point, followed by a string of digits, followed by an exponent field for exponent *e*, in one of the forms shown in the following table:

Table: Forms of Exponents for the D Edit Descriptor		
Edit descriptor	Absolute value of exponent	Form of exponent
D <i>w.d</i>	$ exp \leq 99$	D followed by plus or minus, followed by the two-digit exponent
D <i>w.d</i>	$99 < exp \leq 999$	Plus or minus, followed by the three-digit exponent

The form **D***w.d* must not be used if the absolute value of the exponent to be printed exceeds 999.

The scale factor controls the decimal normalization of the printed **D** field. If the scale factor, *k*, is greater than $-d$ and less than or equal to 0, then the output field contains exactly *k* leading zeros after the decimal point and $d+k$ significant digits after this. If $(0 < k < d+2)$, then the output field contains exactly *k* significant digits to the left of the decimal point and $(d-k-1)$ places after the decimal point. Other values of *k* are errors.

Engineering-Notation Editing (EN)

Syntax

EN*w.d [Ee]*

The **EN** edit descriptor is the same as the **E** descriptor, except that the absolute value of the nonexponential part of the output data (the significand) is constrained to be between 1 and 1000 ($1 \leq |\text{significand}| < 1000$) and the exponent is divisible by 3.

The field, including exponent, is *w* characters wide, with *d* characters after the decimal, and an optional exponent width of *e*. The exponent takes the same forms shown in the exponent table [Forms of Exponents for the E Edit Descriptor](#).

```
REAL x, y, z
DATA x /-12345.678/, y /0.456789/, z /7.89123E+23/
WRITE (*, 100) x, z
```

```

100  FORMAT (EN13.5,1X,EN13.5)
      WRITE (*, 200) Y, Z
200  FORMAT (EN13.2E4,1X,EN13.2E4)

```

This produces the following output:

```

-12.34568E+03  789.12300E+21
 456.79E-0003  789.12E+0021

```

Scientific-Notation Editing (ES)

Syntax

ES*w.d [Ee]*

The **ES** edit descriptor is the same as the same as **E** descriptor, except that the absolute value of the nonexponential part of the output data (the significand) is constrained to be between 1 and 10 ($1 \leq |\text{significand}| < 10$).

The field, including exponent, is *w* characters wide, with *d* characters after the decimal, and an optional exponent width of *e*. The exponent takes the same forms shown in the exponent table [Forms of Exponents for the E Edit Descriptor](#).

Logical Editing (L)

Syntax

L*w*

The field is *w* characters wide.

On input, the field consists of optional blanks, followed by an optional decimal point, followed by **T** (for true) or **F** (for false). Any further characters in the field are ignored, but accepted on input; so, for example, **.TR.**, **.TRU**, **TCX**, **.TRUE.**, **.FA.**, and **.FALSE.** are all valid inputs. Note that either upper- or lower-case letters can be used.

The *iolist* element associated with an **L** edit descriptor for output must be of type logical or integer, or a run-time error occurs. On output, *w*-1 blanks are followed by either **T** or **F**, as appropriate.

Character Editing (A)

Syntax

A[*w*]

If the field width (*w*) is omitted, it defaults to the number of characters in the *iolist* associated item. The *iolist* item can be of any type.

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.](#)

If the number of characters input is less than w , the input field is padded with blanks. If the number of characters input is greater than w , the input field is truncated on the right to the length of w . After these adjustments have been made, the input field is put into the *iolist* item. For example, using the following program fragment:

```
CHARACTER(10) char
READ (*, '(A15)') char
```

Assume the following 13 characters are typed in at the keyboard:

```
ABCDEFGHIJKLM
```

The following two steps occur:

1. Spaces are added to pad the input field to 15 characters:

```
'ABCDEFGHIJKLM '
```

2. The rightmost 10 characters are transmitted to the *iolist* element char:

```
'FGHIJKLM '
```

On output, if w exceeds the number of characters produced by the *iolist* item, leading spaces are provided. Otherwise, the leftmost w characters of the *iolist* item are output.

Generalized Editing (G)

Syntax

G $w.d$ [**E** e]

The **G** edit descriptor can be used with any intrinsic data type. For integer input/output the **G** $w.d$ generalized edit descriptor is the same as the **I** w edit descriptor. For logical data the **G** w edit descriptor is the same as **L** w , and for character data the **G** w descriptor is the same as the **A** w descriptor.

For real numerical data, the **G** $w.d$ [**E** e] descriptor is useful and flexible. With **F** editing of wide-range reals numbers, significant digits can be lost if the number becomes too large or too small for the field. However, **F** descriptor data is often easier to read; for example, 123.45 is more readable than 0.12345E+03. The **G** descriptor switches automatically from **F** format to **E** or **D** format, depending on the magnitude of the data.

When **G** $w.d$ [**E** e] is used as a real edit descriptor, the input field is w characters wide, with a fractional part consisting of d digits. If an exponent is specified, it consists of e digits. When **G** is used as a real edit descriptor, **G** input editing is the same as **F** input editing.

The form in which a **G** edit descriptor value is written is a function of the magnitude of the value, as described in the following table:

Table: Effect of Data Magnitude on G Format Conversions	
Data Magnitude	Effective Conversion

$0 < m < 0.1 - 0.5 \times 10^{-d-1}$	Ew.d[Ee]
$m = 0$	F(w - n).(d - 1), n('b')
$0.1 - 0.5 \times 10^{-d-1} \leq m < 1 - 0.5 \times 10^{-d}$	F(w - n).d, n('b')
$1 - 0.5 \times 10^{-d} \leq m < 10 - 0.5 \times 10^{-d+1}$	F(w - n).(d - 1), n('b')
$10 - 0.5 \times 10^{-d+1} \leq m < 100 - 0.5 \times 10^{-d+2}$	F(w - n).(d - 2), n('b')
.	.
.	.
.	.
$10^{d-2} - 0.5 \times 10^{-2} \leq m < 10^{d-1} - 0.5 \times 10^{-1}$	F(w - n).1, n('b')
$10^{d-1} - 0.5 \times 10^{-1} \leq m < 10^d - 0.5$	(w - n).0, n('b')
$m \geq 10^d - 0.5$	Ew.d[Ee]

The 'b' is a blank following the numeric data representation. For $Gw.d$, n('b') is 4 blanks. For $Gw.dEe$, n('b') is +2 blanks.

The $Gw.d [De]$ descriptor is equivalent to $Gw.d [Ee]$ except that a 'D' is printed instead of 'E' on output.

Nonrepeatable Edit Descriptors

Nonrepeatable edit descriptors modify the way the repeatable edit descriptors are interpreted, and can also change or modify the way I/O is performed. The following table summarizes the nonrepeatable edit descriptors. The following sections discuss each nonrepeatable edit descriptor.

Form	Name	Use	Used for Input	Used for Output
'string' or "string"	Character string editing	Transmits <i>string</i> to output unit	No	Yes
nH	Hollerith editing	Transmits next <i>n</i> characters to output unit	No	Yes
Q	Character count editing	Returns remaining number of characters in record	Yes	No
Tc, TLc, TRc	Positional editing (Tabs)	Specifies position in record	Yes	Yes
nX	Positional editing (X)	Specifies position in record	Yes	Yes
SP, SS, S	Optional-plus editing	Controls output of plus signs	No	Yes
/	Slash editing	Positions to next record or writes end-of-record mark	Yes	Yes
\	Backslash editing	Continues same record	No	Yes
\$	Dollar-sign editing	Same as backslash	No	Yes
:	Format control termination	If no more items in <i>iolist</i> , terminates statement	No	Yes
kP	Scale-factor editing	Sets scale for exponents in subsequent F	Yes	Yes

		and E (repeatable) edit descriptors		
BN, BZ	Blank interpretation	Specifies interpretation of blanks in numeric fields	Yes	No

The comma (,) used to separate list items can be omitted before or after several nonrepeatable edit descriptors as follows:

- Between a **P** edit descriptor and an immediately following **F, E, EN, ES, D,** or **G** edit descriptor; for example: `I3,2PF8.6,4F4.3`
- Before or after an apostrophe ('), double-quotation mark ("), **backslash (\), dollar sign (\$),** or colon (:) edit descriptor; for example: `A14,A35,I5$`
- Before a slash (/) edit descriptor when the optional repeat specification is not present, and after a slash (/) edit descriptor in all cases
- After an **nH** or **X** edit descriptor; for example: `2I3,8HF5.3,A12`

Character String Editing

Apostrophe (') and quotation mark (") edit descriptors can be used to specify how character data is output. A format specifier containing a character constant is transmitted to the output unit as the format to be followed during output. For example, 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)
```

Embedded blanks in character-constant formats are significant. Apostrophe and quote delimited character strings cannot be used as input formats, such as for **READ** statements.

If a character constant delimited by an apostrophe contains an apostrophe, or a character constant delimited by quotation marks contains quotation marks, the embedded delimiting character must be doubled to show that it is being used as a character, not a delimiter. For example, two adjacent apostrophes (single quotation marks) must be used to represent an apostrophe within a character constant.

Each additional level of nested apostrophes requires twice as many apostrophes as the previous level to resolve the ambiguity of the apostrophe's meaning. Note how in the second **WRITE** statement in the example following, the set of apostrophes that delimit the output string within the **FORMAT** statement is doubled, and four apostrophes are required within the output string itself to specify a single output apostrophe.

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

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

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

Hollerith Editing (H)

You can use Hollerith editing to read or write a text string of known length to a file or device. The *nH* edit descriptor transmits the next *n* characters, including blanks, to the output unit. **On input, the *nH* edit descriptor transfers *n* characters from the external field to the edit descriptor. Hollerith editing can be used in every context where character constants can be used.** It is most useful when you want to write a character string containing many double and single quotation marks, because you don't have to insert additional apostrophes and quotation marks into the string.

As with apostrophe and quotation mark editing, Hollerith descriptors can only be used in output formats, not in input formats.

The *n* characters transmitted are called a *Hollerith constant*.

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

Character Count Editing (Q)

The **Q** edit descriptor obtains the number of characters in an input record remaining to be transferred during a read operation. The I/O list element corresponding to the **Q** descriptor must be integer or logical type. The following demonstrates the use of the **Q** descriptor:

```
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 the section Variable Format Expressions.)

Assume the file Q.DAT contains:

```
1234.567Hello, Q Edit
```

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

```

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

```

The output on the screen is:

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

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

```

abcdefg
abcd

```

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

```

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

```

The output to the screen would be:

```
7
4
```

Positional Editing: Tab, Tab Left, Tab Right (T, TL, TR)

The **T**, **TL**, and **TR** edit descriptors specify the position in the record to which or from which the next character will be transmitted. The new position may be in either direction from the current position. This allows a record to be processed more than once on input.

The **Tc** edit descriptor specifies absolute tabbing; the transmission of the next character is to occur at the character position *c*. The **TRc** edit descriptor specifies relative tabbing to the right; the transmission of the next character is to occur *c* characters beyond the current position. The **TLc** edit descriptor specifies relative tabbing to the left; the transmission of the next character is to occur *c* characters prior to the current position.

If **TLc** specifies a position before the first position of the current record, **TLc** editing causes transmission to or from position 1.

Left tabbing is legal within records written to devices. However, if the record that is written is longer than the buffer associated with the device, you cannot left-tab to a position corresponding to the

previous buffer.

For example, if the buffer associated with the console is 132 bytes, and a record of 140 bytes is written to the console, left tabbing is allowed for only eight bytes. The first 132 bytes of the record have been sent to the device and are no longer accessible.

If one of these edit descriptors is used to move to a position to the right of the last data item transmitted and another data item is then written, the space between the previous end of data in the record and the new position is filled with spaces. The following code fragment writes three adjacent copies of the number ruler string for a reference, and then demonstrates how tab editing works. The following **FORMAT** statement specifies to write 5 as a 5-byte integer, skip forward 20 spaces, back up 10 spaces, and then write 9 as a 5-byte integer.

```

        WRITE (*, '( ' ' ', 3('1234567890'))')
        WRITE (*, 100) 5, 9
100    FORMAT (I5, 20X, TL10, I5)

```

This example produces the following output:

```

123456789012345678901234567890
   5                               9

```

Be careful when using these edit descriptors if you read data from files that use commas (,) as field delimiters. If you move backward in a record using **TLc** or **Tc** (where *c* is less than the current position in the record), commas are disabled as field delimiters. If the format controller encounters a comma after you have moved backward in a record with **TLc** or **Tc**, a run-time error occurs. If you want to move backward in a record without disabling commas as field delimiters, advance to the end-of-record mark, and then use the **BACKSPACE** statement to move to the beginning of the record.

Positional Editing (X)

The **nX** edit descriptor advances the file position *n* characters. On output, if the **nX** edit descriptor moves past the end of data in the record, and if there are further items in the *iolist*, blanks are output, as described for the **Tc** and **TRc** edit descriptors. If *n* is absent, the **X** edit descriptor defaults to **1X**.

```

!      This example writes the following to the screen:
!          1          2
!      -----+-----+
!      WRITE (*, 100)
!      WRITE (*, 200)
100    FORMAT (9X, '1', 9X, '2')
200    FORMAT ('-----', TL15, '+', 3(4X, '+'))

```

Optional-Plus Editing (SP, SS, S)

The **SP**, **SS**, and **S** edit descriptors control the printing of optional leading plus (+) signs in numeric output fields. **SP** causes output of the plus sign in all subsequent positions that the processor recognizes as optional-plus fields. **SS** causes plus sign suppression in all subsequent positions that the processor recognizes as optional-plus fields. **S** restores **SS**, the default.

```

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)

```

Slash Editing (/)

The *r* (slash) edit descriptor indicates the end of data transfer on the current record. The *r* is a repeat specification. It must be a positive default integer literal constant.

On input, the file is positioned to the beginning of the next record. On output, an end-of-record mark is written, and the file is positioned to write at the beginning of the next record.

```

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

```

This example generates the following output:

```

  A      B      C      D      E
1
2
3
4
5

```

Multiple slashes cause the system to skip input records or to output blank records, as follows:

- When *n* consecutive slashes appear between two edit descriptors, *n* - 1 records are skipped on input, or *n* - 1 blank records are output. The first slash terminates the current record. The second slash terminates the first skipped or blank record, and so on.
- When *n* consecutive slashes appear at the beginning or end of a format specification, *n* records are skipped or *n* blank records are output, because the opening and closing parentheses of the format specification are themselves a record initiator and terminator, respectively. For example, suppose the following statements are specified:

```

          WRITE (6,99)
99       FORMAT ('1',T51,'HEADING LINE'///T51,'SUBHEADING LINE'///)

```

The following lines are written:

```

          Column 50, top of page
          |
          HEADING LINE
(blank line)
          SUBHEADING LINE

```

```
(blank line)
(blank line)
```

Note that the first character of the record printed was reserved as a control character (see [Carriage Control Specifier](#)).

Backslash (\) and Dollar-Sign (\$) Editing

The backslash (\) and dollar sign (\$) edit descriptors prevent Visual Fortran from writing an end-of-record mark after all items in the *iolist* and *editlist* have been processed. They are typically used for formatted output to terminal devices, such as the screen or a printer.

When the format controller terminates a transmission to an output device, it writes an end-of-record mark (typically carriage return and line feed for consoles). If the last edit descriptor encountered by the format controller is a backslash or dollar sign, no end-of-record mark is written, so the next I/O statement continues writing on the same line.

For units connected to terminal devices, the end-of-record mark is not written until the next record is written to the unit. If the device is the screen, you can use the backslash or dollar sign to suppress the end-of-record mark. This mechanism can be used to write a prompt to the screen and then read a response from the same line, as in the following example:

```
! This example advances two lines, prompts for console input,
! awaits input on the same line as the prompt,
! and prints the input.
CHARACTER(20) MYNAME
WRITE (*,9000)
9000 FORMAT ('Please type your name:',\ )
READ (*,9001) MYNAME
9001 FORMAT (A20)
WRITE (*,9002) ' ',MYNAME
9002 FORMAT (A20)
```

Terminating Format Control (:)

The colon (:) edit descriptor terminates format control if there are no more items in the *iolist*. This feature is used to suppress output when some of the edit descriptors in the format do not have corresponding data in the *iolist*.

```
! 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
```

Scale-Factor Editing (P)

The *kP* edit descriptor sets the scale factor for all subsequent **F** and **E** edit descriptors and **G** edit descriptors with real variables, until another *kP* edit descriptor is encountered. (For information on **F**, **E**, and **G** edit descriptors, see [Repeatable Edit Descriptors](#).) At the start of each I/O statement, the scale factor is initialized to zero. The scale factor affects format editing in the following ways:

- On input, with **F** and **E** editing, if there is no explicit exponent, the value read in is divided by 10k before it is assigned to a variable. If there is an explicit exponent, the scale factor has no effect.
- On output, with **F** editing, the value to be written out is multiplied by 10k before it is displayed.
- On output, with **E** editing, the real part of the value to be displayed is multiplied by 10k, and its exponent is reduced by k . This alters the column position of the decimal point but not the value of the number.

The following code uses scale-factor editing when reading:

```

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

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

```

If the following data is entered:

```

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

```

The program's output is:

```

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

```

The next code shows scale-factor editing when writing:

```

      a = 12.34

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

```

This program's output is:

```

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

```

Blank Interpretation (BN, BZ)

The edit descriptors **BN** and **BZ** control the interpretation of blanks in numeric input fields.

The **BN** edit descriptor ignores blanks; it takes all of the nonblank characters in the field and right-aligns them. For example, if an input field formatted as a six-digit integer (I6) contains '2 3 4', it is interpreted as '234'.

The **BZ** edit descriptor makes trailing blanks and interspersed blanks identical to zeros, while leading blanks remain blanks. For example, 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'. Note that the

blanks following the **E** or **D** in real-number input are ignored, regardless of the form of blank interpretation in effect.

The default, **BN**, is set at the beginning of each I/O statement, unless the **BLANK=ZERO** option was specified in the **OPEN** statement (in which case each I/O statement on that unit begins with the setting **BZ**). If you specify a **BZ** edit descriptor, **BZ** editing is in effect until the **BN** edit descriptor is specified. Both **BN** and **BZ** override the **BLANK=** option setting.

For example, 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. If **BZ** editing were in effect, those three blanks would be interpreted as zeros, and the record would be equal to 123000. However, with **BN** editing in effect (the default), the nonblank characters (123) are right-aligned, so the record is equal to 123.

Variable Format Expressions

Wherever an integer constant is required by an edit descriptor, you can specify a numeric expression in a **FORMAT** statement. If the expression is not of type integer, it is converted to integer type before being used. The numeric expression must be enclosed by angle brackets (< and >). The following are valid format specifications:

```
      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 numeric expression can be any valid Visual Fortran expression, including function calls and references to dummy arguments, with the following restrictions:

- The **H** edit descriptor cannot use variable format expressions.
- Expressions cannot contain graphical relational operators (i.e, cannot contain < or >, but can contain **.LT.** or **.GT.**).

A variable format expression may not appear in an assigned-expression format statement such as:

```
CHARACTER(80) S
S = '(I<J+K>)'
WRITE(6, S) N
```

But can be used in statements such as:

```
WRITE(6, '(I<J+K>') N
```

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

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

When given input 3123 will print 123 and not 12.

Interaction Between Format Specifications and I/O Lists

If an *iolist* contains one or more items, at least one repeatable edit descriptor must appear in the format specification. The empty edit specification, (), can be used only if no items are specified in the *iolist*. A formatted **WRITE** statement with an empty edit specification writes a carriage return and line feed. A **READ** statement with an empty edit specification skips to the next record, unless I/O is set to be non-advancing (with the **ADVANCE='NO'** option), in which case the file position remains unchanged.

A record with fewer characters than the length specified by its edit descriptor is automatically padded on the right with blanks, unless you specify **PAD='NO'** in an **OPEN** statement. By default **PAD='YES'**. Any blanks entered by the user are interpreted according to the blank-editing descriptor in effect (**BN** or **BZ**) or the **BLANK=** option in an **OPEN** statement. **BN** and **BZ** override the **BLANK=** option.

As an example, consider the following **READ** statement that uses **BZ** editing (**PAD='YES'**, the default):

```
READ (*, '(BZ, I5)') n
```

Consider you enter the following in response:

5

The total number of characters in the input record is two (a blank followed by a 5). The record is padded on the right with three blanks, but these additional blanks added by the formatter are ignored. The input record is thus interpreted as 5, rather than 5000. Alternatively, if you had entered a blank, followed by a 5, followed by three blanks, your blanks, unlike formatter-added blanks, would be

interpreted as zeroes, and the result would be 5000.

Each item in the *iolist* is associated with a repeatable edit descriptor during the I/O statement execution. Each **COMPLEX** item in the *iolist* requires two edit descriptors in the **FORMAT** statement or format descriptor. Nonrepeatable edit descriptors are not associated with items in the *iolist*, **except for the Q edit descriptor**.

During the formatted I/O process, the format controller scans and processes the format items from left to right. Following is a list detailing situations the format controller might encounter and their explanations:

- If a repeatable edit descriptor is encountered and a corresponding item appears in the *iolist*, the item and the edit descriptor are associated, and I/O of that item proceeds under the format control of the edit descriptor.
- If a repeatable edit descriptor is encountered and no corresponding item appears in the *iolist*, the format controller terminates I/O. For example, consider the following statements:

```

      i = 5
      WRITE (*, 100) i
100   FORMAT ('I= ', I5, ' J= ', I5, ' K= ', I5)

```

In this case, the output would look like this:

```
I=      5 J=
```

The output terminates after J= because no corresponding item for the second I5 appears in the *iolist*.

- If a colon (:) edit descriptor (terminate format control) is encountered and there are no further items in the *iolist*, the format controller terminates I/O.
- If a colon (:) edit descriptor is encountered but there are further items in the *iolist*, the colon edit descriptor is ignored.
- If the matching final right parenthesis of the format specification is encountered and there are no further items in the *iolist*, the format controller terminates I/O.
- If the matching final right parenthesis of the format specification is encountered and there are further items in the *iolist*, the file is positioned at the beginning of the next record and the format controller starts over at the beginning of the format specification terminated by the last preceding right parenthesis.
- If there is no such preceding right parenthesis, the format controller rescans the format from the beginning. Within the portion of the format rescanned, there must be at least one repeatable edit descriptor.
- If the rescan of the format specification begins with a repeated nested format specification, the repeat factor indicates the number of times to repeat that nested format specification. The rescan does not change the previously set scale factor or the **BN** or **BZ** blank control in effect.

In advancing I/O, when the format controller terminates on input, any remaining characters of the record are ignored. When the format controller terminates on output, an end-of-record mark is written. Nonadvancing I/O leaves the file positioned after the last character read or written. No end-of-record mark is written.

Note: For units connected with **CARRIAGECONTROL='FORTRAN'** (typically terminals and

printers), the end-of-record mark is not written until the next record is written to the unit. You can use the backslash (\) or dollar-sign (\$) edit descriptor to suppress the end-of-record mark.

List-Directed I/O

When you use formatted I/O, you specify how the data is represented on the external device. With list-directed I/O, you can read and write data items in an *iolist* without a **FORMAT** statement. I/O is controlled by the number and type of data items in the *iolist*. For example:

```
INTEGER(2)    INT1, INT2
REAL(8)      REAL1, REAL2
CHARACTER    CHAR(7)
READ (10, *) INT1, INT2, REAL1, REAL2, CHAR(7)
```

The **READ** statement expects two 2-byte integer numbers, two 8-byte real numbers and seven characters.

List-directed I/O can be used to read and write external or internal files. (For a discussion of internal and external files, see [Logical Devices](#).) List-directed I/O statements include an asterisk (*) in place of the **FORMAT** number. For more about *iolists*, see [Input/Output Lists](#).

List-directed input is most useful in situations where the data is error-free and provided in a fixed and known form. List-directed output is most useful when you are more interested in what the data is than how it is formatted.

Note: Unformatted I/O (list-directed and namelist I/O) does not use explicit formatting or a **FORMAT** statement. When you perform unformatted I/O, Visual Fortran dumps the contents of the memory to the output device instead of formatting the data in the I/O list. If you consistently use unformatted I/O statements (and avoid mixing with formatted I/O), fewer run-time routines will be added to your program, and you will create smaller executable files.

When you use list-directed formatting to read or write data, the data items must match the internal representations to which they are mapped. To perform list-directed input, you must provide an *iolist* and a matching data stream. To perform list-directed output, you provide the *iolist* and Visual Fortran provides the formatting.

List-directed I/O is described in detail in the following sections:

- [List-Directed Input](#)
- [List-Directed Output](#)

List-Directed Input

To perform list-directed input, you provide an *iolist* containing the names of the variables into which the data is to be read. A list-directed input data record is a sequence of values separated by commas (,) or blanks. Each item in a list-directed data record must be either a value or a null (placeholder).

A null has no effect on the variable to which it is mapped: variables with values retain them, and those without values remain empty. A slash (/) terminates the input stream. Any further items in the input list are treated as if they were null values and have no effect. For example:

```

INTEGER I1, I2, I3
I1 = 1
I2 = 2
READ (*, *) I1, I2, I3

```

If the user inputs the following:

```
8, , / 9
```

I1 is assigned the value 8, I2 corresponds to an input null so its value remains 2, then the input record is terminated with the slash before a value for I3 is read, so I3's value remains null (zero).

Only string constants can contain embedded blanks. Blanks between two numeric values are interpreted as a value separator. Blanks next to value separators (commas, slashes, or other blanks) are ignored. For example:

```
5 , 6 / 7
```

This is equivalent to:

```
5,6/7
```

Repeated input values can be indicated by multiplying the value by the number of repetitions. For instance, the entry 3*5 tells Visual Fortran to read the value 5 three times. Consider the following:

```

REAL R(10)
READ (5,*) R

```

If the input record contains 10*3.1416, the ten elements of array R are set to 3.1416.

Most of the input forms available for formatted I/O are also available for list-directed formatting. The following rules apply to list-directed input for all values:

- The form of the input value must be acceptable for the type of input list item.
- Blanks are always treated as separators and never as zeros.
- Embedded blanks can appear only within character constants.

The end-of-record mark has the same effect as a blank, except when it appears within a character constant.

In addition to these rules, the following restrictions apply to the specified values:

Type of Value	Restrictions
Single- or double-precision real constants	A real or double-precision constant must be a numeric input field (a field suitable for F editing). It is assumed to have no fractional digits unless there is a decimal point within the field.
Complex constants	A complex constant is an ordered pair of real or integer constants separated by a comma and surrounded by opening and closing parentheses. The first constant of the pair is the real part of the complex constant, and the second is the imaginary part.
Logical constants	A logical constant must not include either slashes or commas among the optional characters permitted for L editing.
Character constants	A character constant is a nonempty string of characters enclosed in single quotation marks. Each single quotation mark within a character constant

	<p>delimited by single quotation marks must be represented by two single quotation marks, with no intervening blanks.</p> <p>Character constants can be continued from the end of one record to the beginning of the next; the end of the record doesn't cause a blank or other character to become part of the constant. The constant can be continued on as many records as needed and can include the blank, comma, and slash characters.</p> <p>If the length n of the list item is less than or equal to the length m of the character constant, the leftmost n characters of the latter are transmitted to the list item.</p> <p>If n is greater than m, the constant is transmitted to the leftmost m characters of the list item. The remaining n minus m characters of the list item are filled with blanks. The effect is the same as if the constant were assigned to the list item in a character assignment statement.</p>
<p>Derived types</p>	<p>A derived type in an input list is equivalent to putting all the elements of the derived type in the input list, in the same order they have in the derived-type declaration.</p>
<p>Null values</p>	<p>Null values indicate the absence of a data item for a given variable. You can specify a null value in one of three ways:</p> <ul style="list-style-type: none"> • No characters between successive value separators. For example, in a data record containing 54.23, , , 141, the two values after 54.23 are empty. • No characters preceding the first value separator in the first record read by each execution of a list-directed input statement. • You can also indicate a group of nulls using the asterisk form. For example, 10* is equivalent to 10 null values, or a piece of a data stream containing 16, , , , , , , , , 23.8. <p>A null value has no effect on the current definition of the corresponding input list item. If the input list item is defined, it retains its previous value; if it is undefined, it remains so.</p> <p>A slash encountered as a value separator during execution of a list-directed input statement stops execution of that statement after the assignment of the previous value. Any further items in the input list are treated as if they were null values.</p>
<p>Blanks</p>	<p>All blanks in a list-directed input record are considered to be part of some value separator, except for the following:</p> <ul style="list-style-type: none"> • Blanks embedded in a character constant. • Leading blanks in the first record read by each execution of a list-directed input statement unless immediately followed by a slash or comma.

The following example uses list-directed input and output:

```
REAL    a
INTEGER i
```

```

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

```

The preceding program produces the following output:

```

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

```

List-Directed Output

In list-directed output, you provide the *iolist* and Visual Fortran provides the formatting. The formatting is the same as that required for input, except as noted in this section.

New records are created as necessary, but neither the end of a record nor blanks can occur within a constant (except in character constants). To provide carriage control when the record is printed, each output record automatically begins with a blank character.

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

Default Formats for List-Directed Output	
Data Type	Output Format
BYTE	I5
LOGICAL(1)	L2
LOGICAL(2)	L2
LOGICAL(4)	L2
LOGICAL(8) ¹	L2
INTEGER(1)	I5
INTEGER(2)	I7
INTEGER(4)	I12
INTEGER(8) ¹	I22
REAL(4)	1PG15.7E2
REAL(8) T_floating	1PG24.15E3
REAL(8) D_floating	1PG24.16E2
REAL(8) G_floating	1PG24.15E3
REAL(16) ²	1PG43.33E4
COMPLEX(4)	'(,1PG14.7E2, ', ',1PG14.7E2, ')'
COMPLEX(8) T_floating	'(,1PG23.15E3, ', ',1PG23.15E3, ')'
COMPLEX(8) D_floating	'(,1PG23.16E2, ', ',1PG23.16E2, ')'
COMPLEX(8) G_floating	'(,1PG23.15E3, ', ',1PG23.15E3, ')'

CHARACTER	A _w ³
-----------	-----------------------------

¹Alpha only.

²VMS, U*X.

³Where *w* is the length of the character expression.

A derived type in an output list is equivalent to putting all the elements of the derived type in the output, in the same order they have in the derived-type declaration.

The following example uses list-directed output:

```

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

```

Namelist I/O

Namelist I/O is a powerful method for reading data in or writing data out to a file (or the terminal). By specifying one or more variables in a namelist group, you can read or write the values of all of them with a single I/O statement.

A namelist group is created with the **NAMELIST** statement. It takes the form:

NAMELIST / *namelist* / *variable-list*

The *namelist* is an identifying name for the group, and *variable-list* is a list of variables, derived types and/or array names.

Namelist I/O is described in the following sections:

- [Namelist Input](#)
- [Namelist Output](#)
- [Namelist READ](#)

See also [NAMELIST](#) in the *Reference*.

Namelist Input

A namelist input statement scans the input file for the group name. After it finds the group name, the statement then scans for assignment statements that give values to one or more of the variables in the group. Namelist input starts with an ampersand (&) or dollar sign (\$).

Namelist input is terminated with a slash (/), ampersand (&), or dollar sign (\$). The word END in uppercase, lowercase or mixed-case can appear after the terminating ampersand (&) or dollar sign (\$) with no intervening spaces, but cannot appear after a terminating slash (/). For example:

```
INTEGER a, b
NAMELIST /mynml/ a, b
```

...

! The following are all valid namelist variable assignments:

```
&mynml a = 1 /
$mynml a = 1 $
$mynml a = 1 $end
&mynml a = 1 &
&mynml a = 1 $END
&mynml
a = 1
b = 2
/
```

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

Namelist Output

A namelist output statement writes the name of the namelist group, followed by the name of each variable in the namelist, an equal sign (=), and the variable's current value. Namelist output is terminated with a slash (\). The values of the namelist variables are written to a file or the screen with a **WRITE** statement in which the *namelist* group name appears instead of a format specifier. Note that no *iolist* is needed or permitted.

```
WRITE (*, [NML=] namelist)
```

NML= is optional and is required only if other keywords (such as **END=**) are used.

The first output record is an ampersand (&), immediately followed by the namelist group name, in uppercase. Succeeding records list all variable names in the group and their values. Each output record begins with a blank character to provide carriage control if the record is printed. The last output record is a slash.

Values take the output format they would have in list-directed I/O. Unless **DELIM= 'QUOTE'** or **'APOSTROPHE'** when the output file is opened, character constants will not be delimited, and the file created cannot be read by a namelist **READ** statement, which requires string delimiters.

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

```
INTEGER(1) int1
INTEGER    int2, int3, array(3)
LOGICAL(1) log1
LOGICAL    log2, log3
REAL      real1
REAL(8)    real2
```

```

COMPLEX z1, z2
CHARACTER(1) char1
CHARACTER(10) char2

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

int1      = 11
int2      = 12
int3      = 14
log1      = .TRUE.
log2      = .TRUE.
log3      = .TRUE.
real1     = 24.0
real2     = 28.0d0
z1        = (38.0,0.0)
z2        = (316.0d0,0.0d0)
char1     = 'A'
char2     = '0123456789'
array(1)  = 41
array(2)  = 42
array(3)  = 43
WRITE (*, example)

```

Output of the preceding example is:

```

&EXAMPLE
INT1 = 11,
INT2 =      12,
INT3 =      14,
LOG1 = T,
LOG2 = T,
LOG3 = T,
REAL1 = 24.00000 ,
REAL2 = 28.000000000000000 ,
Z1 = (38.00000,0.0000000E+00),
Z2 = (316.0000,0.0000000E+00),
CHAR1 = A,
CHAR2 = 0123456789,
ARRAY =      41,      42,      43
/

```

Namelist READ

The operation of a namelist **READ** statement is almost the reverse of a **WRITE** operation. The statement first scans the file (either at the terminal or on disk) from its current position until it finds an ampersand (&) immediately followed by the *namelist* group name, or until it reaches the end of the file. (Ampersands followed by other names are ignored.) There must be at least one blank or carriage return following the group name to separate it from the following value-assignment pairs.

A value-assignment pair consists of a variable name, array element, or substring followed by an equal sign (=) and one or more values and value separators. The equal sign can be preceded or followed by any number of blanks (including no blanks). A value separator is a single comma (,), one or more blanks, or a tab. A comma that is not preceded by a value is treated as a null value, and the corresponding variable or array element is not altered.

Variables can appear in any order. The same variable can appear in more than one assignment pair. Its final value is the value it received in its last assignment. All the variables in a namelist do not

need to be assigned values; those that do not appear, or that are associated with null values, keep their current values. A variable name in the input file that is not in the namelist group causes a run-time error.

If a derived-type name appears in the input list, the first value in the record is given to the first element in the derived-type definition, the second value to the second element and so on. The input data types must match the element types. An individual derived-type element can also appear in an input list, like any other variable.

If an array name appears without a qualifying subscript, the first value in the input record is given to the first array element, the second to the second element, and so forth. Assignment to arrays is by row-major order.

You cannot assign more values than there are elements in an array. For example, you cannot specify 101 values for a 100-element array. However, an array need not have values assigned to all its elements. Any missing values are treated as nulls, and the corresponding array elements are not altered. Individual values can also be assigned to subscripted array elements.

A value can be repeated by placing a repeat factor and an asterisk (*) in front of the value. For example, 7*'Hello' assigns 'Hello' to the next seven elements in an array or variable list. A repeat factor and asterisk without a value indicates multiple null values. The corresponding variables are not altered. Consider the following array matrix(0:101):

```
matrix = 10, 50*25, 50*, -101
matrix(42) = 63
```

These statements assign 10 to element 0, assign 25 to the elements 1 through 50, leave 51 through 100 alone, assign -101 to element 101, and then change the value of matrix(42) to 63.

Character strings must be delimited by apostrophes or quotation marks to be read.

A namelist **READ** statement is terminated by a slash (/) or when the end of the file is reached. If the **READ** statement reaches the end of the file, an error occurs. Don't use slashes as valueseparators unless you want to end the read prematurely.

Suppose you wanted the [preceding program](#) to read new values for some of the variables in *namelist* group *example*. Consider that a file connected to unit four contains the following namelist specifier and assignment statements:

```
&example
Z1 = (99.0,0.0)
INT1=99
array(1)=99
REAL1 = 99.0
CHAR1='Z'
CHAR2(4:9) = 'Inside'
LOG1=.FALSE.
/
```

In this case, the following namelist **READ** statement would assign new values to the specified variables:

```
READ (UNIT = 4, example)
```

A second **WRITE** (*, example) statement would display their changed values, as follows:

```
&example
INT1   =    99,
INT2   =           12,
INT3   =           14,
LOG1   = F,
LOG2   = T,
LOG3   = T,
REAL1  =  99.00000 ,
REAL2  = 28.000000000000000 ,
Z1     = (99.00000,0.0000000E+00),
Z2     = (316.0000,0.0000000E+00),
CHAR1  = Z,
CHAR10 = 012Inside9,
ARRAY  =           99,           42,           43
/
```

Input/Output Statements

In the input/output (I/O) system, data is stored in and retrieved from devices and files. I/O editing determines how that data is organized when it is read from the files and devices or written to them. I/O statements determine what I/O operations are performed on the data. This topic describes Visual Fortran I/O statements and discusses the specifiers you can use to modify the operations they perform.

The I/O statements are listed in a table. The I/O statement specifiers are also listed in a table.

See also Improve Overall I/O Performance.

Overview of I/O Statements

The following are data transfer statements: **READ**, **ACCEPT**, **WRITE**, **PRINT (or TYPE)**, and **REWRITE**.

The following are file connection, inquiry, and positioning statements: **BACKSPACE**, **CLOSE**, **DELETE**, **ENDFILE**, **INQUIRE**, **OPEN**, **REWIND**, and **UNLOCK**.

The following table gives a brief description of the Visual Fortran I/O statements. For more detailed descriptions of each statement, see the *Reference*.

Statement	Function
<u>ACCEPT</u>	<u>Inputs data. Similar to a formatted, sequential READ statement.</u>
<u>BACKSPACE</u>	Positions a file at the beginning of the preceding record
<u>CLOSE</u>	Disconnects a unit
<u>DELETE</u>	<u>Deletes a record from a relative file.</u>
<u>ENDFILE</u>	Writes an end-of-file record
<u>INQUIRE</u>	Returns the properties of a unit or external file
<u>OPEN</u>	Associates a unit number with a file or device
<u>PRINT (or TYPE)</u>	Outputs data to the asterisk (*) unit
<u>READ</u>	Inputs data
<u>REWIND</u>	Repositions a file to the beginning
<u>REWRITE</u>	<u>Rewrites the current record.</u>
<u>UNLOCK</u>	<u>Frees a record in a relative or sequential file that was previously locked.</u>
<u>WRITE</u>	Outputs data

The EOF intrinsic function can be used to determine whether there is data remaining in the file after the current position.

I/O Statement Specifiers

I/O statements can be modified with optional parameters. For example, consider the following **OPEN** statement:

```
OPEN (UNIT= 4, FILE= 'BESSEL.DAT', POSITION= 'APPEND')
```

This **OPEN** statement associates the file BESSEL.DAT with unit number 4. It also uses the optional **POSITION='APPEND'** to position the opened file at the end. Any subsequent **WRITE** to the file will append data to the end, and not overwrite existing data.

Optional I/O parameters let you designate file structure, position and access, data delimiters and blank interpretation, error handling and many other I/O features. With I/O specifiers, you can modify I/O operations to fit your needs. The following table lists the specifiers, the values they can take, what they do, and the I/O statements they are used in. Further information about the options can be found in the *Reference* under the I/O statements that use them.

Table: I/O Specifiers			
Specifier	Values	Description	Used with:
<u>ACCESS</u> = <i>access</i>	'SEQUENTIAL', 'DIRECT', or 'APPEND'	Specifies the method of file access.	<u>INQUIRE</u> , <u>OPEN</u>
<u>ACTION</u> = <i>permission</i>	'READ', 'WRITE' or 'READWRITE' (default is 'READWRITE')	Specifies file I/O mode.	<u>INQUIRE</u> , <u>OPEN</u>
<u>ADVANCE</u> = <i>ad_switch</i>	'NO' or 'YES' (default is 'YES')	Specifies formatted sequential data input as advancing, or non-advancing.	<u>READ</u>
<u>ASSOCIATEVARIABLE</u> = <i>var</i>	Integer variable	Specifies a variable to be updated to reflect the record number of the next sequential record in the file.	<u>INQUIRE</u> , <u>OPEN</u>
<u>BINARY</u> = <i>bin</i>	'NO' or 'YES'	Returns whether file format is binary.	<u>INQUIRE</u>
<u>BLANK</u> = <i>blank_control</i>	'NULL' or 'ZERO' (default is 'NULL')	Specifies whether blanks are ignored in numeric fields or interpreted as zeros.	<u>INQUIRE</u> , <u>OPEN</u>
<u>BLOCKSIZE</u> = <i>blocksize</i>	Positive integer variable or expression	Specifies or returns the internal buffer size used in I/O.	<u>INQUIRE</u> , <u>OPEN</u>
<u>BUFFERCOUNT</u> = <i>bc</i>	Numeric expression	Specifies the number of buffers to be associated with the unit for multibuffered I/O.	<u>OPEN</u>
<u>CARRIAGECONTROL</u> = <i>control</i>	'FORTRAN', 'LIST', or 'NONE'	Specifies carriage control processing.	<u>INQUIRE</u> , <u>OPEN</u>
<u>CONVERT</u> = <i>form</i>	'LITTLE_ENDIAN', 'BIG_ENDIAN', 'CRAY', 'FDX', 'FGX', 'IBM', 'VAXD', 'VAXG' or 'NATIVE' (default is 'NATIVE')	Specifies a numeric format for unformatted data.	<u>INQUIRE</u> , <u>OPEN</u>
<u>DEFAULTFILE</u> = <i>var</i>	Character expression	Specifies a default file pathname string.	<u>INQUIRE</u> , <u>OPEN</u>

<u>DELIM</u> = <i>delimiter</i>	'APOSTROPHE', 'QUOTE' or 'NONE' (default is 'NONE')	Specifies the delimiting character for list-directed or namelist data.	<u>INQUIRE</u> , <u>OPEN</u>
<u>DIRECT</u> = <i>dir</i>	'NO' or 'YES'	Returns whether file is connected for direct access.	<u>INQUIRE</u>
<u>DISPOSE</u> = <i>dis</i> (or <u>DISP</u> = <i>dis</i>)	'KEEP', 'SAVE', 'DELETE', 'PRINT', 'PRINT/DELETE', 'SUBMIT', or 'SUBMIT/DELETE' (default is 'DELETE' for scratch files; 'KEEP' for all other files)	Specifies the status of a file after the unit is closed.	<u>OPEN</u> , <u>CLOSE</u>
<i>formatlist</i>	Character variable or expression	Lists edit descriptors. Used in FORMAT statements and format specifiers (the FMT = <i>formatspec</i> option) to describe the format of data.	<u>FORMAT</u> , <u>PRINT</u> , <u>READ</u> , <u>WRITE</u>
<u>END</u> = <i>endlabel</i>	Integer between 1 and 99999	When an end of file is encountered, transfers control to the statement whose label is specified.	<u>READ</u>
<u>EOR</u> = <i>eorlabel</i>	Integer between 1 and 99999	When an end of record is encountered, transfers to the statement whose label is specified.	<u>READ</u>
<u>ERR</u> = <i>errlabel</i>	Integer between 1 and 99999	Specifies the label of an executable statement where execution is transferred after an I/O error.	All except PRINT
EXIST = <i>ex</i>	.TRUE. or .FALSE.	Returns whether a file exists and can be opened.	<u>INQUIRE</u>
<u>FILE</u> = <i>file</i> (or <u>NAME</u> = <i>name</i>)	Character variable or expression. Length and format of the name are determined by the operating system	Specifies the name of a file	<u>INQUIRE</u> , <u>OPEN</u>
[FMT]= <i>formatspec</i>	Character variable or expression	Specifies an <i>editlist</i> to use to format data.	<u>PRINT</u> , <u>READ</u> , <u>WRITE</u>
<u>FORM</u> = <i>form</i>	'FORMATTED', 'UNFORMATTED', or 'BINARY'	Specifies a file's format.	<u>INQUIRE</u> , <u>OPEN</u>
<u>FORMATTED</u> = <i>fmt</i>	'NO' or 'YES'	Returns whether a file is connected for formatted data transfer.	<u>INQUIRE</u>
<u>IOFOCUS</u> = <i>iof</i>	.TRUE. or .FALSE.	Specifies whether a unit is the	<u>INQUIRE</u> ,

	(default is <code>.TRUE.</code> unless unit <code>'*' is specified)</code>	active window in a QuickWin application.	<u>OPEN</u>
<u><i>iolist</i></u>	List of variables of any type, character expression, or <code>NAMELIST</code>	Specifies items to be input or output.	<u>PRINT</u> , <u>READ</u> , <u>WRITE</u>
<u>IOSTAT</u> = <i>iostat</i>	Integer variable	Specifies a variable whose value indicates whether an I/O error has occurred.	All except PRINT
<u>MAXREC</u> = <i>var</i>	Numeric expression	Specifies the maximum number of records that can be transferred to or from a direct access file.	<u>OPEN</u>
<u>MODE</u> = <i>permission</i>	' <code>READ</code> ', ' <code>WRITE</code> ' or ' <code>READWRITE</code> ' (default is ' <code>READWRITE</code> ')	Same as ACTION .	<u>INQUIRE</u> , <u>OPEN</u>
<u>NAMED</u> = <i>var</i>	<code>.TRUE.</code> or <code>.FALSE.</code>	Returns whether a file is named.	<u>INQUIRE</u>
<u>NEXTREC</u> = <i>nr</i>	Integer variable	Returns where the next record can be read or written in a file.	<u>INQUIRE</u>
[<u>NML</u> =] <i>nmlspec</i>	Namelist name	Specifies a <i>namelist</i> group to be input or output.	<u>PRINT</u> , <u>READ</u> , <u>WRITE</u>
<u>NUMBER</u> = <i>num</i>	Integer variable	Returns the number of the unit connected to a file.	<u>INQUIRE</u>
<u>OPENED</u> = <i>od</i>	<code>.TRUE.</code> or <code>.FALSE.</code>	Returns whether a file is connected.	<u>INQUIRE</u>
<u>ORGANIZATION</u> = <i>org</i>	' <code>SEQUENTIAL</code> ' or ' <code>RELATIVE</code> ' (default is ' <code>SEQUENTIAL</code> ')	Specifies the internal organization of a file.	<u>INQUIRE</u> , <u>OPEN</u>
<u>PAD</u> = <i>pad_switch</i>	' <code>YES</code> ' or ' <code>NO</code> ' (default is ' <code>YES</code> ')	Specifies whether an input record is padded with blanks when the input list or format requires more data than the record holds, or whether the input record is required to contain the data indicated.	<u>INQUIRE</u> , <u>OPEN</u>
<u>POSITION</u> = <i>file_pos</i>	' <code>ASIS</code> ', ' <code>REWIND</code> ' or ' <code>APPEND</code> ' (default is ' <code>ASIS</code> ')	Specifies position in a file.	<u>INQUIRE</u> , <u>OPEN</u>
<u>READ</u> = <i>rd</i>	' <code>NO</code> ' or ' <code>YES</code> '	Returns whether a file can be read.	<u>INQUIRE</u>
<u>READONLY</u>		Specifies that only READ statements can refer to this connection.	<u>OPEN</u>

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

<code>WRITE=rd</code>	'NO' or 'YES'	Returns whether a file can be written to.	<u>INQUIRE</u>
-----------------------	---------------	---	----------------

Format and Namelist Specifiers

The format and namelist specifiers are: `FMT=` and `NML=`.

These specifiers indicate how I/O statements interpret data. The format specifier (`FMT=`) can be a character expression containing edit descriptors or the label of a **FORMAT** statement containing edit descriptors. For information on formats see [Formatted I/O](#). The namelist specifier (`NML=`) directs I/O statements to use a namelist, a group of variables in a certain order, as the template for input and output. For more information on namelists, see [Namelist I/O](#).

Only one of these I/O descriptors (`FMT=` or `NMT=`) can be used in a single I/O statement.

Format Specifier (FMT=)

The `FMT=` specifier is used in **PRINT**, **READ**, and **WRITE** statements to specify the format itself or to refer to the label of a **FORMAT** statement. The `FMT=` syntax can be omitted, but if it is, the format specifier must be the second parameter in **READ** and **WRITE** statements (after the unit specifier). If the `FMT=` syntax is used, the format specifier can appear anywhere in the I/O statement's argument list.

```
! FMT= omitted, formatting must come after unit specifier.
  READ (8, '(2I5)') int1, int2

! FMT= present, formatting can appear anywhere in the argument list.
  WRITE (8, ERR= 200, IOSTAT= ios, FMT= 800) int1, int2
800  FORMAT(2I5)
```

Namelist Specifier (NML=)

The **NAMELIST** specifier in **WRITE** or **READ** statements replaces a **FORMAT** specifier. If `NML=` (like `FMT=`) is absent, a namelist specifier must be the second parameter after the unit specifier. If `NML=` is present, the namelist can appear anywhere in the I/O statement's argument list. With a namelist, you can read or write the values of all the variables in the namelist with a single I/O statement. The following is a sample namelist:

```
INTEGER a, b
NAMELIST /exnml / a, b
a = 1
b = 2
WRITE (*, exnml)
```

Outputs the following:

```
&EXNML
A =      1,
B =      2
/
```

Errors, End-of-File, and End-of-Record Handling Specifiers

The errors, end-of-file, and end-of-record handling specifiers are: `ERR=`, `END=`, `EOR=`, and

IOSTAT=.

When you use these specifiers, your program can recover from conditions that would otherwise terminate execution. These specifiers control how your program responds to errors, end-of-file, and end-of-record conditions encountered during I/O operations:

- **ERR=** identifies a statement label to which control should transfer if an error occurs. **ERR=** can be used with any I/O statement except **PRINT**.
- **END=** identifies a statement label to which control should transfer if an end of file is encountered. **END=** can only be used in a sequential access **READ** statement. If you **WRITE** to a file after reaching the end of file, the data will be appended to the file.
- **EOR=** identifies a statement label to which control should transfer if an end of record is encountered during a **READ** statement with **ADVANCE='NO'**. **EOR=** can only be used with a non-advancing **READ** statement. A non-advancing **READ** can only be performed on a file opened as formatted sequential (the default).
- **IOSTAT=** sets the value of *iostat* to -1 if end-of-file is encountered, to -2 if an end-of-record is encountered, and to the run-time error number (a positive integer) if an error occurs. (See [Run-time Errors](#) for a list of error numbers and their meaning.) If none of these conditions occurs, *iostat* is set to 0. **IOSTAT=** can be used with any I/O statement except **PRINT**.

If none of these options has been used and there is a run-time error during the I/O operation, the program will terminate with a run-time error message. Because you cannot specify **ERR=**, **END=**, **EOR=**, and **IOSTAT=** specifiers with the **PRINT** statement, an error during execution of a **PRINT** statement always causes a run-time error.

The following table indicates the action taken when an error or end-of-file marker is encountered by a **READ** statement. Any time an error occurs during a **READ** statement, all items in the *iolist* become undefined.

IOSTAT= set	END= set	ERR= set	End-of-file occurs	Run-time error occurs
No	No	No	Run-time error occurs	Program termination with run-time error message
No	No	Yes	Go to <i>errlabel</i>	Go to <i>errlabel</i>
No	Yes	No	Go to <i>endlabel</i>	Program termination with run-time error message
No	Yes	Yes	Go to <i>endlabel</i>	Go to <i>errlabel</i>
Yes	No	No	Set <i>iostat</i> = -1 and execute next statement	Set <i>iostat</i> = run-time error number and execute next statement
Yes	Yes	Yes	Set <i>iostat</i> = -1 and go to <i>endlabel</i>	Set <i>iostat</i> = run-time error number and go to <i>errlabel</i>

An advancing **READ** statement reads a file that has been opened without an **ADVANCE=** specifier or with **ADVANCE='YES'**. When an advancing **READ** statement encounters an end-of-record marker, the file is positioned after the current record and execution continues. A nonadvancing **READ** statement, one reading a file opened with **ADVANCE='NO'**, is character-oriented, not record-oriented, and when it encounters an end-of-record marker an error occurs. For example,

consider the following:

```
CHARACTER(8) string
INTEGER ios

OPEN (4, FILE= "Mydat.dat")
WRITE(4, *) 'abc'
REWIND(4)
READ(4, '(A8)', ADVANCE= 'NO', EOR= 200, IOSTAT = ios) string
```

The **READ** statement attempts to input eight characters (the A8 edit descriptor), but the record holds only three characters. So, at the **READ** statement, the program will set *ios* to -2 and branch to the statement at label 200, specified by the **EOR=200** option.

The following table shows the action taken when a nonadvancing **READ** statement encounters an end-of-record marker, if no other error condition occurs. (An error or end-of-file condition takes precedence over an end-of-record condition, and the program behaves as described in [Table 9.3](#).)

IOSTAT set	EOR set	End-of-Record occurs
No	No	Program termination with run-time error message
No	Yes	Set SIZE= , if specified, to the number of characters read and go to <i>eorlabel</i>
Yes	No	Set <i>iostat</i> = -2 and execute next statement
Yes	Yes	Set <i>iostat</i> = -2, set SIZE= , if specified, to the number of characters read, and go to <i>eorlabel</i>

The following table shows what happens when an error occurs during any I/O statement other than **READ** or **PRINT**:

Situation	Result
Neither <i>errlabel</i> nor <i>iostat</i> is present	The program is terminated, and a run-time error message is given
Only <i>errlabel</i> is present	Control is transferred to the statement at <i>errlabel</i>
Only <i>iostat</i> is present	The value of <i>iostat</i> is set to the run-time error number and control is returned as if the statement had executed without error
Both <i>errlabel</i> and <i>iostat</i> are present	The value of <i>iostat</i> is set to the run-time error number and control is transferred to the statement at <i>errlabel</i>

If an I/O statement terminates without encountering an error, end-of-file record, or an end-of-record marker, and if *iostat* is specified, *iostat* is set to zero.

In the following example, no available specifiers (**ERR=**, **END=**, **EOR=**, or **IOSTAT=**) are set. So, if an invalid value is entered for *i* (for example, the user enters a character string, such as abc), a run-time error occurs:

```
INTEGER i
WRITE (*, *) 'Please enter i'
READ (*, *) i
! If the user has entered abc the program will terminate
! with error "list-directed I/O syntax error" and doesn't execute the WRITE below.
WRITE (*, *) 'This is i:', i
```

```
END
```

The following example uses the **ERR=** option to prompt the user to enter a valid number:

```

      INTEGER i
      WRITE (*, *) 'Please enter i:'
50    READ (*, *, ERR = 100) i
      WRITE (*, *) 'This is i:', i
      STOP ' '
100   WRITE (*, *) 'Invalid value. Please enter new i:'
      GOTO 50
      END
```

This example uses both the **ERR=** and **IOSTAT=** specifiers to handle invalid input:

```

      INTEGER i, j
      WRITE (*, *) 'Please enter i:'
50    READ (*, *, ERR = 100, IOSTAT = j) i
      WRITE (*, *) 'This is i:', i, ' iostat = ', j
      STOP ' '
100   WRITE (*, *) 'Failed with error #', j
      WRITE (*, *) 'Please enter new i:'
      GOTO 50
      END
```

You can also control end-of-file with the **ENDFILE** statement. **ENDFILE** inserts an end-of-file record at the file's current position, then positions the file after the end-of-file record. Any data past this position is lost.

Record Specifiers

The record specifiers are: **REC=**, **RECL=** (or **RECORDSIZE=**), **DELIM=**, **BLANK=**, **PAD=**, **RECORDTYPE=**, **MAXREC=**, and **ASSOCIATEVARIABLE=**.

The **REC=***recnum* option specifies a record number in a direct access file. In **READ** and **WRITE** statements, *recnum* specifies the first record to be read or written. The first record in a file is record number 1.

The **RECL=length** specifies the record length in bytes for direct access files, and the maximum record length in bytes for sequential access files. **RECL=** is optional when opening a file for sequential access, but required for direct access files. The **RECL=** value unit for formatted files is always 1-byte units. For unformatted files, the **RECL=** unit is 4-byte units, unless you specify the compiler option `/assume:byterecl` to request 1-byte units.

RECORDSIZE= is a nonstandard synonym for **RECL=**.

The **DELIM=** specifier sets the record delimiter in list-directed and namelist formatted output. **DELIM=** is ignored on input. The options are **DELIM='APOSTROPHE'**, **'QUOTE'**, and **NONE'**. The default is **'NONE'**. If the **DELIM=** specifier is set to **'APOSTROPHE'** each occurrence of an apostrophe (') within a character constant is doubled. If the **DELIM=** specifier is set to **'QUOTE'** each occurrence of a quote within a character constant is doubled. In namelist input, character strings must be delimited by apostrophes or quotes or they will not be interpreted as strings. Unless you specify **DELIM='QUOTE'** or **'APOSTROPHE'** when a output file is opened, character constants

written to the file will not be delimited, and the file created cannot be read by a namelist **READ** statement.

The **BLANK=** specifier controls the interpretation of blanks within numeric data in formatted I/O. If **BLANK='NULL'** blanks are ignored in numeric fields. If **BLANK='ZERO'** all blanks are interpreted as zeros except leading blanks. The default is 'NULL'.

The **PAD=** specifier determines whether input will be blank padded if the input record contains less data than the input list or format requires. If **PAD='YES'**, blanks will be added. This is the default. If **PAD='NO'**, blanks will not be added, and the input record must contain the data indicated by the input list or format. **PAD=** has no effect on output.

The **RECORDTYPE=** specifier indicates the type of records in a file. The options are 'FIXED', 'VARIABLE', 'SEGMENTED', 'STREAM', 'STREAM_LF', AND 'STREAM_CR'. When you open a file, the defaults are 'FIXED' for relative files and direct access sequential files, 'STREAM_LF' for formatted sequential access files, and 'VARIABLE' for unformatted sequential access files.

The **MAXREC=** specifier indicates the maximum number of records that can be transferred to or from a direct access file while the file is connected. The default is an unlimited number of records.

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. The argument cannot be a dummy argument to the routine in which the **OPEN** statement appears. Direct access **READs**, direct access **WRITEs**, and the **FIND**, **DELETE**, and **REWRITE** statements can affect the value of the argument. This specifier is only valid for direct access; it is ignored for other access modes.

Input/Output Buffer Size Specifiers

The input/output buffer size specifiers are: **BLOCKSIZE=** and **BUFFERCOUNT=**.

The **BLOCKSIZE=** specifier indicates the input/output buffer size in bytes. In Windows NT and Windows 95, I/O operations are buffered by the operating system, which makes **BLOCKSIZE=** far less important than it used to be. Under older operating systems, the speed of I/O operations could be improved by increasing the I/O buffer size, because a large buffer reduced the total number of reads and writes needed to transfer a given amount of data. However, with virtual memory, the amount of memory accessible to a process is usually not an issue, except that memory swapping can cause a program to execute more slowly as memory use increases.

Increasing the size of the I/O buffers will not have much effect on I/O execution speed. The default buffer size is 1024 bytes. Because the buffers are only allocated when the file is opened, using the option **BLOCKSIZE=** does not affect the size of the program's executable file.

The **BUFFERCOUNT=** specifier indicates the number of buffers to be associated with the unit for multibuffered I/O. 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 the argument, the default is 1.

Carriage Control Specifier

The carriage control specifier is: **CARRIAGECONTROL=**.

This specifier indicates the type of carriage control used when a file is displayed at a terminal. The options are 'FORTRAN' (normal Fortran interpretation of first character), 'LIST' (one line feed between records), and 'NONE' (no carriage-control processing).

The default for unformatted and binary 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.

The following table lists the valid control characters for printing:

Table: Carriage-Control Characters	
Character	Effect
space	Outputs the record (at the current position in the current line) and a carriage return.
0	Advances two lines. Outputs the record and a carriage return.
1	Advances to top of next page. Outputs the record and a carriage return.
+	Does not advance. Outputs the record and a carriage return. Permits overprinting.
\$	Advances to top of next line. Outputs the record, but no carriage return. Permits prompting.
ASCII NUL ¹	Does not advance. Outputs the record, but no carriage return. Permits overprinting.

¹ Specify as CHAR(0).

Any other character is interpreted as a blank and is deleted from the print line. If you do not specify a control character for printing, the first character of the record is not printed.

See also /fpscomp, for information on how this compiler option can affect carriage control.

QuickWin Specifiers

The QuickWin specifiers are: **IOFOCUS=** and **TITLE=**.

The **IOFOCUS=** specifier indicates whether the unit is the active window in a QuickWin application. The default is `.TRUE`. A value of `.TRUE` causes a call to **SETFOCUSQQ** immediately before any **READ**, **WRITE**, or **PRINT** statement to that window. See also Giving a Window Focus and Setting the Active Window in Using QuickWin.

The **TITLE=** specifier indicates the name of a child window in a QuickWin application.

File Property Specifiers

You can specify the following file properties by using I/O specifiers:

- [Filenames](#)
- [File Status and Disposition](#)
- [File Structure](#)
- [File Access Methods](#)
- [File Access Privileges](#)
- [File Sharing](#)
- [File Data Transfer Methods](#)
- [File Position](#)
- [File Numeric Format](#)

Specifying Filenames

The filenames specifiers are: **FILE=** (or **NAME=**) and **DEFAULTFILE=**.

The name of an internal file is the name of the character variable, character array, or **noncharacter array** that makes up the file. The name of an external file must be a character string that the operating system recognizes as a filename (including device names). If you do not specify a path, the operating system uses the current working directory. External filenames must follow the filenaming conventions of the host operating system. Wildcards are not permitted.

In Windows NT and Windows 95, a filename can be longer than eight characters and have an extension longer than three characters. The filename specified with **FILE=** can take these forms:

- filename
- path\filename
- drive:\path\filename
- \\server\path\filename

The filename specification (including the path, drive, or share) can be up to **\$MAXPATH** in length. **\$MAXPATH** is defined in module DFLIB.

Visual Fortran provides the following possible ways of specifying all or part of a file specification (full pathname), such as \proj\testdata:

- The **FILE=** keyword in an **OPEN** statement typically specifies only a file name (such as testdata) or a pathname that contains both a directory and file name (such as \proj\testdata).
- The **DEFAULTFILE=** keyword in an **OPEN** statement typically specifies a pathname that contains only a directory (such as d:\proj) or both a directory and file name (such as d:\proj\testdata).
- If you used an implied **OPEN** or if the **FILE=** keyword in an **OPEN** statement did not specify a file name, you can use an environment variable to specify a file name or a pathname that contains both a directory and file name.

Visual Fortran recognizes environment variables for each logical I/O unit number, in the form **FORT n** , where n is the logical I/O unit number. If a file name is not specified in the **OPEN** statement and the corresponding **FORT n** environment variable is not set for that unit number, Visual Fortran generates a file name in the form **FORT. n** , where n is the logical unit number.

Certain Fortran environment variables are recognized and preconnected files exist for certain unit numbers.

Performing an implied **OPEN** means that the **FILE=** and **DEFAULTFILE=** keyword values are not specified and an environment variable is used, if present.

Rules for Applying Default File Specifications

Visual Fortran determines file name and the pathname based on certain rules. It determines a file name string as follows:

- If the **FILE=** keyword is present, its value is used.
- If the **FILE=** keyword is not present, Visual Fortran examines the corresponding environment variable.
 - If the corresponding environment variable is set, that value is used.
 - If the corresponding environment variable is not set, a file name in the form `fort.n` is used.

Once Visual Fortran determines the resulting file name string, it determines the directory (which optionally precedes the file name) as follows:

- If the resulting file name string contains an absolute pathname, it is used and the **DEFAULTFILE=** keyword, environment variable, and current directory values are ignored.
- If the resulting file name string does not contain an absolute pathname, Visual Fortran examines the **DEFAULTFILE=** keyword and current directory value:
 - If the corresponding environment variable is set and specifies an absolute pathname, Visual Fortran uses that value.
 - The **DEFAULTFILE=** keyword value is examined and, if present, Visual Fortran uses its value.
 - If the **DEFAULTFILE=** keyword is not present, Visual Fortran uses the current directory as an absolute pathname.

Examples of Applying Default File Specifications

For example, for an implied **OPEN** of unit number 3, Visual Fortran would check the environment variable `FORT3`. If the environment variable `FORT3` was set, its value is used. If it was not set, the system supplies the file name `FORT.3`.

In the following table assume the current directory is `c:\users\smith` and the I/O uses unit 1, as in the statement:

```
READ (1,100)
```

OPEN FILE Value	OPEN DEFAULTFILE Value	FORT1 Environment Variable Value	Resulting Pathname
not specified	not specified	not specified	<code>c:\users\smith\fort.1</code> 1
not specified	not specified	<code>test.dat</code>	<code>c:\users\smith\test.dat</code> 2
not specified	not checked	<code>c:\temp\t.dat</code>	<code>c:\temp\t.dat</code> 3
not specified	<code>d:\temp</code>	not specified	<code>d:\temp\fort.1</code> 4
not specified	<code>d:\temp</code>	<code>testdata</code>	<code>d:\temp\testdata</code> 5
not specified	<code>d:\stable</code>	<code>lib\testdata</code>	<code>d:\stable\lib\testdata</code> 6
<code>file.dat</code>	<code>c:\user\group</code>	not checked	<code>c:\user\group\file.dat</code> 7

d:\temp\file.dat	not checked	not checked	d:\temp\file.dat 3
------------------	-------------	-------------	---------------------------

- 1** The current directory is used and the unit number determines the file name.
- 2** The current directory is used and the environment variable provides the file name.
- 3** The environment variable provides both the directory and file name.
- 4** The directory is provided by the **OPEN DEFAULTFILE= keyword value** and the unit number determines the file name.
- 5** The directory is provided by the **OPEN DEFAULTFILE= keyword value** and the environment variable provides the file name.
- 6** The directory is provided by the **OPEN DEFAULTFILE= keyword value** and the environment variable provides a subdirectory and file name.
- 7** The directory is provided by the **OPEN DEFAULTFILE= keyword value** and the file name is provided by the **OPEN FILE= keyword value**.
- 8** The directory and file name are provided by the **OPEN FILE= keyword value**.

See [I/O Hardware](#) in Files, Devices and I/O Hardware, for a list of devices. See also [/fpscomp](#), for information on how this compiler option can affect files and file names.

Specifying File Status and Disposition

The file status and disposition specifiers are: **STATUS=** and **DISPOSE=** (or **DISP=**).

A file can be opened with **STATUS='OLD'**, **'NEW'**, **'REPLACE'**, **'SCRATCH'**, **'UNKNOWN'**. A file can be closed with **STATUS='KEEP'** or **'DELETE'**. If **STATUS=** is absent in an **OPEN** statement, **OPEN** searches first for an existing file of the given name, and if such a file doesn't exist, creates a new one. If **STATUS=** is absent in a **CLOSE** statement, **CLOSE** defaults to **KEEP** unless the file was opened as a **SCRATCH** file, in which case **CLOSE** defaults to **DELETE**. For further information, see [OPEN](#) and [CLOSE](#) in the *Reference*.

The **DISPOSE=** specifier indicates the status of a file after the unit closes. The options are **'KEEP'** or **'SAVE'**, **'DELETE'**, **'PRINT'** (which prints and then saves the file), **'PRINT/DELETE'** (which prints and then deletes the file), **'SUBMIT'** (which forks a process to execute the file), and **'SUBMIT/DELETE'** (which forks a process to execute the file, then deletes it after the fork is completed). **'PRINT'** and **'PRINT/DELETE'** can only be used on sequential files. The default is **'DELETE'** for scratch files and **'KEEP'** for all other files.

Specifying File Structure

The file structure specifier is: **FORM=**.

The data is stored and retrieved in a file according to the file's access (set by the **ACCESS=** option

described in the section [File Access Methods](#)) and the form of the data the file contains. Files are structured in one of three ways: formatted, unformatted, or [binary](#). These formats are specified by setting **FORM**='FORMATTED', 'UNFORMATTED', or [BINARY](#):

- 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=** option of the **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.

The **FORM** defaults are as follows:

- If a file has been opened for sequential access, **FORM** defaults to 'FORMATTED'. If a file has been opened for direct access, **FORM** defaults to 'UNFORMATTED'.
- If neither the access nor the format specifiers are set, the file defaults to sequential and 'FORMATTED'.

Specifying File Access Methods

The file access methods specifiers are: **ACCESS=** and **ORGANIZATION=**.

Visual Fortran supports two methods of file access: sequential and direct. *Sequential files*, **ACCESS**='SEQUENTIAL', contain data recorded in the order in which it was written to the file. *Direct files*, **ACCESS**='DIRECT', are random-access files. Sequential access files have no fixed record size and are not positionable. Direct files have a fixed record size and can be positioned to any record. If no **ACCESS=** specifier is set, **ACCESS=** defaults to 'SEQUENTIAL'. [An existing file of either type can be opened with **ACCESS**='APPEND' to add new records after the last record in the file.](#)

The following statements read the third and fourth records of the direct-access file `xxx`:

```
OPEN (1, FILE = 'xxx', ACCESS = 'DIRECT', RECL = 15,      &
& FORM = 'FORMATTED')
READ (1, '(3I5)', REC = 3) i, j, k
READ (1, '(3I5)') l, m, n
```

The **ORGANIZATION=** specifier can also be used to indicate 'SEQUENTIAL' or 'RELATIVE' internal organization of a file. The default is 'SEQUENTIAL'.

Specifying File Access Privileges

The file access privileges specifiers are: **ACTION=** (or **MODE=**) and **READONLY**.

Use the **ACTION** option to declare which I/O operations you intend to perform on a file while you have it open. The value of **ACTION=** can be 'READ', 'READWRITE', or 'WRITE'. The default is 'READWRITE'. If you try to write to a file that you opened with **ACTION='READ'** or read a file opened with **ACTION='WRITE'**, you will get an error message. The **MODE=** option has the same effect as **ACTION=**.

The **READONLY** specifier indicates the file can only be read. This is the same as specifying **ACTION='READ'**.

Use the **INQUIRE** statement to determine the access permissions for a file.

The compiler option `/fpscomp` can affect **ACTION=** and **MODE=**.

Specifying File Sharing

The file sharing specifiers are: **SHARE=** and **SHARED=**.

In systems that use networking or allow multitasking, more than one program can try to access the same file at the same time. When this happens, the new user's program compares the intended I/O operations (declared in the **OPEN** statement's **ACTION=** option) to the file-sharing privileges given to other users by the program that opened the file first. If the intended use is permitted, the file can be opened. Otherwise, an error message is produced. File-sharing privileges established by the first user remain in force until all users have closed the shared file.

Concurrent users of a given file can be prevented from writing ('DENYWR'), reading ('DENYRD'), neither ('DENYNONE'), or both ('DENYRW'). 'COMPAT' is accepted for compatibility with previous versions. It is equivalent to 'DENYNONE'.

Use the **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'**.

The **SHARED=** specifier indicates the file is connected for shared access by more than one program executing simultaneously.

Specifying File Data Transfer Methods

The file data transfer method specifier is: **ADVANCE=**.

READ and **WRITE** operations on sequential files are by default advancing. This means that a file is automatically positioned at the beginning of the next record before data transfer and at the end of the record when input/output is completed. Advancing I/O can be set explicitly by setting **ADVANCE='YES'**. If you omit the **ADVANCE=** specifier, it defaults to 'YES'.

Nonadvancing **READ** and **WRITE** operations make it possible to read or write only part of a record. They leave the file positioned after the last character read or written instead of skipping to the end of the record upon completion. Nonadvancing I/O is selected by setting **ADVANCE='NO'** and can be used only for formatted sequential data transfer to external files or devices.

Nonadvancing input continues within the current record until an end-of-record condition occurs. If a nonadvancing **READ** attempts to input more data than a record holds and encounters an end-of-record marker, a run-time error will be generated. The program will terminate unless this condition is handled with an **EOR**, **ERR**, or **IOSTAT** specifier. If an end-of-record condition is encountered and the error is handled through one of the error specifiers, the **READ** can return the record length through the **SIZE=** specifier.

You cannot use the **ADVANCE=** specifier for I/O operations with **NAMELIST** or list-directed formatting. You also cannot use the **ADVANCE=** specifier for I/O operations on internal files. These I/O operations are all advancing, but you cannot use any **ADVANCE=** specifier with them, not even **ADVANCE='YES'**.

```
! Non-advancing READ example.
  INTEGER recsize
  CHARACTER(45) string
  READ (UNIT=4, FMT= '(A45)', ADVANCE='NO', SIZE=recsize,      &
& EOR=300) string
  WRITE (*,*) string
  STOP
! If the record size is less than 45 characters, the READ will
! encounter an END-OF-RECORD marker and branch to the statement at
! label 300, returning the true record size in recsize. A new READ
! of recsize characters then generates no error.
300  READ(4, '(A<recsize>)', ADVANCE= 'NO') string
```

Specifying File Position

The file position specifier is: **POSITION=**.

You can control file position either by opening a file with an explicit position specifier (**POSITION=**), or by using one of the I/O file position statements.

If you open a sequential file without a **POSITION=** specifier, the file is positioned at its beginning (unless the file was opened with **ACCESS='APPEND'**) If you then write to the file, all records after

the current record are discarded. With the **POSITION=** specifier you can control where the file is positioned on opening. Setting **POSITION='APPEND'** positions the file at its terminal point but just before an end-of-file record, if it exists.

Data written to the file will be appended to the end and will not erase previous records. **POSITION='REWIND'** positions the file at its initial point. **POSITION='ASIS'** means the file position of an opened file remains unchanged. This lets a new **OPEN** statement change connection options of an already opened file, without changing its position. If an unopened file is opened with **POSITION='ASIS'**, it is positioned at the beginning. New files are always positioned at the beginning, regardless of the **POSITION=** specifier. 'ASIS' is the default position value.

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.

Specifying File Numeric Format

The file numeric format specifier is: **CONVERT=**.

You can use **CONVERT=** to control whether numeric data in unformatted files is converted or not. This functionality can also be specified by using a compiler option. The conversion options are:

- 'LITTLE_ENDIAN' - Little endian integer and IEEE® floating-point data
- 'BIG_ENDIAN' - Big endian integer and IEEE® floating-point data
- 'CRAY' - Big endian integer and CRAY® floating-point data
- 'FDX' - Little endian integer and DIGITAL™ VAX™ F_floating, D_floating, and IEEE X_floating data
- 'FGX' - Little endian integer and DIGITAL VAX F_floating, G_floating, and IEEEEX_floating data
- 'IBM' - Little endian integer and IBM® System\370 floating-point data
- 'VAXD' - Little endian integer and DIGITAL VAX F_floating, D_floating, and H_floating data
- 'VAXG' - Little endian integer and DIGITAL VAX F_floating, G_floating, and H_floating data
- 'NATIVE' - No data conversion. This is the default.

You can use **CONVERT=** to specify multiple formats in a single program, usually one format for each specified unit number.

When reading a nonnative format, the nonnative format on disk is converted to native format in memory. If a converted nonnative value is outside the range of the native data type, a run-time message appears.

There are other ways to specify numeric format for unformatted files: you can specify an environment variable, the compiler option /convert, or **OPTIONS/CONVERT**. The following shows the order of precedence:

Method Used	Precedence
-------------	------------

An environment variable	Highest (1)
OPEN (CONVERT = <i>convert</i>)	2
OPTIONS/CONVERT	3
The /convert:keyword compiler option	Lowest (4)

The **/convert** compiler option and **OPTIONS/CONVERT** affect all unit numbers used by the program, while environment variables and **OPEN** (**CONVERT**=) affect specific unit numbers. The following example shows how to code the **OPEN** statement to read unformatted CRAY® numeric data from unit 15, which might be processed and possibly written in native little endian format to unit 20:

```

OPEN (CONVERT='CRAY', FILE='graph3.dat', FORM='UNFORMATTED',
1     UNIT=15)
...
OPEN (FILE='graph3_native.dat', FORM='UNFORMATTED', UNIT=20)

```

See Also: [Environment Variables Used with the DF Command](#), [Run-Time Environment Variables](#)

Using An External User-Written Function To Open A File

The external user-written function specifier is: **USEROPEN**=.

The **USEROPEN**=*function-name* specifier passes control to a user-written external function that directly opens a file. The called routine can use system calls or library routines to open the file and establish a special context that changes the effect of subsequent I/O statements.

The Visual Fortran Run-Time Library I/O support routines call the **USEROPEN**= function in place of the system calls usually used when the file is first opened for I/O. The called function must open the file (or pipe) and return the file descriptor of the file when it returns control to the calling program.

The *function-name* is the name of a user-written external open function. Although the called function can be written in other languages (such as Fortran), C is usually the best choice for making system calls, such as open or create.

In the calling Visual Fortran program, the function must be declared external. For example, the following statements could be used to call the **USEROPEN**= procedure UOPEN (known to the linker as `uopen_`):

```

EXTERNAL UOPEN
INTEGER UOPEN
...
OPEN (UNIT=10, FILE='/usr/test/data', STATUS='NEW', USEROPEN=UOPEN)

```

After the **OPEN** statement, the `uopen_` function receives control. The function opens the file, may perform other operations, and subsequently returns control (with the file descriptor) to the calling Visual Fortran program.

If the **USEROPEN**= function is written in C, declare it as a C function that returns a 4-byte integer (int) result to contain the file descriptor. For example:

```

int uopen_ (           /* function is declared as a 4-byte integer */

```

```
char *file_name,      /* 1st arg is the pathname (includes filename) to be opened
int  *open_flags,    /* flags are described in header file /usr/include/sys/fi
int  *create_mode,   /* the create mode protection */
int  *lun,           /* the logical unit number */
int  file_length);   /* the pathname length (hidden length argument of the path
```

Of the arguments, the `open` system call requires the passed pathname, the open flags (which define the type access needed, whether the file exists, and so on), and the create mode. The logical unit number specified in the **OPEN** statement is passed in case the called function needs it. The hidden length of the pathname is also passed.

Changing I/O Specifications with OPEN

You can change some I/O specifications by opening a unit that has already been connected to a file and indicating different values for the specifiers. The specifiers that can be modified in this way are **BLANK=**, **DELIM=**, **PAD=**, **ERR=**, and **IOSTAT=**. For example:

```
OPEN(4, DELIM='QUOTE')
```

New values for **BLANK=**, **DELIM=**, and **PAD=** are used in all subsequent data transfer statements. New values for **ERR=** and **IOSTAT=** affect only the **OPEN** statement being executed.

You cannot change the file association of a currently open unit or file.

General Compiler Directives

A compiler directive is a special statement that tells the compiler to take certain actions while compiling a program. You can also control compilation with compiler options, selected from the Options/Project/Compiler menu in Microsoft Developer Studio or specified when you compile from the command line.

All these methods modify the way the compiler compiles your program, but unlike compiler options, compiler directives can be turned on and off and modified as often as you like within your program. If there is a conflict between a compiler option and a compiler directive, the compiler directive takes precedence.

Compiler directives affect everything in a program after the point they appear in the code. Some compiler directives have a matching opposite compiler directive (for instance, **DECLARE** and **NODECLARE**) or can be set to different values (for instance, **!DEC\$ REAL:8** sets the default real type to eight bytes and **!DEC\$ REAL:4** sets it back to the standard four bytes). So, one section of the code can be compiled according to a compiler directive, then the compiler directive can be turned off or modified, and other code in the program can be compiled differently.

The following general compiler directives are available:

- Directives that assure your code follows strict forms, and is therefore transferable to other compilers: **STRICT** and **NOFREEFORM** (and their converses, **NOSTRICT** and **FREEFORM**), and **FIXEDFORMLINESIZE**.
- Directives that specify conditional compilation. If the condition in the compiler directive is true, the statements in the compiler directive block are compiled; otherwise they are not. These compiler directives include **DEFINE** and **UNDEFINE**, **IF** and **IF DEFINED**, **ELSE**, **ELSEIF**, and **ENDIF**.
- Directives that control debugging features: **DECLARE** and **NODECLARE**, and **MESSAGE**.
- Directives that change default data types: **INTEGER** and **REAL**.
- Directives that affect headers in source code listings: **TITLE** and **SUBTITLE**.
- A directive that assigns certain properties to a variable: **ATTRIBUTES**.
- A directive that specifies an alternate external name to be used when referring to external objects such as subroutines and functions: **ALIAS**.
- A directive that specifies an identifier for an object module: **IDENT**.
- A directive that specifies a library search path in the object file: **OBJCOMMENT**.
- A directive that controls whether fields in records and data items in common blocks are naturally aligned or packed on arbitrary byte boundaries: **OPTIONS**.
- A directive that controls the beginning storage address of derived-type components: **PACK**.
- A directive that modifies certain characteristics of a common block: **PSECT**.

This section also discusses:

- **Rules for General Directives**
- **Compiler Directives and Compiler Options**
- **Using the ATTRIBUTES Directive**
- **Using Conditional-Compilation Directives**

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 a compiler option 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.

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 (/ /).

Compiler directives apply to the file they are in until overridden by another compiler directive or until the end of the file. This gives you the flexibility to enable and disable compilation features for various parts of your source code. The **INTEGER**, **REAL**, **STRICT**, and **NOSTRICT** compiler directives can appear only at the top of a program unit, which includes main programs, external subroutines and functions, modules, and block data program units.

Compiler directives apply to any **INCLUDE** files, and an included file can contain its own compiler directives. Compiler directives inside an included file generally apply both to the file and to the rest of the host file. However, if compiler directives within an **INCLUDE** file change the source form or line length (**FREEFORM**, **NOFREEFORM**, or **FIXEDFORMLINESIZE**), those changes are local to the included file and do not affect the host file.

Compiler directives in effect when a module is compiled affect the module. But compiler directives within a program that invokes a module with the **USE** statement have no effect on the module.

You cannot place the **PACK** compiler directive inside control blocks or structures (for example, **IF** blocks and derived-type definition blocks) because **PACK** affects memory locations.

Compiler directive **IF** blocks can include ordinary Visual Fortran statements. You may need to modify those statements when porting your code to other systems.

While the **STRICT** compiler directive is in effect, all compiler directives are interpreted as comments and ignored, producing no warning or error message.

You can add comments at the end of compiler directive lines, but they must begin with an exclamation point (!). For example:

```
!DEC$ DEFINE test                ! Defines the symbol test.
cDEC$ MESSAGE:'Compiling Subroutine ERF' ! Prints a message to the screen.
```

The form **!MS\$** is also allowed as a general directive prefix; for example: **!MS\$DEFINE**.

For more information, see [General Compiler Directives](#).

Compiler Directives and Compiler Options

Some compiler directives and compiler options have the same effect (see the following table). However, compiler directives can be turned on and off throughout a program, while compiler options remain in effect for the whole compilation unless overridden by a compiler directive.

Compiler directive	Equivalent command-line compiler option
DECLARE	/warn:declarations or /4Yd
NODECLARE	/warn:nodeclarations or /4Nd
DEFINE <i>symbol</i>	/define: <i>symbol</i> or /D <i>symbol</i>
FIXEDFORMLINESIZE : <i>option</i>	/extend_source[: <i>option</i>] or /4L <i>option</i>
FREEFORM	/free or /nofixed, or /4Yf
NOFREEFORM	/nofree, /fixed, or /4Nf
INTEGER : <i>option</i>	/integer_size: <i>option</i> or /4I <i>option</i>
OBJCOMMENT	/libdir
PACK : <i>option</i>	/alignment[: <i>option</i>] or /Z <i>option</i>
REAL : <i>option</i>	/real_size: <i>option</i> or /4R <i>option</i>
STRICT	/warn:stderrs with /stand:f90 or /4Ys
NOSTRICT	/4Ns

Note that any of the compiler directive names above can be specified using the prefix **!MS\$**; for example, **!MS\$NOSTRICT** is allowed.

For rules on using compiler directives, see [Rules for General Directives](#).

Using the ATTRIBUTES Directive

Fortran 90 has attributes, such as **ALLOCATABLE**, **INTENT** and **SAVE**, specifying properties that can be assigned to a variable in a separate statement or when the variable is declared. For example:

```
REAL, ALLOCATABLE:: A(:) ! Assigns the attribute in a
                          ! declaration.
```

```
ALLOCATABLE B(:)           ! Assigns the attribute in a
                           ! statement.
```

In addition to using the Fortran 90 attributes, you can use the **ATTRIBUTES** compiler directive to specify properties. Many of these properties can be used to simplify passing variables and procedures between Visual Fortran and another language, such as C. The properties are not assigned the way standard Fortran 90 attributes are; for example:

```
INTERFACE
  SUBROUTINE My_Sub (I)
    !DEC$ ATTRIBUTES C, ALIAS: '_My_Sub' :: My_Sub
    INTEGER I
  END SUBROUTINE My_Sub
END INTERFACE
```

In this example, the **ATTRIBUTES** compiler directive gives the subroutine the C property, which changes the way arguments are passed and referenced, and preserves the mixed-case letters in the name `_My_Sub` with the ALIAS property.

The properties assigned with the **ATTRIBUTES** compiler directive are:

- ALIAS
- C
- DLLEXPORT
- DLLIMPORT
- EXTERN
- REFERENCE
- STDCALL
- VALUE
- VARYING

These properties are described within **ATTRIBUTES** in the *Reference*. For a detailed description of their use in mixed-language programming, see [Programming with Mixed Languages](#).

You can assign more than one property to multiple variables with the same compiler directive. The properties are separated from each other by commas (,), the property are separated from the variables by a double colon (::), and the variables are separated from each other by commas. All properties apply to all the specified variables. For example:

```
!DEC$ ATTRIBUTES REFERENCE, VARYING :: A, B, C
```

In this case, the variables A, B, and C are assigned the REFERENCE and VARYING 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 instance, the following is not allowed:

```
!DEC$ ATTRIBUTES C :: A(10) ! This is illegal.
```

The **ATTRIBUTES** compiler directive can be used for any identifier in a program unit, but it must appear within the program unit that defines it.

For rules on using compiler directives, see [Rules for General Directives](#).

Using Conditional-Compilation Directives

The conditional-compilation compiler directives (**IF** or **IF DEFINED**, **ELSE**, **ELSEIF**, and **ENDIF**) control the compilation of source code. These compiler directives let you do the following:

- Include or omit test code.
- Customize code for specific applications or different operating system platforms by controlling which sections are included.
- Bypass incomplete code during development.

For example:

```
!DEC$ DEFINE a = 4
!DEC$ DEFINE b = 5
...
!DEC$ IF a .LT. b
    WRITE(*,*) "Compiling Section 1"      ! This is compiled if a .LT. b.
!DEC$ ELSE
    WRITE(*,*) "Compiling Section 2"      ! This is compiled if a .GE. b.
!DEC$ ENDIF
```

The **IF** and **ELSEIF** compiler directives include a logical expression. At compilation, this expression is evaluated, and if it is true the code within the **IF** or **ELSEIF** block is compiled. Otherwise, the code is not compiled. For example:

```
!DEC$ IF          logical expression #1
    block of code to be compiled if logical expression #1 is true
!DEC$ ELSEIF     logical expression #2
    block of code to be compiled if logical expression #1 is false and
    logical expression #2 is true
!DEC$ ELSE
    block of code to be compiled if both logical expressions #1 and #2
    are false
!DEC$ ENDIF
```

You can use any Visual Fortran logical or relational operator in the logical expression of the compiler directive, including: **.LT.**, **<**, **.GT.**, **>**, **.EQ.**, **=**, **.LE.**, **<=**, **.GE.**, **>=**, **.NE.**, **/=**, **.EQV.**, **.NEQV.**, **.NOT.**, **.AND.**, **.OR.**, and **.XOR.** Logical expressions within conditional-compilation compiler directives can be as complex as you like, except that the whole directive must fit on one line.

The rules of precedence and logical association in the logical expressions are the same as in standard Visual Fortran logical expressions. However, within conditional-compilation expressions, the logical operators **.EQV.**, **.NEQV.**, **.NOT.**, **.AND.**, **.OR.**, and **.XOR.** can only operate on logical values. For a discussion of logical operators see [Expressions](#).

Symbols in the logical expressions of conditional compiler directives must be defined with the **DEFINE** compiler directive and must be assigned an integer value or no value. These compiler directive symbols are not declared in your Visual Fortran program and are not available to it. So, they can have the same names as your program variables and identifiers and will not conflict with them. Your Visual Fortran program cannot use these symbols, nor can a compiler directive test the value or existence of a variable or constant defined in your program.

The following example shows how to define and use compiler directive defined symbols:

```
CDEC$ DEFINE a = 10
CDEC$ DEFINE b = 8
CDEC$ IF a < b
    WRITE(*,*) "Compiling Section 1"    ! This is compiled if a < b.
CDEC$ ELSEIF a == 1 .AND. DEFINED(b)
    WRITE(*,*) "Compiling Section 2"    ! This is compiled if a equals 1
                                        ! AND b is defined.
CDEC$ ELSE
    WRITE(*,*) "Compiling Section 3"    ! This is compiled if both logical
                                        ! expressions above were false.
CDEC$ ENDIF
```

Another way to perform conditional compilation is to use the **IF DEFINED** compiler directive to test whether or not a symbol has been defined; for example:

```
!DEC$ IF DEFINED (symbol)
```

This statement is **.TRUE.** if *symbol* was defined in a previous **DEFINE** compiler directive or if the /define compiler option was specified on the command line.

The **UNDEFINED** compiler directive cancels the **DEFINE** compiler directive. For example:

```
!DEC$ DEFINE sym          ! sym is now defined.
...
!DEC$ IF DEFINED (sym)
    WRITE(*,*) "About to compile the Hankel code"
...
!DEC$ ELSE
    WRITE(*,*) "About to compile the Bessel code"
...
!DEC$ ENDIF
...
!DEC$ UNDEFINE sym       ! sym is now undefined.
```

You can define as many compiler directive symbols as you want anywhere in a program. If the **DEFINE** compiler directive gives an integer value to a symbol, the value can be assigned to another symbol with another **DEFINE** compiler directive. For example:

```
!DEC$DEFINE firstsym = 100000
!DEC$ DEFINE receiver = firstsym
...
!DEC$ IF receiver .NE. 100000
    WRITE(*,*) "Compile this part of the code"
...
!DEC$ ELSE
    WRITE(*,*) "Compile alternative part of the code"
...
!DEC$ ENDIF
```

Note that the assignment of *receiver* to *firstsym* is not permitted unless *firstsym* has already been given an integer value.

You can use conditional compilation commands within a module, but you cannot use conditional compilation compiler directives to compile modules conditionally within Microsoft Developer Studio, because the proper dependencies cannot be determined in the project structure before compiling. For example, the following is incorrect:

```
CDEC$ IF DEFINED debug
  MODULE mod
    REAL b
  END MODULE
CDEC$ END IF
```

However, the following is correct:

```
MODULE mod
  CDEC$ IF DEFINED debug
    b = 3.0
  CDEC$ END IF
END MODULE
```

For rules on using compiler directives, see [Rules for General Directives](#).

See also the [/define](#) compiler option.

Portability Library

Visual Fortran includes functions and subroutines that ease porting of code from a different platform to a PC, or allow you to write code on a PC that is compatible with other platforms. Frequently used functions are included in a module called DFPORT.

This chapter describes how to use the portability module, and describes routines available in the following categories:

- [Using the Portability Library](#)
- [Routines for Information Retrieval](#)
- [Device and Directory Information Routines](#)
- [Process Control Routines](#)
- [Numeric Routines](#)
- [Input and Output With Portability Routines](#)
- [Date and Time Routines](#)
- [Error Handling Routines](#)
- [Miscellaneous String and Sorting Routines](#)
- [Other Compatibility Routines](#)

Fortran 90 contains intrinsic procedures for many of these functions. New code should use standard Fortran 90 procedures whenever possible.

Using the Portability Library

You can use the portability library in one of two ways:

- Add the statement **USE DFPORT** to your program
- Call portability routines using the correct parameters and return value, which requires that you specify the `/[no]fpcomp` option or otherwise pass the DFPORT.LIB library to the linker during linking.

The portability library is available to your program by default. Using the DFPORT module provides interface blocks and parameter definitions for the routines, as well as compiler verification of calls.

For more information, see: [The DFPORT Module](#)

The DFPORT Module

Some routines in this library can be called with different sets of arguments, and sometimes even as a function instead of a subroutine. In these cases, the arguments and calling mechanism determine the meaning of the routine. The DFPORT module contains generic interface blocks that give procedure definitions for these routines.

If you do not include the statement **USE DFPORT**, you must take care to do the following:

- Declare parameters and return values for all portability routines you use. If you do not do this, the execution stack may not be correctly maintained, causing your program to experience problems. For example, to use the **ACCESS** routine, declare it as "INTEGER(4) ACCESS".

problems. For example, to use the **ACCESS** routine, declare it as "INTEGER(4) ACCESS".

- Be consistent with all calls to portability procedures. When it is possible to call a portability routine in two different ways, only one way can be used in the same source file. For example, if your program calls **DATE** as a subroutine, the same program may not use **DATE** again as a function.
- Specify the `/[no]fpscomp` option or otherwise pass the `DFPORT.LIB` library to the linker during linking.

Routines for Information Retrieval

Functions classified as information retrieval functions return information about system commands, command-line arguments, environment variables, and process or user information.

All portability routines that take path names also accept long file names or UNC (Universal Naming Convention) file names. A forward slash in a path name is treated as a backslash. All path names can contain drive specifications as well as MBCS (multiple-byte character set) characters. For information on MBCS characters, see [Using National Language Support Routines](#).

Portability routine	Description
<u>IARGC</u>	Returns the index of the last command-line argument
<u>GETENV</u>	Searches the environment for a given string, and returns its value if found
<u>GETGID</u>	Returns the group ID of the user
<u>GETLOG</u>	Get user's login name
<u>GETPID</u>	Returns the process ID of the process
<u>GETUID</u>	Returns the user ID of the user of the process
<u>HOSTNAM</u>	Returns the name of the user's host

Group, user, and process ID are INTEGER(4) variables. Login name and host name are character variables. The functions **GETGID** and **GETUID** are provided for portability, but always return 1.

IARGC is best used with **GETARG**. **GETARG**, which returns command line arguments, is available in the standard Visual Fortran library; you do not have to specify `USE DFPORT` in your program unit.

For more information, see [Device and Directory Information](#).

Device and Directory Information Routines

You can retrieve information about devices, directories, and files with the functions listed below. File names can be long file names or UNC file names. A forward slash in a path name is treated as a backslash. All path names can contain drive specifications.

Portability routine	Description
<u>CHDIR</u>	Changes the current working directory
<u>FSTAT</u>	Returns information about a logical file unit
<u>GETCWD</u>	Returns the current working directory path name

<u>RENAME</u>	Renames a file
<u>STAT, LSTAT</u>	Returns information about a named file
<u>UNLINK</u>	Removes a directory entry from the path

Standard Fortran 90 provides the INQUIRE statement, which returns detailed file information either by file name or unit number. Use **INQUIRE** as an equivalent to **FSTAT**, **LSTAT** or **STAT**. **LSTAT** and **STAT** return the same information; **STAT** is the preferred function.

Process Control Routines

Process control functions control the operation of a process or subprocess. You can wait for a subprocess to complete with either **SLEEP** or **ALARM**, monitor its progress and send signals via **KILL**, and stop its execution with **ABORT**.

In spite of its name, **KILL** does not necessarily stop execution of a program. Rather, the routine signaled could include a handler routine that examines the signal and takes appropriate action depending on the code passed.

Portability routine	Description
<u>ABORT</u>	Stops execution of the current process, clears I/O buffers, and writes a string to external unit 0
<u>ALARM</u>	Executes an external subroutine after waiting a specified number of seconds
<u>KILL</u>	Sends a signal code to a process ID
<u>SIGNAL</u>	Changes the action for a signal
<u>SLEEP</u>	Suspends program execution for a specified number of seconds
<u>SYSTEM</u>	Executes a command in a separate shell

Note that when you use **SYSTEM**, commands 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 portability library does not include the **FORK** routine. On U*X systems, **FORK** creates a duplicate image of the parent process. Child and parent processes each have their own copies of resources, and become independent from one another. In Windows NT or Windows 95, you can create a child process (called a thread), but both parent and child processes share the same address space and share system resources. If you need to create another process, use the **CreateProcess** call through the Win32 API.

For information on how to implement threading, see [Creating Multithread Applications](#).

Numeric Routines

Numeric functions are available for calculating Bessel functions, data type conversion, and generating random numbers:

Portability routine	Description
---------------------	-------------

<u>BESJ0, BESJ1, BESJN, BESY0, BESY1, BESYN</u>	Computes the single precision values of Bessel functions of the first and second kind of orders 1, 2, and n , respectively
<u>DBESJ0, DBESJ1, DBESJN, DBESY0, DBESY1, DBESYN</u>	Computes the double-precision values of Bessel functions of the first and second kind of orders 1, 2, and n , respectively
<u>LONG</u>	Converts an INTEGER(2) variable to an INTEGER(4) type
<u>SHORT</u>	Converts an INTEGER(4) variable to an INTEGER(2) type
<u>IRAND, IRANDM</u>	Returns a positive integer in the range 0 through $(2^{**}31)-1$, or $(2^{**}15)-1$ if called without an argument
<u>RAN</u>	Returns random values in the range 0 through 1.0
<u>RAND, DRAND</u>	Returns random values in the range 0 through 1.0
<u>DRANDM, RANDOM</u>	Returns random values in the range 0 through 1.0
<u>SRAND</u>	Seeds the random number generator used with IRAND and RAND .
<u>BIC, BIS, BIT</u>	Perform bit level clear, set, and test for integers

Some of these functions have equivalents in standard Fortran 90. Object conversion can be accomplished by using the INT intrinsic function instead of **LONG** or **SHORT**. The intrinsic subroutines RANDOM_NUMBER and RANDOM_SEED perform the same functions as the random number functions listed in the previous table.

Other bit manipulation functions such as AND, XOR, OR, LSHIFT, and RSHIFT are intrinsic functions. You do not need the DFPORT module to access them. Standard Fortran 90 includes many bit operation procedures; these are listed in the Bit Operation Procedures table in the *Reference*.

Input and Output With Portability Routines

The portability library contains routines that change file properties, read and write characters and buffers, and change the offset position in a file. These input and output routines can be used with standard Fortran input or output statements such as **READ** or **WRITE** on the same files, provided that you take into account the following:

- When used with direct files, after an FSEEK, GETC, or PUTC operation, the record number is the number of the next whole record. Any subsequent normal Fortran I/O to that unit occurs at the next whole record. For example, if you seek to absolute location 1 of a file whose record length is 10, the **NEXTREC=** returned by an inquire would be 2. If you seek to absolute location 10, **NEXTREC=** would still return 2.
- Doing a PUTC (writing to unit 6) clears the input buffer for GETC (reading from unit 5). This is because the I/O library treats 5 and 6 as aliases for the same stream: "con". If either unit is redirected from the command line, this is no longer true.
- On units with **CARRIAGECONTROL='FORTRAN'** (the default), PUTC and FPUTC characters are treated as carriage control characters if they appear in column 1.
- On sequentially formatted units, the C string "\n", which represents the carriage return/line feed escape sequence, is written as **CHAR(13)** (carriage return) and **CHAR(10)** (line feed), instead of just line feed, or **CHAR(10)**. On input, the sequence 13 followed by 10 is returned as just 10. (The length of character string "\n" is 1 character, whose ASCII value, indicated by

ICHAR('n'c), is 10.)

- Reading and writing is in a raw form for direct files. Separators between records can be read and overwritten. Therefore, be careful if you continue using the file as a direct file.

I/O errors arising from the use of these routines result in a Visual Fortran run-time error.

Portability routine	Description
<u>ACCESS</u>	Checks a file for accessibility according to mode
<u>CHMOD</u>	Changes file attributes
<u>FGETC</u>	Reads a character from an external unit
<u>FLUSH</u>	Flushes the buffer for an external unit to its associated file
<u>FPUTC</u>	Writes a character to an external unit
<u>FSEEK</u>	Repositions a file on an external unit
<u>FTELL</u>	Returns the offset, in bytes, from the beginning of the file
<u>GETC</u>	Reads a character from unit 5
<u>PUTC</u>	Writes a character to unit 6

All path names can include drive specifications, forward slashes, or backslashes.

Some portability file I/O routines have equivalents in standard Fortran 90. The **ACCESS** function checks a file specified by name for accessibility according to mode. It tests a file for read, write, or execute permission, as well as checking to see if the file exists. It works on the file attributes as they exist on disk, not as a program's **OPEN** statement specifies them. You can use the **INQUIRE** statement, with the **ACTION=** parameter, to arrive at the same information. (The **ACCESS** function always returns 0 for read permission on FAT files, meaning that all files have read permission.)

Date and Time Routines

Various date and time functions are available to determine system time, or convert it to local time, Greenwich Mean Time, arrays of date and time elements, or an ASCII character string.

The sample output column of the following table assumes the current date to be 2/24/97 7:11 pm Pacific Daylight Time. The third column shows what each routine returns, either when reporting the current time or when that date and time is passed to it in an appropriate argument. Full details of parameters and output are given in the *Reference*.

Portability routine	Description	Sample output
<u>CLOCK</u>	Current time in "hh:mm:ss" format using a 24-hour clock	19:11:00
<u>CTIME</u>	Converts a system time to a 24-character ASCII string	"Wed Feb 24 19:11:00 1997"
<u>DATE</u>	A string representation of the current date	As a subroutine: "24-Feb-97" As a function: "02/24/97"
<u>DTIME</u> ¹	Elapsed CPU time since later of (1) start of program, or (2) most recent call to DTIME	(/0.0, 0.0/ (Actual results depend on the

		program and the system)
<u>ETIME</u> ¹	Elapsed CPU time since the start of program execution	(/0.0, 0.0/) (Actual results depend on the program and the system)
<u>FDATE</u>	The current date and time as an ASCII string	"Wed Feb 24 19:11:00 1997"
<u>GMTIME</u>	Greenwich Mean Time as a 9-element integer array	(/0,12,03,24,2,97,3,55,0/)
<u>IDATE</u>	Current date either as one 3-element array or three scalar parameters (month, day, year)	(1) (/24,2,1997/) (2) month=2, day=24, year=97
<u>ITIME</u>	Current time as a 3-element array (hour, minute, second)	(/7,11,00/)
<u>JDATE</u>	Current date as an 8-character string with the Julian date	"97055 "
<u>LTIME</u>	Local time as a 9-element integer array	(/0,11,7,24,2,97,3,55,0/)
<u>RTC</u>	Number of seconds since 00:00:00 GMT, Jan 1, 1970	762145860
<u>SECNDS</u>	The number of seconds since midnight, less the value of its argument	0.00
<u>TIME</u>	As a subroutine, returns the time formatted as hh:mm:ss As a function, returns the time in seconds since midnight GMT Jan 1, 1970	Subroutine: "07:11:00" Function: 762145860
<u>TIMEF</u>	The number of seconds since the first time this function was called (or zero)	0.0

¹ WNT only

TIME and **DATE** are available as either a function or subroutine. Because of the name duplication, if your programs do not include the **USE DFPORT** statement, each separately compiled program unit can use only one of these versions. For example, if a program calls the subroutine **TIME** once, it cannot also use **TIME** as a function.

Standard Fortran 90 includes new date and time intrinsic subroutines. For more information, see DATE AND TIME in the *Reference*.

Error Handling Routines

The following routines are available for detecting and reporting errors:

Portability routine	Description
<u>IERRNO</u>	Returns the last error code
<u>GERROR</u>	Returns the IERRNO error code as a string variable
<u>PERROR</u>	Sends an error message, preceded by a string, for the last error detected

IERRNO error codes are analogous to *errno* on U*X systems. The DFPORT module provides

parameter definitions for many of U*X's *errno* names, found typically in *errno.h* on U*X systems.

IERRNO is updated only when an error occurs. For example, if a call to the **GETC** function results in an error, but two subsequent calls to **PUTC** succeed, a call to **IERRNO** returns the error for the **GETC** call. Examine **IERRNO** immediately after returning from one of the portability library routines. Other standard Fortran 90 routines might also change the value to an undefined value.

If your application uses multithreading, remember that **IERRNO** is set on a per-thread basis.

Miscellaneous String and Sorting Routines

The following routines perform miscellaneous string and sorting operations:

Portability routine	Description
<u>LNBLNK</u>	Returns the index of the last non-blank character in a string.
<u>QSORT</u>	Sorts a one-dimensional array of a specified number of elements of a named size.
<u>RINDEX</u>	Returns the index of the last occurrence of a substring in a string.

Other Compatibility Routines

If you need to call a routine not listed in the portability library, you might find it in the standard Visual Fortran library. Routines implemented as intrinsic or in the DFLIB module are:

Procedure	Description
<u>AND</u>	Bitwise AND
<u>OR</u>	Bitwise OR
<u>XOR</u>	Bitwise XOR
<u>FREE</u>	Frees dynamic memory
<u>GETARG</u>	Returns command line arguments
<u>MALLOC</u>	Allocates dynamic memory
<u>LSHIFT</u>	Left bitwise shift
<u>RSHIFT</u>	Right bitwise shift
<u>EXIT</u>	Exits program with a return code

Visual Fortran does not support certain other functions, such as:

Routine	Description	Similar Visual Fortran Functionality
CMVGM, CMVGN, CMVGP, CMVGT, CMVGZ	Conditional merge	<u>MERGE</u> intrinsic function
FORK	Creates an identical process	CreateProcess, System
LINK	Creates a hard link between two files	none
SYMLNK	Creates a symbolic link between two files	none

Note: **CreateProcess** is a Win32 API call described in [Creating Multithread Applications](#).

Replace conditional merge routines with the standard Fortran 90 intrinsic **MERGE** routine, using the following arguments:

Routine	Fortran 90 Replacement
CVMGP (<i>tsrc, fsrc, mask</i>)	MERGE (<i>tsrc, fsrc, mask</i> >= 0)
CVMGM (<i>tsrc, fsrc, mask</i>)	MERGE (<i>tsrc, fsrc, mask</i> < 0)
CVMGZ (<i>tsrc, fsrc, mask</i>)	MERGE (<i>tsrc, fsrc, mask</i> = 0)
CVMGN (<i>tsrc, fsrc, mask</i>)	MERGE (<i>tsrc, fsrc, mask</i> /= 0)
CVMGT (<i>tsrc, fsrc, mask</i>)	MERGE (<i>tsrc, fsrc, mask</i> = .TRUE.)

There is no analogy to U*X's file system links or soft links under Windows.

There is also no analogy to the U*X **FORK** routine, since **FORK** creates a duplicate image of the parent process which is independent from the parent process. In Windows NT and Windows 95, both parent and child processes share the same address space and share system resources. For more information on creating child processes, see [Creating Multithread Applications](#).

Using QuickWin

The Visual Fortran QuickWin run-time library helps you turn graphics programs into simple Windows applications. Though the full capability of Windows is not available through QuickWin, QuickWin is simpler to learn and to use. QuickWin applications do support pixel-based graphics, real-coordinate graphics, text windows, character fonts, user-defined menus, mouse events, and editing (select/copy/paste) of text, graphics, or both.

A program using the QuickWin features must explicitly access the QuickWin graphics library routines with the statement **USE DFLIB**, and you must choose your project type as QuickWin Graphics or Standard Graphics.

In Visual Fortran, graphics programs must be either QuickWin, Standard Graphics, Windows, or OpenGL applications. [Standard Graphics applications](#) are a subset of QuickWin that support only one window. You can choose the QuickWin or Standard Graphics application type from the drop-down list of available project types when you create a new project in Developer Studio. Or you can use the [/libs:qwin](#) compiler option for QuickWin or the [/libs:qwins](#) compiler option for Standard Graphics.

Note that QuickWin and Standard Graphics applications cannot be DLLs, and QuickWin and Standard Graphics cannot be linked with run-time routines that are in DLLs. This means that the `/libs=qwin` option and the `/libs=dll` with `/threads` options cannot be used together.

This chapter introduces the major categories of QuickWin library routines. It gives an overview of QuickWin features and their use in creating and displaying graphics, and customizing your QuickWin applications with custom menus and mouse routines. [Drawing Graphics Elements](#), and [Using Fonts from the Graphics Library](#) cover graphics and fonts in more detail.

You can access the QuickWin library from Visual Fortran as well as other languages that support the Fortran calling conventions. The graphics package supports all video modes supported by Windows NT and Windows 95.

Any program using the QuickWin features must include the statement **USE DFLIB** to access the QuickWin graphics library. The DFLIB.MOD module file contains subroutine and function declarations in **INTERFACE** statements, derived-type declarations, symbolic constant declarations, and **EXTERNAL** declarations for each QuickWin routine.

Because **INTERFACE** statements must appear outside the body of a program, the **USE DFLIB** statement must appear outside the body of a program unit. This usually means putting **USE DFLIB** before any other statement.

If a graphics routine does not have a **PROGRAM** statement, then **USE DFLIB** must appear in each subprogram that makes graphics calls, before any declaration statements (such as **IMPLICIT NONE** or **INTEGER**) or any other modules containing declaration statements.

This section includes the following topics:

- [Capabilities of QuickWin](#)
- [Comparing QuickWin with Windows-Based Applications](#)

- [Types of QuickWin Programs](#)
- [The QuickWin User Interface](#)
- [Creating QuickWin Windows](#)
- [Using Graphics and Character-Font Routines](#)
- [Defining Graphics Characteristics](#)
- [Working with Screen Images](#)
- [Enhancing QuickWin Applications](#)
- [Customizing QuickWin Applications](#)
- [QuickWin Programming Precautions](#)
- [Simulating Nonblocking I/O](#)

Capabilities of QuickWin

You can use the QuickWin library to do the following:

- Compile console programs into simple applications for Windows.
- Minimize and maximize QuickWin applications like any Windows-based application.
- Call graphics routines.
- Load and save bitmaps.
- Select, copy and paste text, graphics, or a mix of both.
- Detect and respond to mouse clicks.
- Display graphics output.
- Alter the default application menus or add programmable menus.
- Create custom icons.
- Open multiple child windows.

Comparing QuickWin with Windows-Based Applications

QuickWin does not provide the total capability of Windows. Although you can call many Win32 APIs (Application Programming Interface) from QuickWin and console programs, many other Win32 APIs (such as GDI functions) should be called only from a full Windows application. You need to use Windows-based applications, not QuickWin, if any of the following applies:

- Your application has an OLE (Object Linking and Embedding) container.
- You want direct access to GDI (Graphical Data Interface) functions.
- You want to add your own customized Help information to QuickWin Help.
- You want to create something other than a standard SDI (Single Document Interface) or MDI (Multiple Document Interface) application. (For example, if you want your application to have a dialog such as Windows' Calculator in the client area.)

Types of QuickWin Programs

The QuickWin library creates a Standard Graphics application or a QuickWin Graphics application, depending on the project type you choose. Standard Graphics applications support only one window and do not support programmable menus. QuickWin Graphics applications support multiple windows and user-defined menus. Any Fortran program, whether it contains graphics or not, can be compiled as a QuickWin application. You can use Microsoft Developer Studio to create, debug, and execute Standard Graphics programs and QuickWin Graphics programs.

To build a QuickWin application in Developer Studio, select QuickWin Application from the drop-down list of available project types you see when you create a new project.

To build a Standard Graphics application in Developer Studio, select Standard Graphics Application from the drop-down list of available project types.

To build a QuickWin application from the command line, use the `/libs:qwin` option. For example:

```
DF /libs=qwin qw_app.f90
```

To build a Standard Graphics application from the command line, use the `/libs:qwins` option. For example:

```
DF /libs=qwins stdg_app.f90
```

Complete details on how to build projects in Developer Studio are available in see Working With Projects in the *Developer Studio Environment User's Guide*.

The following sections discuss the two types of QuickWin applications:

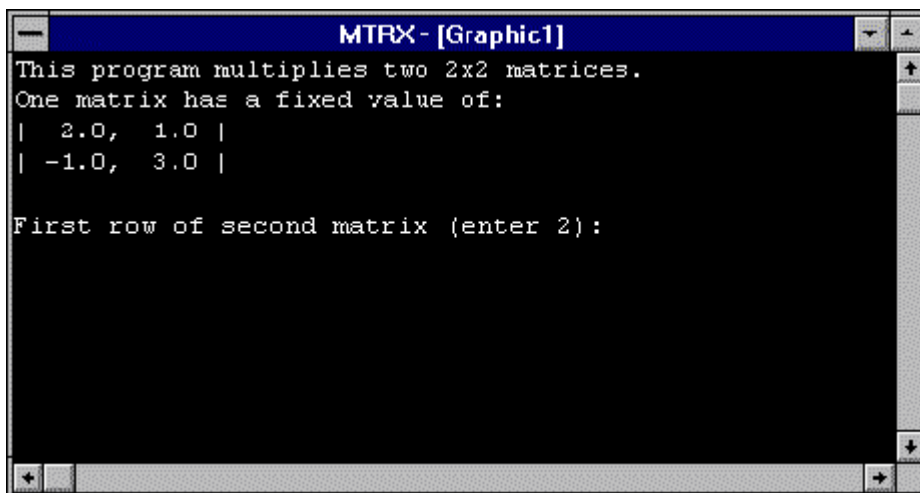
- [Standard Graphics Application](#)
- [QuickWin Graphics Application](#)

Standard Graphics Applications

A standard graphics application is a window with a single maximized application window covering the whole available area. The application window can contain both text and graphics input and output, and it defaults to a scrollable text window. The frame window has only the border, title bar, and scroll bars. Programmable menus and multiple child windows cannot be created in this mode.

The following figure shows a typical Standard Graphics application, which resembles an MS-DOS application running in a window.

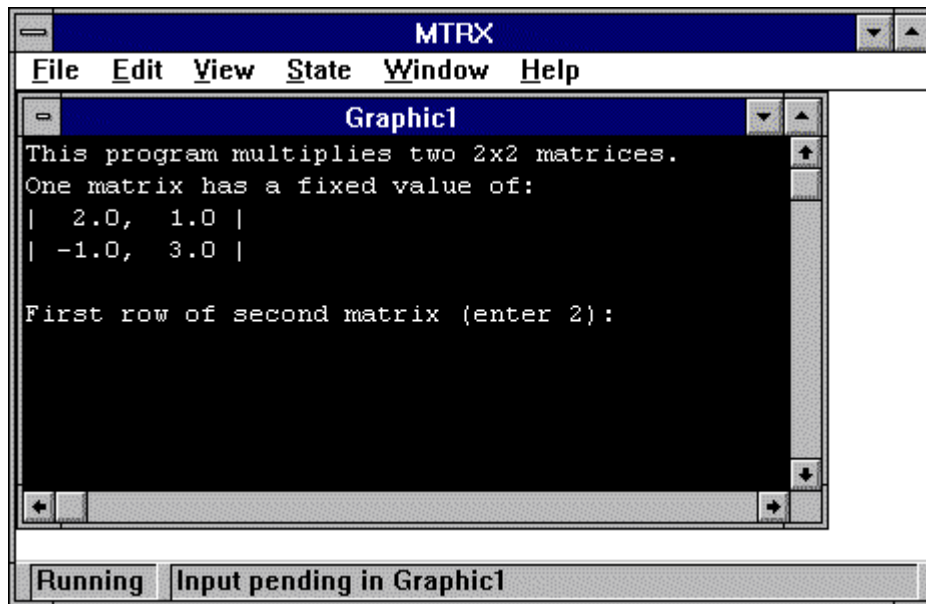
Figure: MTRX.F90 Compiled as a Standard Graphics Application



QuickWin Graphics Applications

The following shows a typical QuickWin Graphics application. The frame window has a border, title bar, scroll bars, and default menu bar. You can modify, add, or delete the default menu items, respond to mouse events, and create multiple child windows within the frame window using QuickWin enhanced features. Routines to create enhanced features are listed in [Enhancing QuickWin Applications](#). Using these routines to customize your QuickWin application is described in [Customizing QuickWin Applications](#).

Figure: MTRX.FOR Compiled as a QuickWin Application



The QuickWin User Interface

All QuickWin applications create an application window; child windows are optional. Standard Graphics applications and QuickWin Graphics applications have these general characteristics:

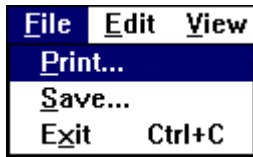
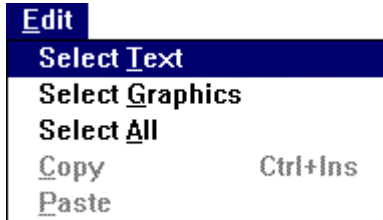
- Window contents can be copied as bitmaps or text to the Clipboard for printing or pasting to other applications. In Standard Graphics applications, the entire window is copied since there is no Edit menu. In QuickWin Graphics applications, any portion of the window can be selected and copied.
- Vertical and horizontal scroll bars appear automatically, if needed.
- The base name of the application's .EXE file appears in the window's title bar.
- Closing the application window terminates the program.

In addition, the QuickWin Graphics application has a status bar and menu bar. The status bar at the bottom of the window reports the current status of the window program (for example, running or input pending).

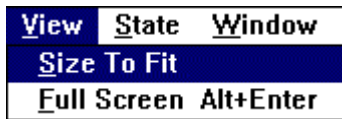
[Default QuickWin Menus](#) shows the default QuickWin menus.

Default QuickWin Menus

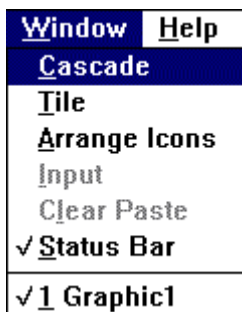
The default MDI (Multiple Document Interface) menu bar has six menus: [File](#), [Edit](#), [View](#), [State](#), [Window](#), and [Help](#).

Figure: File Menu**Figure: Edit Menu**

For instructions on using the Edit options within QuickWin see [Editing Text and Graphics from the QuickWin Edit Menu](#).

Figure: View Menu

The resulting graphics might appear somewhat distorted whenever the logical graphics screen is enlarged or reduced with the Size to Fit and Full Screen commands. While in Full Screen or Size To Fit mode, cursors are not scaled.

Figure: State Menu**Figure: Window Menu****Figure: Help Menu**

For instructions on replacing the About default information within the Help menu with your own text message, see [Defining an About Box](#).

For instructions on how to create custom QuickWin menus, see [Customizing QuickWin Applications](#).

Creating QuickWin Windows

The QuickWin library contains many routines to create and control your QuickWin windows. These routines are discussed in the following topics:

- [Accessing Window Properties](#)
- [Creating Child Windows](#)
- [Giving a Window Focus and Setting the Active Window](#)
- [Keeping Child Windows Open](#)
- [Controlling Size and Position of Windows](#)

Accessing Window Properties

[SETWINDOWCONFIG](#) and [GETWINDOWCONFIG](#) set and get the current window properties. These properties are stored in the `windowconfig` derived type defined in `DFLIB.MOD`, which contains the following parameters:

```

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 scrollable text lines available.
  INTEGER(2) numcolors            ! Number of color indexes.
  INTEGER(4) fontsize             ! Size of default font. Set to
                                ! QWIN$EXTENDFONT when using multibyte
                                ! characters, in which case
                                ! extendfontsize sets the font size.
  CHARACTER(80) title             ! Window title, where title is a C string.
  INTEGER(2) bitsperpixel         ! Number of bits per pixel. This value
                                ! is calculated by the system and is an
                                ! output-only parameter.
                                ! The next three parameters support multibyte
                                ! character sets (such as Japanese)
  CHARACTER(32) extendfontname    ! Any non-proportionally spaced font
                                ! available on the system.
  INTEGER(4) extendfontsize       ! Takes same values as fontsize, but
                                ! used for multiple-byte character sets
                                ! when fontsize set to QWIN$EXTENDFONT.
  INTEGER(4) extendfontattributes ! Font attributes such as bold and
                                ! italic for multibyte character sets.
END TYPE windowconfig

```

If you use **SETWINDOWCONFIG** to set the variables in `windowconfig` to -1, the highest resolution will be set 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 8 by 16. The number of colors depends on the video driver used. In the following example, the number of *x* and *y* pixels is specified and the system calculates the number of rows and columns for the

window:

```

USE DFLIB
TYPE (windowconfig) wc
LOGICAL status
! Set the x & y pixels to 800X600 and font size to 8x12.
wc.numxpixels = 800
wc.numypixels = 600
wc.numtextcols = -1
wc.numtextrows = 302
wc.numcolors = -1
wc.title = " "C
wc.fontsize = #0008000C
status = SETWINDOWCONFIG(wc)

```

In this example, the variable `wc.numtextrows` is set to 302 to allow 300 lines of scollable text ($n-2$ is used).

If the requested configuration cannot be set, **SETWINDOWCONFIG** returns **.FALSE.** and calculates parameter values that will work and best fit the requested configuration. Another call to **SETWINDOWCONFIG** establishes these values:

```
IF(.NOT.status) status = SETWINDOWCONFIG(wc)
```

Creating Child Windows

The **FILE='USER'** option in the **OPEN** statement opens a unit that Visual Fortran treats like any other unit. However, Windows NT and Windows 95 treat the unit as a child window. The child window defaults to a scrollable text window, 30 rows by 80 columns. If the **OPEN** statement contains **FILE=' '**, Visual Fortran displays a Windows File Open dialog box that prompts for a filename to open. You can open up to 40 child windows.

Opening a window displays the frame window, but not the child window. You must call **SETWINDOWCONFIG** or execute an I/O statement or a graphics statement to display the child window. The window receives output by its unit number, as in:

```

OPEN (UNIT= 12, FILE= 'USER', TITLE= 'Product Matrix')
WRITE (12, *) 'Enter matrix type: '

```

Child windows opened with **FILE='USER'** must be opened as sequential-access formatted files (the default). Other file specifications (direct-access, binary, or unformatted) result in run-time errors.

Giving a Window Focus and Setting the Active Window

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 thus does *not* have the *focus*. When a window acquires focus, either by a mouse click, I/O to it, or by a **FOCUSQQ** call, it also becomes the *active* window.

If a window needs to be brought to the foreground, it must be given *focus*. 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.

Under most circumstances, *focus* and *active* should apply to the same window. This is the default behavior of QuickWin and a programmer must consciously override this default.

Certain QuickWin routines (such as GETCHARQQ, PASSDIRKEYSQQ, and SETWINDOWCONFIG) that do not take a unit number as an input argument usually effect the *active* window whether or not it is in *focus*.

If another window is made *active* but is not in *focus*, these routines effect the window *active* at the time of the routine call. This may appear unusual to the user since a **GETCHARQQ** under these circumstances will expect input from a grayed, background window. The user would then have to click on that window before input could be typed to it.

To use these routines (that effect the the *active* window), either do I/O to the unit number of the window you wish to put in *focus* (and also make *active*), or call FOCUSQQ (with a unit number specified). If only one window is open then that window is the one effected. If several windows are opened, then the last one opened is the one effected since that window will get *focus* and *active* as a side effect of being opened.

The **OPEN (IOFOCUS=)** parameter also can determine whether a window receives the focus when a I/O statement is executed on that unit. For example:

```
OPEN (UNIT = 10, FILE = 'USER', IOFOCUS = .TRUE.)
```

IOFOCUS= defaults to **.TRUE.**, except for child windows opened as Unit 0, 5, or 6 and directed at a terminal device, in which case **IOFOCUS=** defaults to **.FALSE.** If **IOFOCUS= .TRUE.**, the child window receives focus prior to each **READ**, **WRITE**, or **PRINT**. Calls to **OUTTEXT** or graphics functions (for example, **OUTGTEXT**, **LINETO**, and **ELLIPSE**) do not cause the focus to shift. If you use **IOFOCUS=** with any unit other than a QuickWin child window, a run-time error occurs.

The focus shifts to a window when it is given the focus with **FOCUSQQ**, when it is selected by a mouse click, or when an I/O operation other than a graphics operation is performed on it, unless the window was opened with **IOFOCUS=.FALSE.** INQFOCUSQQ determines which unit has the focus. For example:

```
USE DFLIB
INTEGER(4) status, focusunit
OPEN(UNIT = 10, FILE = 'USER', TITLE = 'Child Window 1')
OPEN(UNIT = 11, FILE = 'USER', TITLE = 'Child Window 2')
!Give focus to Child Window 2 by writing to it:
WRITE (11, *) 'Giving focus to Child 2.'
! Give focus to Child Window 1 with the FOCUSQQ function:
status = FOCUSQQ(10)
...
! Find out the unit number of the child window that currently has focus:
status = INQFOCUSQQ(focusunit)
```

SETACTIVEQQ makes a child window active without bringing it to the foreground.

GETACTIVEQQ returns the unit number of the currently active child window. GETHWNDQQ converts the unit number into a Windows handle for functions that require it.

Keeping Child Windows Open

A child window remains open as long as its unit is open. The **STATUS=** parameter in the **CLOSE** statement determines whether the child window remains open after the unit has been closed. If you set **STATUS='KEEP'**, the associated window remains open but no further input or output is permitted. Also, the Close command is added to the child window's menu and the word Closed is appended to the window title. The default is **STATUS='DELETE'**, which closes the window.

A window that remains open when you use **STATUS='KEEP'** counts as one of the 40 childwindows available for the QuickWin application.

Controlling Size and Position of Windows

SETWSIZEQQ and **GETWSIZEQQ** set and get the size and position of a window. The positions 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 position and dimensions are returned in the the derived type **qwinfo** defined in **DFLIB.MOD** as follows:

```
TYPE QWINFO
  INTEGER(2) TYPE      ! Type of action performed by SETWSIZEQQ.
  INTEGER(2) X        ! x-coordinate for upper left corner.
  INTEGER(2) Y        ! y-coordinate for upper left corner.
  INTEGER(2) H        ! Window height.
  INTEGER(2) W        ! Window width.
END TYPE QWINFO
```

The options for the element **qwinfo** type are listed under **SETWSIZEQQ** in the *Reference*.

GETWSIZEQQ returns the position and the current or maximum window size of the current frame or child window. To access information about a child window, specify the unit number associated with it. Unit numbers 0, 5, and 6 refer to the default startup window if you have not explicitly opened them with the **OPEN** statement. To access information about the frame window, specify the unit number as the symbolic constant **QWIN\$FRAMEWINDOW**. For example:

```
USE DFLIB
INTEGER(4) status
TYPE (QWINFO) winfo
OPEN (4, FILE='USER')
...
! Get current size of child window associated with unit 4.
status = GETWSIZEQQ(4, QWIN$SIZECURR, winfo)
WRITE (*,*) "Child window size is ", winfo.H, " by ", winfo.W
! Get maximum size of frame window.
status = GETWSIZEQQ(QWIN$FRAMEWINDOW, QWIN$SIZEMAX, winfo)
WRITE (*,*) "Max frame window size is ", winfo.H, " by ", winfo.W
```

SETWSIZEQQ is used to set window position and size. For example:

```
USE DFLIB
INTEGER(4) status
TYPE (QWINFO) winfo
OPEN (4, FILE='USER')
winfo.H = 30
winfo.W = 80
winfo.TYPE = QWIN$SET
status = SETWSIZEQQ(4, winfo)
```


Using Graphics and Character-Font Routines

Graphics routines are functions and subroutines that draw lines, rectangles, ellipses, and similar elements on the screen. Font routines create text in a variety of sizes and styles. The QuickWin graphics library provides routines that:

- Change the window's dimensions.
- Set coordinates.
- Set color palettes.
- Set line styles, fill masks, and other figure attributes.
- Draw graphics elements.
- Display text in several character styles.
- Display text in fonts compatible with Microsoft Windows.
- Store and retrieve screen images.

Defining Graphics Characteristics

The following topics discuss groups of routines that define the way text and graphics are displayed:

- [Selecting Display Options](#)
- [Setting Graphics Coordinates](#)
- [Using Color](#)
- [Setting Figure Properties](#)

Selecting Display Options

The QuickWin run-time library provides a number of routines that you can use to define text and graphics displays. These routines determine the graphics environment characteristics and control the cursor.

SETWINDOWCONFIG is the command you use to configure window properties. You can use DISPLAYCURSOR to control whether the cursor will be displayed. The cursor becomes invisible after a call to SETWINDOWCONFIG. To display the cursor you must explicitly turn on cursor visibility with DISPLAYCURSOR(\$GCURSORON).

SETGTEXTROTATION sets the current orientation for font text output, and GETGTEXTROTATION returns the current setting. The current orientation is used in calls to OUTGTEXT.

For more information on these routines, see the *Reference*.

Setting Graphics Coordinates

The coordinate-setting routines control where graphics can appear on the screen. Visual Fortran graphics routines recognize the following sets of coordinates:

- Fixed *physical coordinates*, which are determined by the hardware and the video mode used
- *Viewport coordinates*, which you can define in the application
- *Window coordinates*, which you can define to simplify scaling of floating-point data values

Unless you change it, the viewport-coordinate system is identical to the physical-coordinate system. The physical origin (0, 0) is always in the upper-left corner of the *display*. For QuickWin, *display* means a child window's client area, not the actual monitor screen (unless you go to Full Screen mode). The x-axis extends in the positive direction left to right, while the y-axis extends in the positive direction top to bottom. The default viewport has the dimensions of the selected mode. In a QuickWin application, you can draw outside of the child window's current client area. If you then make the child window bigger, you will see what was previously outside the frame.

You can also use coordinate routines to convert between physical-, viewport-, and window-coordinate systems. (For more detailed information on coordinate systems, see [Drawing Graphics Elements](#).)

You can set the pixel dimensions of the x- and y-axes with [SETWINDOWCONFIG](#). You can access these values through the *wc.numxpixels* and *wc.numypixels* values returned by [GETWINDOWCONFIG](#). Similarly, [GETWINDOWCONFIG](#) also returns the range of colors available in the current mode through the *wc.numcolors* value.

You can also define the graphics area with [SETCLIPRGN](#) and [SETVIEWPORT](#). Both of these functions define a subset of the available window area for graphics output. [SETCLIPRGN](#) does not change the viewport coordinates, but merely masks part of the screen. [SETVIEWPORT](#) resets the viewport bounds to the limits you give it and sets the origin to the upper-left corner of this region.

The origin of the viewport-coordinate system can be moved to a new position relative to the physical origin with [SETVIEWORG](#). Regardless of the viewport coordinates, however, you can always locate the current graphics output position with [GETCURRENTPOSITION](#) and [GETCURRENTPOSITION_W](#). (For more detailed information on viewports and clipping regions, see [Drawing Graphics Elements](#).)

Using the window-coordinate system, you can easily scale any set of data to fit on the screen. You define any range of coordinates (such as 0 to 5000) that works well for your data as the range for the window-coordinate axes. By telling the program that you want the window-coordinate system to fit in a particular area on the screen (map to a particular set of viewport coordinates), you can scale a chart or drawing to any size you want. [SETWINDOW](#) defines a window-coordinate system bounded by the specified values. See SINE.F90 in the \DF\SAMPLES\TUTORIAL subdirectory for an example of this technique.

[GETPHYSCOORD](#) converts viewport coordinates to physical coordinates, and [GETVIEWCOORD](#) translates from physical coordinates to viewport coordinates. Similarly, [GETVIEWCOORD_W](#) converts window coordinates to viewport coordinates, and [GETWINDOWCOORD](#) converts viewport coordinates to window coordinates.

For more information on these routines, see the *Reference*.

Using Color

If you have a VGA machine, you are restricted to displaying at most 256 colors at a time. These 256 colors are held in a palette. You can choose the palette colors from a range of 262,144 colors (256K), but only 256 at a time. The palette routines [REMAPPALLETTERGB](#) and [REMAPALLPALETTERGB](#) assign Red-Green-Blue (RGB) colors to palette indexes.

Functions and subroutines that use color indexes create graphic outputs that depend on the mapping between palette indexes and RGB colors. **REMAPPALLETTERGB** remaps one color index to an RGB color, and **REMAPALLPALETTERGB** remaps the entire palette, up to 236 colors, (20 colors are reserved by the system). You cannot remap the palette on machines capable of displaying 20 colors or fewer.

SVGA and true color video adapters are capable of displaying 262,144 (256K) colors and 16.7 million colors respectively. If you use a palette, you are restricted to the colors available in the palette.

To access the entire set of available colors, not just the 256 or fewer colors in the palette, you should use functions that specify a color value directly. These functions end in **RGB** and use Red-Green-Blue color values, not indexes to a palette. For example, SETCOLORRGB, SETTEXTCOLORRGB, and SETPIXELRGB specify a direct color value, while SETCOLOR, SETTEXTCOLOR, and SETPIXEL each specify a palette color index. If you are displaying more than 256 colors simultaneously, you need to use the RGB direct color value functions exclusively.

For more information on setting colors, see Adding Color in Drawing Graphics Elements.

Setting Figure Properties

The output routines that draw arcs, ellipses, and other primitive figures do not specify color or line-style information. Instead, they rely on properties set independently by other routines.

GETCOLORRGB (or GETCOLOR) and SETCOLORRGB (or SETCOLOR) obtain or set the current color value (or color index), which FLOODFILLRGB (or FLOODFILL), OUTGTEXT, and the shape-drawing routines all use. Similarly, GETBKCOLORRGB (or GETBKCOLOR) and SETBKCOLORRGB (or SETBKCOLOR) retrieve or set the current background color.

GETFILLMASK and SETFILLMASK return or set the current fill mask. The mask is an 8-by-8-bit array with each bit representing a pixel. If a bit is 0, the pixel in memory is left untouched: the mask is transparent to that pixel. If a bit is 1, the pixel is assigned the current color value. The array acts as a template that repeats over the entire fill area. It "masks" the background with a pattern of pixels drawn in the current color, creating a large number of fill patterns. These routines are particularly useful for shading.

GETWRITEMODE and SETWRITEMODE return or set the current *logical write mode* used when drawing lines. The logical write mode, which can be set to \$GAND, \$GOR, \$GPRESET, \$GPSET, or \$GXOR, determines the interaction between the new drawing and the existing screen and current graphics color. The logical write mode affects the **LINETO**, **RECTANGLE**, and **POLYGON** routines.

GETLINESTYLE and SETLINESTYLE retrieve and set the current line style. The line style is determined by a 16-bit-long mask that determines which of the five available styles is chosen. You can use these two routines to create a wide variety of dashed lines that affect the **LINETO**, **RECTANGLE**, and **POLYGON** routines.

For more information on these routines, see their description in the *Reference*.

Displaying Graphics Output

The run-time graphics library routines can draw geometric features, display text, display font-based characters, and transfer images between memory and the screen. These capabilities are discussed in the following topics:

- [Drawing Graphics](#)
- [Displaying Character-Based Text](#)
- [Displaying Font-Based Characters](#)

Drawing Graphics

If you want anything other than the default line style (solid), mask (no mask), background color (black), or foreground color (white), you must call the appropriate routine before calling the drawing routine. Subsequent output routines employ the same attributes until you change them or open a new child window.

The following is a list of routines that ask about the current graphics settings, set new graphics settings, and draw graphics:

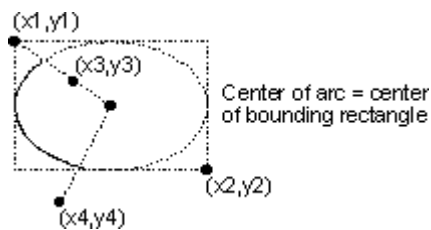
Routine	Use
ARC, ARC_W	Draws an arc
CLEARSCREEN	Clears the screen, viewport, or text window
ELLIPSE, ELLIPSE_W	Draws an ellipse or circle
FLOODFILL, FLOODFILL_W	Fills an enclosed area of the screen with the current color index using the current fill mask
FLOODFILLRGB, FLOODFILLRGB_W	Fills an enclosed area of the screen with the current RGB color using the current fill mask
GETARCINFO	Determines the endpoints of the most recently drawn arc or pie
GETCURRENTPOSITION, GETCURRENTPOSITION_W	Returns the coordinates of the current graphics-output position
GETPIXEL, GETPIXEL_W	Returns a pixel's color index
GETPIXELRGB, GETPIXELRGB_W	Returns a pixel's Red-Green-Blue color value
GETPIXELS	Gets the color indices of multiple pixels
GETPIXELSRGB	Gets the Red-Green-Blue color values of multiple pixels
GRSTATUS	Returns the status (success or failure) of the most recently called graphics routine
INTEGERTORGB	Convert a true color value into its red, green, and blue components
LINETO, LINETO_W	Draws a line from the current graphics-output position to a specified point
MOVETO, MOVETO_W	Moves the current graphics-output position to a specified point
PIE, PIE_W	Draws a pie-slice-shaped figure
POLYGON, POLYGON_W	Draws a polygon

<u>RECTANGLE, RECTANGLE_W</u>	Draws a rectangle
<u>RGBTOINTEGER</u>	Convert a trio of red, green, and blue values to a true color value for use with RGB functions and subroutines
<u>SETPIXEL, SETPIXEL_W</u>	Sets a pixel at a specified location to a color index
<u>SETPIXELRGB, SETPIXELRGB_W</u>	Sets a pixel at a specified location to a Red-Green-Blue color value
<u>SETPIXELS</u>	Set the color indices of multiple pixels
<u>SETPIXELSRGB</u>	Set the Red-Green-Blue color value of multiple pixels

Most of these routines have multiple forms. Routine names that end with _W use the window-coordinate system and REAL(8) argument values. Routines without this suffix use the viewport-coordinate system and INTEGER(2) argument values.

Curved figures, such as arcs and ellipses, are centered within a *bounding rectangle*, which is specified by the upper-left and lower-right corners of the rectangle. The center of the rectangle becomes the center for the figure, and the rectangle's borders determine the size of the figure. In the following figure, the points $(x1, y1)$ and $(x2, y2)$ define the bounding rectangle.

Figure: Bounding Rectangle



For more information on these routines, see the *Reference*.

Displaying Character-Based Text

The routines in the following table ask about screen attributes that affect text display, prepare the screen for text and send text to the screen. To print text in specialized fonts, see [Displaying Font-Based Characters](#) and [Using Fonts from the Graphics Library](#).

In addition to these general text routines, you can customize the text in your menus with [MODIFYMENUSTRINGQQ](#). You can also customize any other string that QuickWin produces, including status bar messages, the state message (for example, "Paused" or "Running"), and dialog box messages, with [SETMESSAGEQQ](#). Use of these customization routines is described in [Customizing QuickWin Applications](#).

The following routines recognize text-window boundaries:

Routine	Use
<u>CLEARSCREEN</u>	Clears the screen, viewport, or text window
<u>DISPLAYCURSOR</u>	Sets the cursor on or off
<u>GETBKCOLOR</u>	Returns the current background color index
<u>GETBKCOLORRGB</u>	Returns the current background Red-Green-Blue color value
<u>GETTEXTCOLOR</u>	Returns the current text color index

<u>GETTEXTCOLORRGB</u>	Returns the current text Red-Green-Blue color value
<u>GETTEXTPOSITION</u>	Returns the current text-output position
<u>GETTEXTWINDOW</u>	Returns the boundaries of the current text window
<u>OUTTEXT</u>	Sends text to the screen at the current position
<u>SCROLLTEXTWINDOW</u>	Scrolls the contents of a text window
<u>SETBKCOLOR</u>	Sets the current background color index
<u>SETBKCOLORRGB</u>	Sets the current background Red-Green-Blue color value
<u>SETTEXTCOLOR</u>	Sets the current text color to a new color index
<u>SETTEXTCOLORRGB</u>	Sets the current text color to a new Red-Green-Blue color value
<u>SETTEXTPOSITION</u>	Changes the current text position
<u>SETTEXTWINDOW</u>	Sets the current text-display window
<u>WRAPON</u>	Turns line wrapping on or off

These routines do not provide text-formatting capabilities. If you want to print integer or floating-point values, you must convert the values into a string (using an internal WRITE statement) before calling these routines. The text routines specify all screen positions in character-row and column coordinates.

SETTEXTWINDOW is the text equivalent of the SETVIEWPORT graphics routine, except that it restricts only the display area for text printed with **OUTTEXT**, **PRINT**, and **WRITE**.

GETTEXTWINDOW returns the boundaries of the current text window set by **SETTEXTWINDOW**. **SCROLLTEXTWINDOW** scrolls the contents of a text window. **OUTTEXT**, **PRINT**, and **WRITE** display text strings written to the current text window.

Warning: The **WRITE** statement sends its carriage return (CR) and line feed (LF) to the screen at the beginning of the first I/O statement following the **WRITE** statement. This can cause unpredictable text positioning if you mix the graphics routines **SETTEXTPOSITION** and **OUTTEXT** with the **WRITE** statement. To minimize this effect, use the backslash (\) or dollar sign (\$) format descriptor (to suppress CR-LF) in the associated FORMAT statement.

For more information on these routines, see the *Reference*.

Displaying Font-Based Characters

Because the Visual Fortran Graphics Library provides a variety of fonts, you must indicate which font to use when displaying font-based characters. After you select a font, you can make inquiries about the width of a string printed in that font or about font characteristics. The following functions control the display of font-based characters:

Routine	Use
<u>GETFONTINFO</u>	Returns the current font characteristics
<u>GETGTEXTTEXTENT</u>	Determines the width of specified text in the current font
<u>GETGTEXTROTATION</u>	Gets the current orientation for font text output in 0.1° increments
<u>INITIALIZEFONTS</u>	Initializes the font library
<u>OUTGTEXT</u>	Sends text in the current font to the screen at the current graphics output position
<u>SETFONT</u>	Finds a single font that matches a specified set of characteristics and

	makes it the current font used by OUTGTEXT
SETGTEXTROTATION	Sets the current orientation for font text output in 0.1° increments

Characters may be drawn ("mapped") in one of two ways: as bitmapped letters (a "picture" of the letter) or as TrueType characters. See [Using Fonts from the Graphics Library](#), for detailed explanations and examples of how to use the font routines from the QuickWin Library.

For more information on these routines, see the *Reference*.

Working With Screen Images

The routines described in the following sections offer the following ways to store and retrieve images:

- [Transfer images between memory buffers and the screen](#)

Transferring images from buffers is a quick and flexible way to move things around the screen. Memory images can interact with the current screen image; for example, you can perform a logical **AND** of a memory image and the current screen or superimpose a negative of the memory image on the screen.

- [Transfer images between the screen and Windows bitmap files](#)

Transferring images from files gives access to images created by other programs, and saves graphs and images for later use. However, images loaded from bitmap files overwrite the portion of the screen they are pasted into and retain the attributes they were created with, such as the color palette, rather than accepting current attributes.

- [Transfer images between the screen and the Clipboard from the QuickWin Edit menu](#)

Editing screen images from the QuickWin Edit menu is a quick and easy way to move and modify images interactively on the screen, retaining the current screen attributes, and also provides temporary storage (the Clipboard) for transferring images among applications.

These routines allow you to cut, paste, and move images around the screen.

Transferring Images in Memory

The **GETIMAGE** and **PUTIMAGE** routines transfer images between memory and the screen and give you options that control the way the image and screen interact. When you hold an image in memory, the application allocates a memory buffer for the image. The **IMAGESIZE** routines calculate the size of the buffer needed to store a given image.

Routines that end with **_W** use window coordinates; the other functions use viewport coordinates.

Routine	Use
GETIMAGE , GETIMAGE_W	Stores a screen image in memory
IMAGESIZE , IMAGESIZE_W	Returns image size in bytes
PUTIMAGE , PUTIMAGE_W	Retrieves an image from memory and displays it

For more information on these routines, see the *Reference*.

Loading and Saving Images to Files

The **LOADIMAGE** and **SAVEIMAGE** routines transfer images between the screen and Windows bitmap files:

Routine	Use
<u>LOADIMAGE</u> , <u>LOADIMAGE W</u>	Reads a Windows bitmap file (.BMP) from disk and displays it as specified coordinates
<u>SAVEIMAGE</u> , <u>SAVEIMAGE W</u>	Captures a screen image from the specified portion of the screen and saves it as a Windows bitmap file

You can use a Windows format bitmap file created with a graphics program as a backdrop for graphics that you draw with the Visual Fortran graphics functions and subroutines.

For more information on these routines, see the *Reference*.

Editing Text and Graphics from the QuickWin Edit Menu

From the QuickWin Edit menu you can choose the Select Text, Select Graphics, or Select All options. You can then outline your selection with the mouse or the keyboard arrow keys. When you use the Select Text option, your selection is highlighted. When you use the Select Graphics or Select All option, your selection is marked with a box whose dimensions you control.

Once you have selected a portion of the screen, you can delete it with the DEL key and/or copy it onto the Clipboard by using the Edit/Copy option or by using the CTRL+INS key combination. If the screen area you have selected contains only text, it is copied onto the Clipboard as text. If the selected screen area contains graphics, or a mix of text and graphics, it is copied onto the Clipboard as a bitmap.

The Edit menu's Paste option will only paste text. Bitmaps can be pasted into other Windows applications from the Clipboard (with the CTRL+V or SHIFT+INS key combinations).

Remember the following when selecting portions of the screen:

- If you have chosen the Select All option from the Edit menu, the whole screen is selected and you cannot then select a portion of the screen.
- Text selections are not bounded by the current text window set with **SETTEXTWINDOW**.
- When text is copied to the Clipboard, trailing blanks in a line are removed.
- Text that is written to a window can be overdrawn by graphics. In this case, the text is still present in the screen text buffer, though not visible on the screen. When you select a portion of the screen to copy, you can select text that is actually present but not visible, and that text will be copied onto the Clipboard.
- When you chose Select Text or Select Graphics from the Edit menu, the application is paused, a caret (^) appears at the top left corner of the currently active window, all user-defined

callbacks are disabled, and the window title changes to "Mark Text - *windowname* " or "Mark Graphics - *windowname* ", where *windowname* is the name of the currently active window.

As soon as you begin selection (by pressing an arrow key or a mouse button), the Window title changes to "Select Text - *windowname* " or "Select Graphics - *windowname* " and selection begins at that point. If you do not want selection to begin in the upper-left corner, your first action when "Mark Text" or "Mark Graphics" appears in the title is to use the mouse to place the cursor at the position where selection is to be begin.

Enhancing QuickWin Applications

In addition to the basic QuickWin features, you can optionally customize and enhance your QuickWin applications with the features described in the following table. The use of these features to create customized menus, respond to mouse events, and add custom icons is described in the section Customizing QuickWin Applications.

Category	QuickWin Function	Description
Initial settings	<u>INITIALSETTINGS</u>	Controls initial menu settings and/or initial frame window
Display/add box	<u>MESSAGEBOXQQ</u>	Displays a message box
	<u>ABOUTBOXQQ</u>	Adds an About Box with customized text
Menu items	<u>CLICKMENUQQ</u>	Simulates the effect of clicking or selecting a menu item
	<u>APPENDMENUQQ</u>	Appends a menu item
	<u>DELETEMENUQQ</u>	Deletes a menu item
	<u>INSERTMENUQQ</u>	Inserts a menu item
	<u>MODIFYMENUFLAGSQQ</u>	Modifies a menu item's state
	<u>MODIFYMENUROUTINEQQ</u>	Modifies a menu item's callback routine
	<u>MODIFYMENUSTRINGQQ</u>	Changes a menu item's text string
	<u>SETWINDOWMENUQQ</u>	Sets the menu to which a list of current child window names are appended
Directional keys	<u>PASSDIRKEYSQQ</u>	Enables (or disables) use of the arrow directional keys and page keys as input (see <u>DIRKEYS.F90</u> in the <u>\DF\SAMPLES\ADVANCED\DIRKEYS</u> subdirectory)
QuickWin messages	<u>SETMESSAGEQQ</u>	Changes any QuickWin message, including status bar messages, state messages and dialog box messages
Mouse actions	<u>REGISTERMOUSEEVENT</u>	Registers the application defined routines to be called on mouse events
	<u>UNREGISTERMOUSEEVENT</u>	Removes the routine registered by REGISTERMOUSEEVENT
	<u>WAITONMOUSEEVENT</u>	Blocks return until a mouse event occurs

Customizing QuickWin Applications

The QuickWin library is a set of routines you can use to create graphics programs or simple

applications for Windows. For a general overview of QuickWin and a description of how to create and size child windows, see the beginning of this section. For information on how to compile and link QuickWin applications, see [Building Programs and Libraries](#).

The following topics describe how to customize and fine-tune your QuickWin applications:

- [Program Control of Menus](#)
- [Changing Status Bar and State Messages](#)
- [Displaying Message Boxes](#)
- [Defining an About Box](#)
- [Using Custom Icons](#)
- [Using a Mouse](#)

Program Control of Menus

You do not have to use the default QuickWin menus. You can eliminate and alter menus, menu item lists, menu titles or item titles. The QuickWin functions that control menus are described in the following sections:

- [Controlling the Initial Menu and Frame Window](#)
- [Deleting, Inserting, and Appending Menu Items](#)
- [Modifying Menu Items](#)
- [Creating a Menu List of Available Child Windows](#)
- [Simulating Menu Selections](#)

Controlling the Initial Menu and Frame Window

You can change the initial appearance of an application's default frame window and menus by defining an **INITIALSETTINGS** function. If no user-defined **INITIALSETTINGS** function is supplied, QuickWin calls a predefined **INITIALSETTINGS** routine to control the default frame window and menu appearance. Your application does not need to call **INITIALSETTINGS**. If you supply the function in your project, QuickWin calls it automatically.

If you supply it, **INITIALSETTINGS** can call QuickWin functions that set the initial menus and the size and position of the frame window. Besides the menu functions, **SETWSIZEQQ** can be called from your **INITIALSETTINGS** function to adjust the frame window size and position before the window is first drawn.

The following is a sample of **INITIALSETTINGS**:

```

LOGICAL(4) FUNCTION INITIALSETTINGS( )
  USE DFLIB
  LOGICAL(4) result
  TYPE (qwinfo) qwi
! Set window frame size.
  qwi.x = 0
  qwi.y = 0
  qwi.w = 400
  qwi.h = 400
  qwi.type = QWIN$SET
  i = SetWSizeQQ( QWIN$FRAMEWINDOW, qwi )
! Create first menu called Games.
  result = APPENDMENUQQ(1, $MENUENABLED, '&Games'C, NUL )

```

```

! Add item called TicTacToe.
    result = APPENDMENUQQ(1, $MENUENABLED, '&TicTacToe'C, WINPRINT)
! Draw a separator bar.
    result = APPENDMENUQQ(1, $MENSEPARATOR, ''C, NUL )
! Add item called Exit.
    result = APPENDMENUQQ(1, $MENUENABLED, 'E&xit'C, WINEXIT )
! Add second menu called Help.
    result = APPENDMENUQQ(2, $MENUENABLED, '&Help'C, NUL )
    result = APPENDMENUQQ(2, $MENUENABLED, '&QuickWin Help'C, WININDEX)
    INITIALSETTINGS= .true.
END FUNCTION INITIALSETTINGS

```

This is an example of the interface for **INITIALSETTINGS**:

```

PROGRAM MENUS
  USE DFLIB
  LOGICAL(4) res
  INTERFACE
    LOGICAL(4) FUNCTION INITIALSETTINGS
  END FUNCTION
END INTERFACE
OPEN (10, FILE="User")
WRITE(10, *) "Hello, child window"
END

```

QuickWin executes your **INITIALSETTINGS** function during initialization, before creating the frame window. When your function is done, control returns to QuickWin and it does the remaining initialization. The control then passes to the Visual Fortran application.

Your function should return **.TRUE.** if it succeeds, and **.FALSE.** otherwise. The QuickWin default function returns a value of **.TRUE.** only.

Note that default menus are created after **INITIALSETTINGS** has been called, and only if you do not create your own menus. Therefore, using **DELETEMENUQQ**, **INSERTMENUQQ**, **APPENDMENUQQ**, and the other menu configuration QuickWin functions while in **INITIALSETTINGS** affects your custom menus, not the default QuickWin menus.

Deleting, Inserting, and Appending Menu Items

Menus are defined from left to right, starting with 1 at the far left. Menu items are defined from top to bottom, starting with 0 at the top (the menu title itself). Within **INITIALSETTINGS**, if you supply it, you can delete, insert, and append menu items in custom menus. Outside **INITIALSETTINGS**, you can alter the default QuickWin menus as well as custom menus at any point in your application. (Default QuickWin menus are not created until after **INITIALSETTINGS** has run and only if you do not create custom menus.)

To delete a menu item, specify the menu number and item number in **DELETEMENUQQ**. To delete an entire menu, delete item 0 of that menu. For example:

```

USE DFLIB
LOGICAL status
status = DELETEMENUQQ(1, 2) ! Delete the second menu item from
                           ! menu 1 (the default FILE menu).
status = DELETEMENUQQ(5, 0) ! Delete menu 5 (the default Windows
                           ! menu).

```

INSERTMENUQQ inserts a menu item or menu and registers its callback routine. QuickWin

supplies several standard callback routines such as WINEXIT to terminate a program, WININDEX to list QuickWin Help, and WINCOPY which copies the contents of the current window to the Clipboard. A list of available callbacks is given in the *Reference* for **INSERTMENUQQ** and **APPENDMENUQQ**. Often, you will supply your own callback routines to perform a particular action when a user selects something from one of your menus.

In general, you should not assign the same callback routine to more than one menu item because a menu item's state might not be properly updated when you change it (put a check mark next to it, gray it out, or disable, or enable it). You cannot insert a menu item or menu beyond the existing number; for example, inserting item 7 when 5 and 6 have not been defined yet. To insert an entire menu, specify menu item 0. The new menu can take any position among or immediately after existing menus.

If you specify a menu position occupied by an existing menu, the existing menu and any menus to the right of the one you add are shifted right and their menu numbers are incremented.

For example, the following code inserts a fifth menu item called `Position` into menu 5 (the default Windows menu):

```
USE DFLIB
LOGICAL(4) status
status = INSERTMENUQQ (5, 5, $MENUCHECKED, 'Position'C, WINPRINT)
```

The next code inserts a new menu called `My List` into menu position 3. The menu currently in position 3 and any menus to the right (the default menus View, State, Windows, and Help) are shifted right one position:

```
USE DFLIB
LOGICAL(4) status
status = INSERTMENUQQ(3,0, $MENUENABLED, 'My List'C, WINSTATE)
```

You can append a menu item with **APPENDMENUQQ**. The item is added to the bottom of the menu list. If there is no item yet for the menu, your appended item is treated as the top-level menu item, and the string you assign to it appears on the menu bar. The following code appends the menu item called `Cascade Windows` to the first menu (the default File menu):

```
USE DFLIB
LOGICAL(4) status
status = APPENDMENUQQ(1, $MENUCHECKED, 'Cascade Windows'C, &
& WINCASCADE)
```

The `$MENUCHECKED` flag in the example puts a check mark next to the menu item. To remove the check mark, you can set the flag to `$MENUUNCHECKED` in the **MODIFYMENUFLAGSQQ** function. Some predefined routines (such as WINSTATUS) take care of updating their own check marks. However, if the routine is registered to more than one menu item, the check marks might not be properly updated. See **APPENDMENUQQ** or **INSERTMENUQQ** in the *Reference* for the list of callback routines and other flags.

Modifying Menu Items

MODIFYMENUSTRINGQQ can modify the string identifier of a menu item, **MODIFYMENUROUTINEQQ** can modify the callback routine called when the item is selected, and **MODIFYMENUFLAGSQQ** can modify a menu item's state (such as enabled, grayed out,

checked, and so on).

The following example code uses **MODIFYMENUSTRINGQQ** to modify the menu string for the fourth item in the first menu (the File menu by default) to `Tile Windows`, it uses **MODIFYMENUROUTINEQQ** to change the callback routine called if the item is selected to `WINTILE`, and uses **MODIFYMENUFLAGSQQ** to put a check mark next to the menu item:

```
status = MODIFYMENUSTRINGQQ( 1, 4, 'Tile Windows'C)
status = MODIFYMENUROUTINEQQ( 1, 4, WINTILE)
status = MODIFYMENUFLAGSQQ( 1, 4, $MENCHECKED)
```

Creating a Menu List of Available Child Windows

By default, the Windows menu contains a list of all open child windows in your QuickWin applications. **SETWINDOWMENUQQ** changes the menu which lists the currently open child windows to the menu you specify. The list of child window names is appended to the end of the menu you choose and deleted from any other menu that previously contained it. For example:

```
USE DFLIB
LOGICAL(4) status
...
! Append list of open child windows to menu 1 (the default File menu)
status = SETWINDOWMENUQQ(1)
```

Simulating Menu Selections

CLICKMENUQQ simulates the effect of clicking or selecting a menu command from the Window menu. The QuickWin application behaves as though the user had clicked or selected the command. The following code fragment simulates the effect of selecting the `Tile` item from the Window menu:

```
USE DFLIB
INTEGER(4) status
status = CLICKMENUQQ(QWIN$TILE)
```

Only items from the Window menu can be specified in **CLICKMENUQQ**.

Changing Status Bar and State Messages

Any string QuickWin produces can be changed by calling **SETMESSAGEQQ** with the appropriate message ID. Unlike other QuickWin message functions, **SETMESSAGEQQ** uses regular Fortran strings, not null-terminated C strings. For example, to change the `PAUSED` state message to `I am waiting`:

```
USE DFLIB
CALL SETMESSAGEQQ('I am waiting', QWIN$MSG_PAUSED)
```

This function is useful for localizing your QuickWin applications for countries with different native languages. A list of message IDs is given in **SETMESSAGEQQ** in the *Reference*.

Displaying Message Boxes

MESSAGEBOXQQ causes your program to display a message box. You can specify the message the box displays and the caption that appears in the title bar. Both strings must be null-terminated C

strings. You can also specify the type of message box. Box types are symbolic constants defined in `DFLIB.MOD`, and can be combined by means of the `IOR` intrinsic function or the `.OR.` operator. The available box types are listed under `MESSAGEBOXQQ` in the *Reference*. For example:

```
USE DFLIB
INTEGER(4) response
response = MESSAGEBOXQQ('Retry or Cancel?'C, 'Smith Chart &
& Simulator'C, MB$RETRYCANCELQWIN .OR. MB$DEFBUTTON2)
```

Defining an About Box

The `ABOUTBOXQQ` function specifies the message displayed in the message box that appears when the user selects the About command from a QuickWin application's Help menu. (If your program does not call `ABOUTBOXQQ`, the QuickWin run-time library supplies a default string.) The message string must be a null-terminated C string. For example:

```
USE DFLIB
INTEGER(4) status
status = ABOUTBOXQQ ('Sound Speed Profile Tables Version 1.0'C)
```

Using Custom Icons

The QuickWin run-time library provides default icons that appear when the user minimizes the application's frame window or its child windows. You can add custom-made icons to your executable files, and Windows will display them instead of the default icons.

► To add a custom child window icon to your QuickWin program:

1. Select Resource from the Insert menu in Developer Studio. Select Icon from the list that appears. The screen will become an icon drawing tool.
2. Draw the icon. (For more information about using the Graphics Editor in Microsoft Developer Studio, see "Resource Editors, Graphics Editor" in the [Developer Studio Environment User's Guide](#).)

-or-

If your icon already exists (for example, as a bitmap) and you want to import it, not draw it, select Resource from the Insert menu, then select Import from the buttons in the Resource dialog. You will be prompted for the file containing your icon.

3. Name the icon. The frame window's icon must have the name "frameicon," and the child window's icon must have the name "childicon." These names must be entered as strings into the Icon Properties dialog box.

To display the Icon Properties dialog box, double-click in the icon editor area outside the icon's grid or press `ALT+ENTER`.

In the ID field on the General tab of Icon Properties dialog box, type over the default icon name with "frameicon" or "childicon." You must add the quotation marks to the text you type in order to make the name be interpreted as a string.

Your icon will be saved in a file with the extension `.ICO`.

4. Create a script file to hold your icons. Select File/Save As. You will be prompted for the name of the script file that will contain your icons. Name the script file. It must end with the extension .RC; for example, myicons.rc. Using this method, the icons and their string values will be automatically saved in the script file. (Alternatively, you can create a script file with any editor and add the icon names and their string values by hand.)
5. Add the script file to the project that contains your QuickWin application. Select Build and the script file will be built into the application's executable. (The compiled script file will have the extension .RES.)

When you run your application, the icon you created will take the place of the default child or frame icon. Your custom icon appears in the upper-left corner of the window frame. When you minimize the window, the icon appears on the left of the minimized window bar.

Using a Mouse

Your applications can detect and respond to mouse events, such as left mouse button down, right mouse button down, or double-click. Mouse events can be used as an alternative to keyboard input or for manipulating what is shown on the screen.

QuickWin provides two types of mouse functions:

- *Event-based functions*, which call an application-defined callback routine when a mouse click occurs
- *Blocking (sequential) functions*, which provide blocking functions that halt an application until mouse input is made

The mouse is an asynchronous device, so the user can click the mouse anytime while the application is running (mouse input does not have to be synchronized to anything). When a mouse-click occurs, Windows sends a message to the application, which takes the appropriate action. Mouse support in applications is most often event-based, that is, a mouse-click occurs and the application does something.

However, an application can use blocking functions to wait for a mouse-click. This allows an application to execute in a particular sequential order and yet provide mouse support. QuickWin performs default processing based on mouse events.

Event-Based Functions

The QuickWin function REGISTERMOUSEEVENT registers the routine to be called when a particular mouse event occurs (left mouse button, right mouse button, double-click, and so on). You define what events you want it to handle and the routines to be called if those events occur. UNREGISTERMOUSEEVENT unregisters the routines so that QuickWin doesn't call them but uses default handling for the particular event.

By default, QuickWin typically ignores events except when mouse-clicks occur on menus or dialog controls. Note that no events are received on a minimized window. A window must be restored or maximized in order for mouse events to happen within it.

For example:

```

USE DFLIB
INTEGER(4) result
OPEN (4, FILE= 'USER')
...
result = REGISTERMOUSEEVENT (4, MOUSE$LBUTTONDBLCLK, CALCULATE)

```

This registers the routine `CALCULATE`, to be called when the user double-clicks the left mouse button while the mouse cursor is in the child window opened as unit 4. The symbolic constants available to identify mouse events are:

Mouse event ¹	Description
<code>MOUSE\$LBUTTONDOWN</code>	Left mouse button down
<code>MOUSE\$LBUTTONUP</code>	Left mouse button up
<code>MOUSE\$LBUTTONDBLCLK</code>	Left mouse button double-click
<code>MOUSE\$RBUTTONDOWN</code>	Right mouse button down
<code>MOUSE\$RBUTTONUP</code>	Right mouse button up
<code>MOUSE\$RBUTTONDBLCLK</code>	Right mouse button double-click
<code>MOUSE\$MOVE</code>	Mouse moved

¹ For every `BUTTONDOWN` and `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`.

To unregister the routine in the preceding example, use the following code:

```
result = UNREGISTERMOUSEEVENT (4, MOUSE$LBUTTONDBLCLK)
```

If `REGISTERMOUSEEVENT` is called again without unregistering a previous call, it overrides the first call. A new callback routine is then called on the specified event.

The callback routine you create to be called when a mouse event occurs should have the following prototype:

```

INTERFACE
  SUBROUTINE MouseCallBackRoutine (unit, mouseevent, keystate, &
    & MouseXpos, MouseYpos)
    INTEGER unit
    INTEGER mouseevent
    INTEGER keystate
    INTEGER MouseXpos
    INTEGER MouseYpos
  END SUBROUTINE
END INTERFACE

```

The `unit` parameter is the unit number associated with the child window where events are to be detected, and the `mouseevent` parameter is one of those listed in the preceding table. The `MouseXpos` and the `MouseYpos` parameters specify the x and y positions of the mouse during the event. The `keystate` parameter indicates the state of the shift and control keys at the time of the mouse event, and can be any **ORed** combination of the following constants:

Keystate parameter	Description
--------------------	-------------

MOUSE\$KS_LBUTTON	Left mouse button down during event
MOUSE\$KS_RBUTTON	Right mouse button down during event
MOUSE\$KS_SHIFT	Shift key held down during event
MOUSE\$KS_CONTROL	Control key held down during event

QuickWin callback routines for mouse events should do a minimum of processing and then return. While processing a callback, the program will appear to be non-responsive because messages are not being serviced, so it is important to return quickly. If more processing time is needed in a callback, another thread should be started to perform this work; threads can be created by calling the Win32 API **CreateThread**. (For more information on creating and using threads, see [Creating Multithread Applications](#).) If a callback routine does not start a new thread, the callback will not be re-entered until it is done processing.

Note: In event-based functions there is no buffering of events. Therefore, issues such as multithreading and synchronizing access to shared resources must be addressed. To avoid multithreading problems, use blocking functions rather than event-based functions. Blocking functions work well in applications that proceed sequentially. Applications where there is little sequential flow and the user jumps around the application are probably better implemented as event-based functions.

Blocking (Sequential) Functions

The QuickWin blocking function WAITONMOUSEEVENT blocks execution until a specific mouse input is received. This function is similar to INCHARQQ, except that it waits for a mouse event instead of a keystroke.

For example:

```

USE DFLIB
INTEGER(4) mouseevent, keystate, x, y, result
...
mouseevent = MOUSE$RBUTTONDOWN .OR. MOUSE$LBUTTONDOWN
result = WAITONMOUSEEVENT (mouseevent, keystate, x , y) ! Wait
! until right or left mouse button clicked, then check the keystate
! with the following:
  if ((MOUSE$KS_SHIFT .AND. keystate) == MOUSE$KS_SHIFT) then      &
& write (*,*) 'Shift key was down'
  if ((MOUSE$KS_CONTROL .AND. keystate) == MOUSE$KS_CONTROL) then &
& write (*,*) 'Ctrl key was down'

```

Your application passes a mouse event parameter, which can be any **OR** ed combination of mouse events, to **WAITONMOUSEEVENT**. The function then waits and blocks execution until one of the specified events occurs. It returns the state of the SHIFT and CTRL keys at the time of the event in the parameter keystate, and returns the position of the mouse when the event occurred in the parameters x and y.

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.

Default QuickWin Processing

QuickWin performs some actions based on mouse events. It uses mouse events to return from the FullScreen mode and to select text and/or graphics to copy to the Clipboard. Servicing the mouse event functions takes precedence over return from FullScreen mode. (ALT+ENTER can always be used to return from FullScreen mode.) Servicing mouse event functions does not take precedence over Cut/Paste selection modes. Once selection mode is over, processing of mouse event functions resumes.

QuickWin Programming Precautions

Two features of QuickWin programming need to be applied thoughtfully to avoid non-responsive programs that halt an application while waiting for a process to execute or input to be entered in a child window. The two features are described in the topics:

- [Blocking Procedures](#)
- [Callback Routines](#)

Blocking Procedures

Procedures that wait for an event before allowing the program to proceed, such as `READ` or `WAITONMOUSEEVENT`, both of which wait for user input, are called *blocking procedures* because they block execution of the program until the awaited event occurs. QuickWin child processes can contain multiple callback routines; for example, a different routine to be called for each menu selection and each kind of mouse-click (left button, right button, double-click, and so on).

Problems can arise when a process and its callback routine, or two callback routines within the same process, both contain blocking procedures. This is because each QuickWin child process supports a primary and secondary thread.

As a result of selecting a menu item, a menu procedure may call a blocking procedure, while the main thread of the process has also called a blocking procedure. For example, say you have created a file menu, which contains an option to LOAD a file. Selecting the LOAD menu option calls a blocking function that prompts for a filename and waits for the user to enter the name. However, a blocking call such as `WAITONMOUSEEVENT` can be pending in the main process thread when the user selects the LOAD menu option, so two blocking functions are initiated.

When QuickWin has two blocking calls pending, it displays a message in the status bar that corresponds to the blocking call first encountered. If there are further callbacks with other blocking procedures in the two threads, the status bar may not correspond to the actual input pending, execution can appear to be taking place in one thread when it is really blocked in another, and the application can be confusing and misleading to the user.

To avoid this confusion, you should try not to use blocking procedures in your callback routines. QuickWin will not accept more than one `READ` or `INCHARQQ` request through user callbacks from the same child window at one time. If one `READ` or `INCHARQQ` request is pending, subsequent `READ` or `INCHARQQ` requests will be ignored and -1 will be returned to the caller.

If you have a child window that in some user scenario might call multiple callback routines containing `READ` or `INCHARQQ` requests, you need to check the return value to make sure the request has been successful, and if not, take appropriate action, for example, request again.

This protective QuickWin behavior does not guard against multiple blocking calls through mouse selection of menu input options. As a general rule, using blocking procedures in callback routines is not advised, since the results can lead to program flow that is unexpected and difficult to interpret.

Callback Routines

All callback routines run in a separate thread from the main program. So, all multithread issues are in full force. In particular, sharing data, drawing to windows, and doing I/O must be properly coordinated and controlled. The sample application POKER.F90 (in the \DF\SAMPLES\GENERAL\POKER subdirectory) is a good example of how to control access to shared resources.

QuickWin callback routines, both for menu callbacks and mouse callbacks, should do a minimum of processing and then return. While processing a callback, the program will appear to be non-responsive because messages are not being serviced. This is why it is important to return quickly.

If more processing time is needed in a callback, another thread should be started to perform this work; threads can be created by calling the Win32 API **CreateThread**. (For more information on creating and using threads, see [Creating Multithread Applications](#).) If a callback routine does not start a new thread, the callback will not be reentered until it is done processing.

Simulating Nonblocking I/O

QuickWin does not accept unsolicited input. You get beeps if you type into an active window if no **READ** or **GETCHARQQ** has been done. Because of this, it is necessary to do a **READ** or **GETCHARQQ** in order for a character to be accepted. But this type of blocking I/O puts the program to sleep until a character has been typed.

In Console applications, **PEEKCHARQQ** can be used to see if a character has already been typed. However, **PEEKCHARQQ** does not work under QuickWin, since QuickWin has no console buffer to accept unsolicited input. Because of this limitation, **PEEKCHARQQ** cannot be used as it is with Console applications to see whether a character has already been typed.

One way to simulate **PEEKCHARQQ** with QuickWin applications is to use a multithread application. One thread does a **READ** or **GETCHARQQ** and is blocked until a character typed. The other thread is in a loop doing useful work and checking in the loop to see if the other thread has received input.

For more information, see PEEKAPP.F90 in the \DF\SAMPLES\ADVANCED\PEEKAPP subdirectory.

Using Dialogs

Dialogs are a user-friendly way to solicit application control. As your application executes, you can make a dialog box appear on the screen and the user can click on a button or scroll bar to enter data or choose what happens next. With the dialog functions provided with Visual Fortran and Developer Studio, you can add dialog boxes to your Windows (Win32), QuickWin (multiple doc.), Standard Graphics (QuickWin single doc.), Console, DLL, and Library project. These functions define dialog boxes and their controls (scroll bars and buttons), and call your subroutines to respond to user selections.

There are two steps to making a dialog:

1. Specify the appearance of the dialog box and the names and properties of the controls it contains.
2. Write an application that activates those controls by recognizing and responding to user selections.

This section covers the following topics:

- [Using the Resource Editor to Design a Dialog](#)
- [Writing a Dialog Application](#)
- [Dialog Functions](#)
- [Dialog Controls](#)
- [Using Dialog Controls](#)

Using the Resource Editor to Design a Dialog

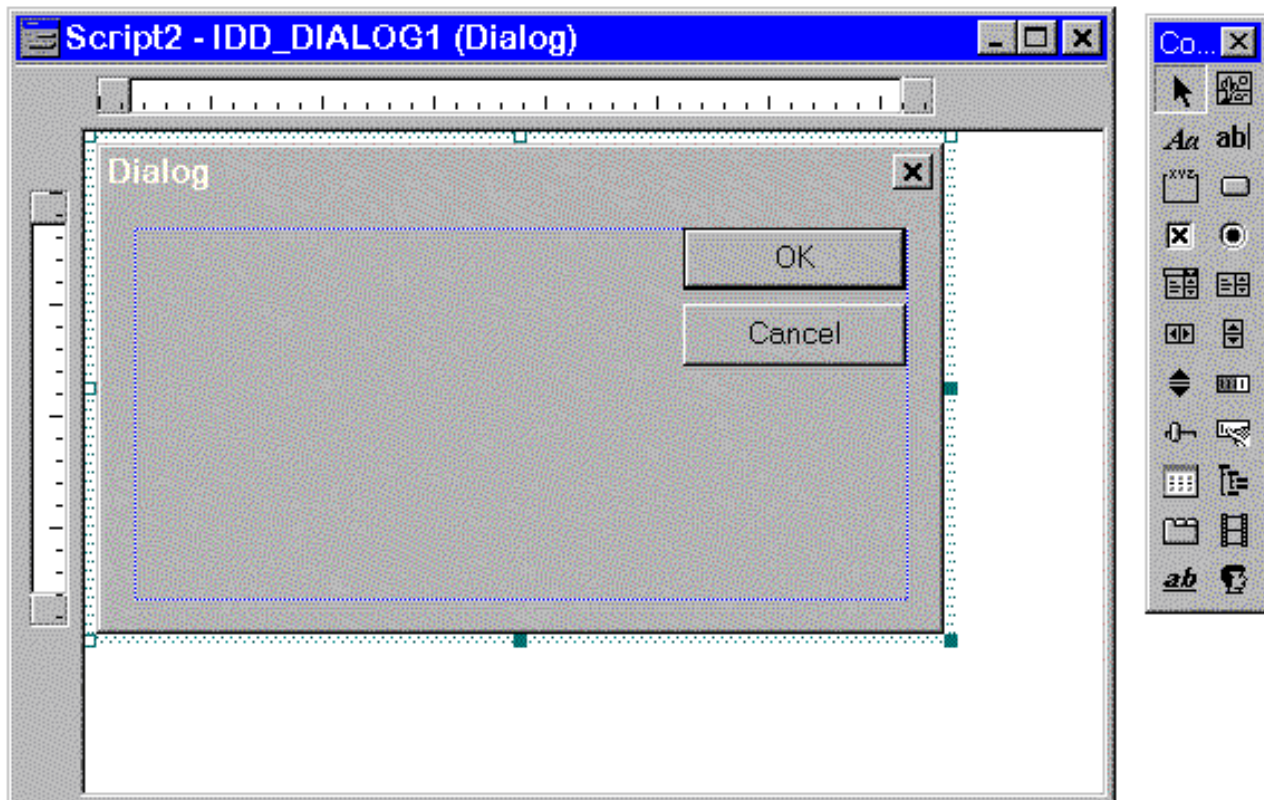
You design the appearance of the dialog box, choose and name the controls within it, and set other control properties with the Resource Editor. This section goes through the design of a dialog box, and uses as an example a dialog that converts temperatures between Celsius and Fahrenheit.

► To open the dialog editor

1. From the Insert menu, choose Resource.
2. From the list of possible resources, choose Dialog.
3. Click the New button. The dialog editor appears on the screen as shown below.

A blank dialog box appears at the left and a toolbar of available controls appears on the right. If you place the cursor over a control on the toolbar, the name of the control appears.

Figure: Dialog Editor Sample 1

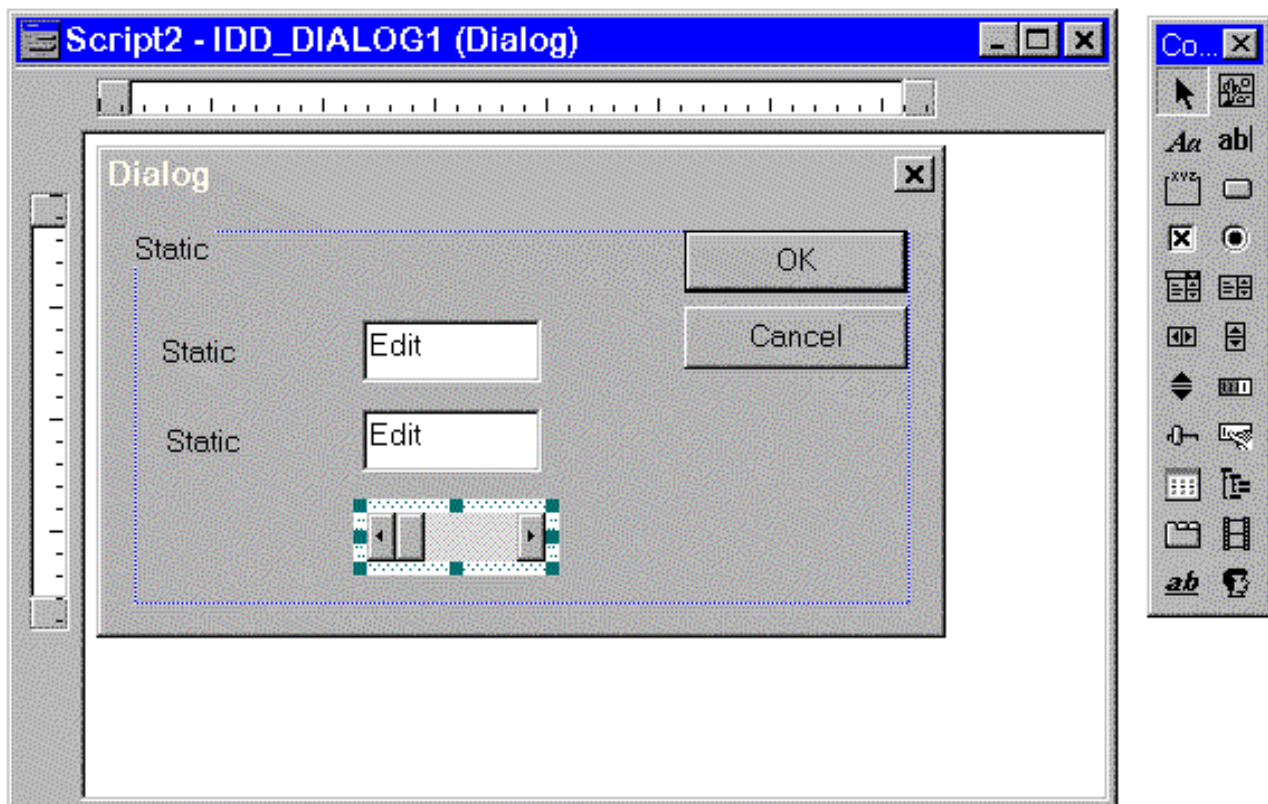


► **To add controls to the dialog box**

1. Point at one of the available controls on the toolbar, hold down the left mouse button and drag the control to the dialog box.
2. Place the control where you want it to be on the dialog box and release the mouse button. You can delete controls by selecting them with the mouse, then pressing the DEL key

The following figure shows a Horizontal Scroll bar, two Edit boxes, two Static text lines, and a Group box added to the dialog box. The OK and CANCEL buttons were added for you by the Resource Editor, but they are not in any way special and can be deleted, moved, resized, or renamed.

Figure: Dialog Editor Sample 2



► **To specify the names and properties of the added controls**

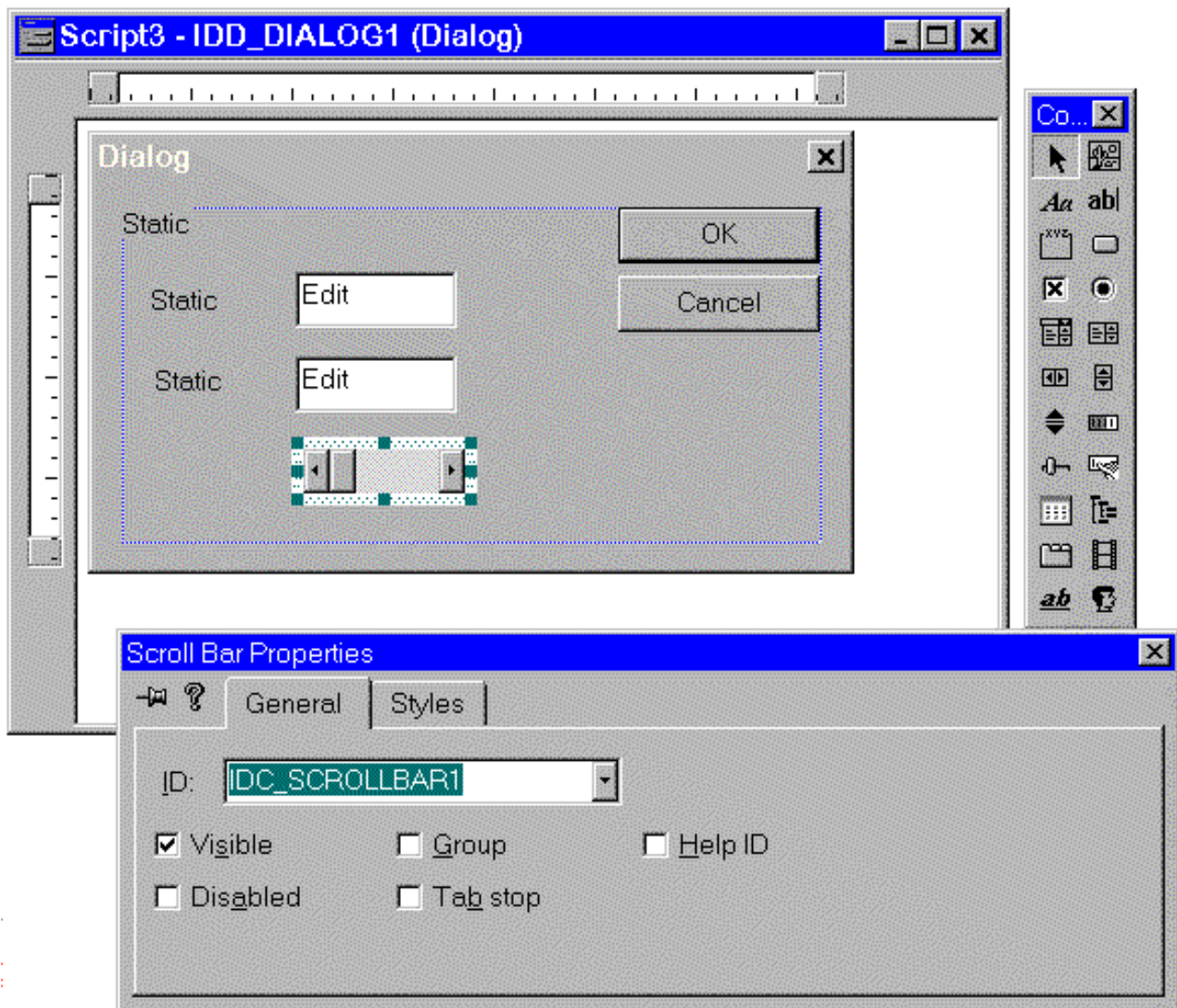
1. Click twice on one of the controls in your dialog box with the left mouse button. A Properties box appears showing the default name and properties for that control.

The following figure shows the Properties box for the Horizontal Scroll bar with the default values.

2. Change the control name by typing over the default name (`IDC_SCROLLBAR1` in the following figure).
3. Check or uncheck the available options to change the control's properties. (The Visible option in the following figure is checked by default.)
4. Click the left mouse button in the upper-right corner of the window Properties box to save the control's properties and to close the box.

Repeat the same process for each control and for the dialog box itself.

Figure: Dialog Editor Sample 3



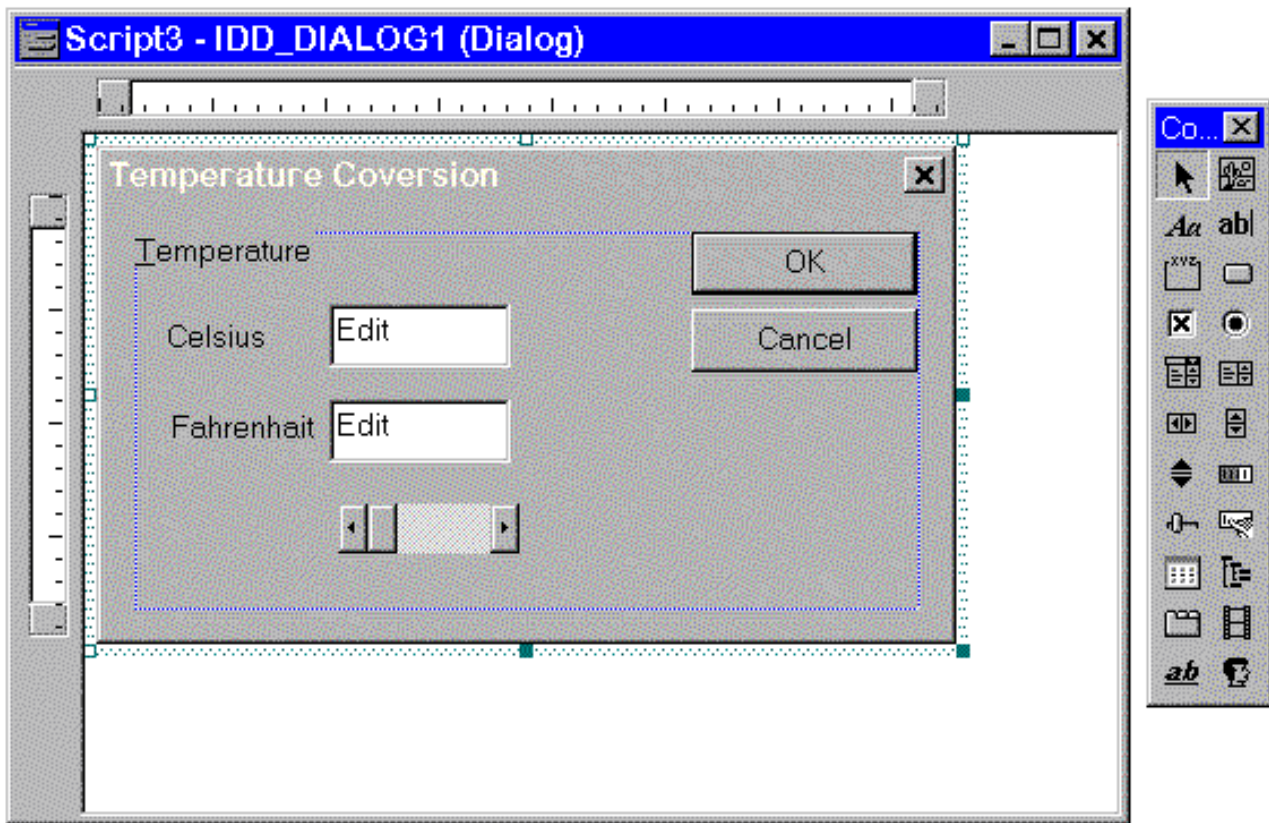
To use the controls from within a program, you need symbolic names for each of them. In this example, the Horizontal Scroll bar symbolic name is changed in the Properties box to `IDC_SCROLLBAR_TEMPERATURE`. This is how the control will be referred to in your program; for example, when you get the slide position:

```
INTEGER slide_position
retlog = DLGGET (dlg, IDC_SCROLLBAR_TEMPERATURE,           &
                slide_position, DLG_POSITION)
```

The top Edit box is named `IDC_EDIT_CELSIUS`. The Static text next to it is named `IDC_TEXT_CELSIUS` and set to the left-aligned text "Celsius". The lower Edit box is named `IDC_EDIT_FAHRENHEIT`, and the Static text next to it is named `IDC_TEXT_FAHRENHEIT` and set to the left-aligned text "Fahrenheit".

The Group box is named `IDC_BOX_TEMPERATURE`, and its caption is set to `&Temperature` (the ampersand (&) underlines the letter "T" and makes it a Windows hotkey, activated with ALT+T). The dialog itself is named `IDD_TEMP` and its caption is set to Temperature Conversion. All other control properties are left at the default values. The resulting dialog box is shown in the following figure:

Figure: Dialog Editor Sample 4



► To save the dialog box as a resource file

1. From the File menu, choose Save As.
2. Enter a resource filename for your file.

In this example, the resource file is given the name `TEMP.RC`. Developer Studio saves the resource file and creates an include file with the name `RESOURCE.FD`.

At this point the appearance of the dialog box is finished and the controls are named, but the box can't function on its own. An application must be created to run it.

For further information on control properties, see:

- [Setting Control Properties](#)
- [The Include \(.FD\) File](#)

Setting Control Properties

Help is available within the Resource Editor to explain the options for each of the dialog controls.

Some of the controls have two Properties sets: General and Styles. Click the mouse on the name of the Properties set you want to view or modify. You can change the dialog box itself by double-clicking the left mouse button in any clear area in the box. The Properties box opens for the dialog.

To change where your dialog appears on the screen, change the x and y values in the Properties box. These specify the screen pixel position of the dialog box's upper-left corner. You can change the size

of the dialog box by holding down the left mouse button as you drag the right or lower perimeter of the box.

You can use the scroll bars to move the view region if you have sized your dialog window to be larger than the edit window. If you want to edit the appearance of the dialog box later, you can open the resource file (.RC) from the File menu, and click on the dialog icon. Alternatively, you can select the Resource View pane. The Resource Editor is automatically invoked and the dialog box is opened.

The Include (.FD) File

Each control in a dialog box has a unique integer identifier. When the Resource Editor creates the include file (.FD), it assigns the PARAMETER attribute to each control and to the dialog box itself, so they become named constants. It also assigns each control and the dialog box an integer value. You can read the list of names and values in your dialog boxes include file (for example, TEMP.FD).

When your application uses a control, it can refer to the control or dialog box by its name (for example, IDC_SCROLLBAR_TEMPERATURE or IDD_TEMP), or by its integer value. If you want to rename a control or make some other change to your dialog box, you should make the change through the Resource Editor in Developer Studio. Do not use a text editor to alter your .FD include file because the dialog resource will not be able to access the changes.

Writing a Dialog Application

When creating a dialog box with the Resource Editor, you specify the types of displays and controls that are to be included in the box. You then must provide procedures to make the dialog box active. These procedures use both dialog functions and your subroutines to control your program's response to the user's dialog box input.

You give your application access to your dialog resource file by adding the .RC file to your project, giving your application access to the dialog include file, and associating the dialog properties in these files with the dialog type (see [Initializing and Activating the Dialog Box](#)).

Your application must include the statement **USE DFLOGM** to access the dialog functions, and it must include the .FD file the Resource Editor created for your dialog. For example:

```
PROGRAM TEMPERATURE
USE DFLOGM
IMPLICIT NONE
INCLUDE 'TEMP.FD'
CALL DoDialog( )
END PROGRAM
```

The following sections describe how to code a dialog application:

- [Initializing and Activating the Dialog Box](#)
- [Callback Routines](#)

Initializing and Activating the Dialog Box

Each dialog box has an associated variable of the derived type `dialog`. The `dialog` derived type is

defined in the DFLOGM.F90 module; you access it with **USE DFLOGM**. When you write your dialog application, refer to your dialog box as a variable of type `dialog`. For example:

```
USE DFLOGM
INCLUDE 'TEMP.FD'
TYPE (dialog) dlg
LOGICAL return
return = DLGINIT( IDD_TEMP, dlg )
```

This code associates the `dialog` type with the dialog (`IDD_TEMP` in this example) defined in your resource and include files (`TEMP.RC` and `TEMP.FD` in this example).

You give your application access to your dialog resource file by adding the `.RC` file to your project. You give your application access to the dialog include file by including the `.FD` file in each subprogram. You associate the dialog properties in these files with the `dialog` type by calling **DLGINIT** with your dialog name.

An application that controls a dialog box should perform the following actions:

1. Call **DLGINIT** to initialize the `dialog` type and associate your dialog and its properties with the type.
2. Initialize the controls with the dialog set functions, such as **DLGSET**.
3. Set the callback routines to be executed when a user manipulates a control in the dialog box with **DLGSETSUB**.
4. Run the dialog with **DLGMODAL**.
5. Retrieve control information with the dialog get functions, such as **DLGGET**.
6. Free resources from the dialog with **DLGUNINIT**.

As an example of activating a dialog box and controls, the following code initializes the temperature dialog box and controls created in the previous example. It also sets the callback routine as `UpdateTemp`, displays the dialog box, and releases the dialog resources when done:

```
SUBROUTINE DoDialog( )
USE DFLOGM
IMPLICIT NONE
INCLUDE 'TEMP.FD'

INTEGER retint
LOGICAL retlog
TYPE (dialog) dlg
EXTERNAL UpdateTemp
! Initialize.
IF ( .not. DlgInit( idd_temp, dlg ) ) THEN
  WRITE (*,*) "Error: dialog not found"
ELSE
! Set up temperature controls.
  retlog = DlgSet( dlg, IDC_SCROLLBAR_TEMPERATURE, 200, DLG_RANGE)
  retlog = DlgSet( dlg, IDC_EDIT_CELSIUS, "100" )
  CALL UpdateTemp( dlg, IDC_EDIT_CELSIUS, DLG_CHANGE)
  retlog = DlgSetSub( dlg, IDC_EDIT_CELSIUS, UpdateTemp )
  retlog = DlgSetSub( dlg, IDC_EDIT_FAHRENHEIT, UpdateTemp )
  retlog = DlgSetSub( dlg, IDC_SCROLLBAR_TEMPERATURE, UpdateTemp )
! Activate the dialog.
  retint = DlgModal( dlg )
! Release dialog resources.
  CALL DlgUninit( dlg )
END IF
END SUBROUTINE DoDialog
```

The dialog functions, such as **DLGSET** and **DLGSETSUB**, refer to the dialog controls by the names you assigned to them in the Properties box while creating the dialog box in the Resource Editor. For example:

```
retlog = DlgSet( dlg, IDC_SCROLLBAR_TEMPERATURE, 200, DLG_RANGE)
```

In this statement, the dialog function **DLGSET** assigns the control named `IDC_SCROLLBAR_TEMPERATURE` a value of 200. The index `DLG_RANGE` specifies that this value is a scroll bar range. Consider the following:

```
retlog = DlgSet( dlg, IDC_EDIT_CELSIUS, "100" )
CALL UpdateTemp( dlg, IDC_EDIT_CELSIUS, DLG_CHANGE)
```

The preceding statements set the dialog's top Edit box, named `IDC_EDIT_CELSIUS` in the Resource Editor, to an initial value of 100, and calls the routine `UpdateTemp` to write this initial value to the screen. Consider the following:

```
retlog = DlgSetSub( dlg, IDC_EDIT_CELSIUS, UpdateTemp )
retlog = DlgSetSub( dlg, IDC_EDIT_FAHRENHEIT, UpdateTemp )
retlog = DlgSetSub( dlg, IDC_SCROLLBAR_TEMPERATURE, UpdateTemp )
```

The preceding statements associate the callback routine `UpdateTemp` with the three controls.

Routines are assigned to the controls with the function **DLGSETSUB**. Its first argument is the dialog variable, the second is the control name, the third is the name of the routine you have written for the control, and the optional fourth argument is an index to select between multiple routines. You can set the callback routines for your dialog controls anywhere in your application: before opening your dialog with **DLGMODAL** or from within another callback routine.

Dialog Callback Routines

All callback routines should have the following interface:

SUBROUTINE callback (*dlg*, *control_name*, *callbacktype*)

dlg

Refers to the dialog box and allows the callback to change values of the dialog controls.

control_name

Is the name of the control that caused the callback.

callbacktype

Indicates what callback is occurring (for example, `DLG_CLICKED`, `DLG_CHANGE`, `DLG_DBLCLICK`).

The last two 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. For example, all the controls in the temperature dialog in the previous example are associated with the same callback routine, `UpdateTemp`. You can also associate more than one callback routine with the same control, but you must then provide an index parameter to indicate which callback is to be used.

The following is an example of a callback routine:

```

SUBROUTINE UpdateTemp( dlg, control_name, callbacktype )
USE DFLOGM
IMPLICIT NONE
TYPE (dialog) dlg
INTEGER control_name
INTEGER callbacktype
INCLUDE 'TEMP.FD'
CHARACTER(256) text
INTEGER cel, far, retint
LOGICAL retlog
! Suppress compiler warnings for unreferenced arguments.
INTEGER local_callbacktype
local_callbacktype = callbacktype

SELECT CASE (control_name)
CASE (IDC_EDIT_CELSIUS)
! Celsius value was modified by the user so
! update both Fahrenheit and Scroll bar values.
retlog = DlgGet( dlg, IDC_EDIT_CELSIUS, text )
READ (text, *, iostat=retint) cel
IF ( retint .eq. 0 ) THEN
far = (cel-0.0)*((212.0-32.0)/100.0)+32.0
WRITE (text,*) far
retlog = DlgSet( dlg, IDC_EDIT_FAHRENHEIT, &
& TRIM(ADJUSTL(text)) )
retlog = DlgSet( dlg, IDC_SCROLLBAR_TEMPERATURE, cel, &
& DLG_POSITION )
END IF
CASE (IDC_EDIT_FAHRENHEIT)
! Fahrenheit value was modified by the user so
! update both celsius and Scroll bar values.
retlog = DlgGet( dlg, IDC_EDIT_FAHRENHEIT, text )
READ (text, *, iostat=retint) far
IF ( retint .eq. 0 ) THEN
cel = (far-32.0)*(100.0/(212.0-32.0))+0.0
WRITE (text,*) cel
retlog = DlgSet( dlg, IDC_EDIT_CELSIUS, TRIM(ADJUSTL(text)) )
retlog = DlgSet( dlg, IDC_SCROLLBAR_TEMPERATURE, cel, &
& DLG_POSITION )
END IF
CASE (IDC_SCROLLBAR_TEMPERATURE)
! Scroll bar value was modified by the user so
! update both Celsius and Fahrenheit values.
retlog = DlgGet( dlg, IDC_SCROLLBAR_TEMPERATURE, cel, &
& DLG_POSITION )
far = (cel-0.0)*((212.0-32.0)/100.0)+32.0
WRITE (text,*) far
retlog = DlgSet( dlg, IDC_EDIT_FAHRENHEIT, TRIM(ADJUSTL(text)) )
WRITE (text,*) cel
retlog = DlgSet( dlg, IDC_EDIT_CELSIUS, TRIM(ADJUSTL(text)) )
END SELECT
END SUBROUTINE UpdateTemp

```

Each control in a dialog box, except a pushbutton, has a default callback that performs no action. The default callback for a pushbutton's click event sets the return value of the dialog to the pushbutton's name and then exits the dialog. This makes all pushbuttons exit the dialog by default, and gives the OK and CANCEL buttons good default behavior. The routine that calls **DLG_MODAL** can then test to see which pushbutton caused the dialog to exit.

Callbacks for a particular control are called after the value of the control has been changed by the user's action. Calling **DLGSET** does not cause a callback to be called for the changing value of a control. In particular, when inside a callback, performing a **DLGSET** on a control will not cause the

associated callback for that control to be called.

Calling **DLGSET** before or after **DLGMODAL** has been called also does not cause the callback to be called. If the callback needs to be called, it can be called manually with **CALL callbackroutine** after the **DLGSET** is performed.

Dialog Functions

You can use dialog functions as you would any intrinsic or run-time function. They are compatible with Standard Graphics, QuickWin, and Windows (Win32) project types. The dialog functions can:

- Initialize and close the dialog box
- Retrieve user input from a dialog box
- Display data in the dialog box
- Modify the dialog box controls

The include file (.FD) of the dialog box contains the names of the dialog controls that you specified in the Properties box of the Resource Editor when you created the dialog box. The module DFLOGM.MOD contains predefined variable names and type definitions. These control names, variables, and type definitions are used in the dialog function argument lists to manage your dialog box.

The dialog functions are listed in the following table:

Dialog function	Purpose
<u>DLGEXIT</u>	Closes an open dialog
<u>DLGGET</u>	Gets the value of a control variable
<u>DLGGETCHAR</u>	Gets the value of a character control variable
<u>DLGGETINT</u>	Gets the value of an integer control variable
<u>DLGGETLOG</u>	Gets the value of a logical control variable
<u>DLGINIT</u>	Initializes the dialog
<u>DLGMODAL</u>	Displays a dialog box
<u>DLGSET</u>	Assigns a value to a control variable
<u>DLGSETCHAR</u>	Assigns a value to a character control variable
<u>DLGSETINT</u>	Assigns a value to an integer control variable
<u>DLGSETLOG</u>	Assigns a value to a logical control variable
<u>DLGSETRETURN</u>	Sets the return value for DLGMODAL
<u>DLGSETSUB</u>	Assigns a defined callback routine to a control
<u>DLGUNINIT</u>	Deallocates memory for an initialized dialog

These functions are described in the *Reference* (see also Dialog Procedures: table).

Dialog Controls

Each control in a dialog box has a unique integer identifier and name. You specify the name in the Properties box for each control within the Resource Editor, and the Resource Editor assigns the

PARAMETER attribute and an integer value to each control name. You can refer to a control by its name, for example `IDC_SCROLLBAR_TEMPERATURE`, or by its integer value, which you can read from the include (.FD) file.

Each control has one or more variables associated with it, called *control indexes*. These indexes can be integer, logical, character, or external. For example, a plain Button has three associated variables: one is a logical value associated with its current state, one is a character variable that determines its title, and the third is an external variable that indicates the subroutine to be called if a mouse click occurs.

Controls can have multiple variables of the same type. For example, the scroll bar control has four integer variables associated with it: scroll bar position, scroll bar range, position change taken if the user clicks the scroll bar arrow (small step) , and the position change if the user clicks on the scroll bar space next to the slide (big step).

Controls and their indexes are discussed in:

- [Control Indexes](#)
- [Available Indexes for Each Dialog Control](#)
- [Specifying Control Indexes](#)

Control Indexes

The value of a dialog control's index is set with the `DLGSET` functions: **DLGSET**, **DLGSETINT**, **DLGSETLOG**, **DLGSETCHAR**, and **DLGSETSUB**. The control name and control index name are arguments to the **DLGSET** functions and specify the particular control index being set. For example:

```
retlog = DlgSet( dlg, IDC_SCROLLBAR_TEMPERATURE, 45, DLG_POSITION )
```

The index `DLG_POSITION` specifies the scroll bar position is to be set to 45. Consider the following:

```
retlog = DlgSet( dlg, IDC_SCROLLBAR_TEMPERATURE, 200, DLG_RANGE )
```

In this statement, the index `DLG_RANGE` specifies the scroll bar range is to be set to 200. The **DLGSET** functions have the following syntax:

```
return = DLGSET (dlg, control_name, value, control_index_name)
```

The *control_index_name* determines what the *value* in the **DLGSET** function means.

The control index names are declared in the module `DFLOGM.MOD` and should not be declared in your routines. Available control indexes and how they specify the interpretation of the *value* argument are listed in the following table.

Table: Control Indexes	
Control index	How the value is interpreted
DLG_BIGSTEP	The amount of change that occurs in a Scroll bar's position when the user clicks beside the Scroll bar slide (default = 10)
DLG_CHANGE	A subroutine called after the user has modified a control and the control has been updated on the screen

DLG_CLICKED	A subroutine called when the control receives a mouse-click
DLG_DBLCLICK	A subroutine called when a control is double-clicked
DLG_DEFAULT	Same as not specifying a control index
DLG_ENABLE	The enable state of the control (<i>value</i> = .TRUE. means enabled, <i>value</i> = .FALSE. means disabled)
DLG_NUMITEMS	The total number of items in a list box or combo box
DLG_POSITION	The current position of the Scroll bar
DLG_RANGE	The maximum value of a Scroll bar position (default = 100); the minimum is always 1
DLG_SELCHANGE	A subroutine called when the selection in a list box or combo box changes
DLG_SMALLSTEP	The amount of change that occurs in a Scroll bar's position when the user clicks on a scroll bar arrow
DLG_STATE	The user changeable state of a control
DLG_TITLE	The title text associated with a control
DLG_UPDATE	A subroutine called after the user has modified the control state but before the control has been updated on the screen

The index names associated with dialog controls do not need to be used unless there is more than one variable of the same type for the control and you do not want the default variable. For example:

```
retlog =DlgSet( dlg, IDC_SCROLLBAR_TEMPERATURE, 45, DLG_POSITION )
retlog =DlgSet( dlg, IDC_SCROLLBAR_TEMPERATURE, 45)
```

These statements both set the Scroll bar position to 45, because `DLG_POSITION` is the default control index for the scroll bar.

For more information on dialog controls, see [Available Indexes for Each Dialog Control](#).

Available Indexes for Each Dialog Control

The available indexes and defaults for each of the controls are listed in the following table:

Control Type	Integer index name	Logical index name	Character index name	Subroutine index name
Static text		DLG_ENABLE	DLG_TITLE	
Group box		DLG_ENABLE	DLG_TITLE	
Button		DLG_ENABLE	DLG_TITLE	DLG_CLICKED
Check box		DLG_STATE (default) DLG_ENABLE	DLG_TITLE	DLG_CLICKED
Radio button		DLG_STATE (default) DLG_ENABLE	DLG_TITLE	DLG_CLICKED
Edit box		DLG_ENABLE	DLG_STATE	DLG_CHANGE (default) DLG_UPDATE
Scroll bar	DLG_POSITION (default)	DLG_ENABLE		DLG_CHANGE

	DLG_RANGE DLG_BIGSTEP DLG_SMALLSTEP			
List box	DLG_NUMITEMS Sets or returns the total number of items in a list, or you can include an index, 1 to <i>n</i> , to determine which list items have been selected and their order	DLG_ENABLE	DLG_STATE By default, sets or returns the text of the first selected item, or you can include an index, 1 to <i>n</i> , to set or return the text of a particular item	DLG_SELCHANGE (default) DLG_DBLCLICK
Combo box	DLG_NUMITEMS Sets or returns the total number of items in a list, or you can include an index, 1 to <i>n</i> , to determine which list item has been selected	DLG_ENABLE	DLG_STATE By default, sets or returns the text of the selected item or first item in the list, or you can include an index, 1 to <i>n</i> , to set or return indicates the text of a particular item	DLG_SELCHANGE (default) DLG_DBLCLICK DLG_CHANGE DLG_UPDATE
Drop-down list box	DLG_NUMITEMS (default) Sets or returns the total number of items in a list, or you can include an index, 1 to <i>n</i> , to determine which list item has been selected DLG_STATE Sets or returns the index of the selected item	DLG_ENABLE	DLG_STATE By default, sets or returns the text of the selected item or first item in the list, or you can include an index, 1 to <i>n</i> , to set or return indicates the text of a particular item	DLG_SELCHANGE (default) DLG_DBLCLICK

For an overview on control indexes, see [Control Indexes](#).

Specifying Control Indexes

Where there is only one possibility for a particular dialog control's index type (integer, logical, character, or subroutine), you do not need to specify the control index name in an argument list. For example, you can set the Static text control `IDC_TEXT_CELSIUS` to a new value with either of the following statements:

```
retlog = DLGSETCHAR (dlg, IDC_TEXT_CELSIUS, "New Celsius Title", &
& DLG_TITLE)
retlog = DLGSET (dlg, IDC_TEXT_CELSIUS, "New Celsius Title")
```

You do not need the control index `DLG_TITLE` because there is only one character index for a Static text control. The generic function [DLGSET](#) chooses the control index to change based on the

argument type, in this case CHARACTER.

For each type of index, you can use the generic **DLGSET** function or the specific **DLGSET** function for that type: **DLGSETINT**, **DLGSETLOG**, or **DLGSETCHAR**. For example, you can disable the Static text control `IDC_TEXT_CELSIUS` by setting its logical value to **.FALSE.** with either **DLGSET** or **DLGSETLOG**:

```
retlog = DLGSETLOG (dlg, IDC_TEXT_CELSIUS, .FALSE., DLG_ENABLE)
retlog = DLGSET (dlg, IDC_TEXT_CELSIUS, .FALSE., DLG_ENABLE)
```

In both these cases, the control index `DLG_ENABLE` can be omitted because there is only one logical control index for Static text controls.

You can query the value of a particular control index with the **DLGGET** functions, **DLGGET**, **DLGGETINT**, **DLGGETLOG**, and **DLGGETCHAR**. For example:

```
INTEGER current_val
LOGICAL are_you_enabled
retlog = DLGGET (dlg, IDC_SCROLLBAR_TEMPERATURE, current_val, &
& DLG_RANGE)
retlog = DLGGET (dlg, IDC_SCROLLBAR_TEMPERATURE, are_you_enabled, &
& DLG_ENABLE)
```

This code returns the range and the enable state of the scroll bar. The arguments you declare (`current_val` and `are_you_enabled` in the preceding example) to hold the queried values must be of the same type as the values retrieved. If you use specific **DLGGET** functions such as **DLGGETINT** or **DLGGETCHAR**, the control index value retrieved must be the appropriate type. For example, you cannot use **DLGGETCHAR** to retrieve an integer or logical value. The **DLGGET** functions return **.FALSE.** for illegal type combinations. You cannot query for the name of an external callback routine.

In general, it is better to use the generic functions **DLGSET** and **DLGGET** rather than their type-specific variations because then you do not have to worry about matching the function to type of value set or retrieved. **DLGSET** and **DLGGET** perform the correct operation automatically, based on the type of argument you pass to them.

For more information on these routines, see the *Reference*.

Using Dialog Controls

The dialog controls provided in the Resource Editor are versatile and flexible and when used together can provide a sophisticated user-friendly interface for your application. This section discusses the available dialog controls.

Any control can be disabled by your application at any time, so that it no longer changes or responds to the user. This is done by setting the control index `DLG_ENABLE` to **.FALSE.** with **DLGSET** or **DLGSETLOG**. For example:

```
LOGICAL retlog
retlog = DLGSET (dlg, IDC_CHECKBOX1, .FALSE., DLG_ENABLE)
```

This example disables the control named `IDC_CHECKBOX1`.

When you create your dialog box in the Resource Editor, the dialog controls are given a tab order. When the user hits the TAB key, the dialog box focus shifts to the next control in the tab order. By default, the tab order of the controls follows the order in which they were created. This may not be the order you want.

You can change the order by opening the Layout menu and choosing Tab Order (or by pressing the key combination CTRL+D) in the Resource Editor. A tab number will appear next to each control. Click the mouse on the control you want to be first, then on the control you want to be second in the tab order and so on. Tab order also determines which control gets the focus if the user presses the Group box hotkey. (See [Using Group Boxes](#).)

The following sections describe the function and use of the dialog controls:

- [Using Static Text](#)
- [Using Edit Boxes](#)
- [Using Group Boxes](#)
- [Using Check Boxes and Radio Buttons](#)
- [Using Buttons](#)
- [Using List Boxes and Combo Boxes](#)
- [Using Scroll Bars](#)
- [Setting Return Values and Exiting](#)

Using Static Text

Static text is an area in the dialog that your application writes text to. The user cannot change it. Your application can modify the Static text at any time, for instance to display a current user selection, but the user cannot modify the text. Static text is typically used to label other controls or display messages to the user.

Using Edit Boxes

An Edit box is an area that your application can write text to at anytime. However, unlike Static Text, the user can write to an Edit box by clicking the mouse in the box and typing. The following statements write to an Edit box:

```
CHARACTER(20) text /"Send text"/  
retlog = DLGSET (dlg, IDC_EDITBOX1, text)
```

The next statement reads the character string in an Edit box:

```
retlog = DLGGET (dlg, IDC_EDITBOX1, text)
```

The values a user enters into the Edit box are always retrieved as character strings, and your application needs to interpret these strings as the data they represent. For example, numbers entered by the user are interpreted by your application as characterstrings. Likewise, numbers you write to the Edit box are sent as character strings. You can convert between numbers and strings by using internal read and write statements to make type conversions.

To read a number in the Edit box, retrieve it as a character string with **DLGGET** or **DLGGETCHAR**, and then execute an internal read using a variable of the numeric type you want (such as integer or real). For example:

```

REAL      x
LOGICAL  retlog
CHARACTER(256) text
retlog = DLGGET (dlg, IDC_EDITBOX1, text)
READ (text, *) x

```

In this example, the real variable `x` is assigned the value that was entered into the Edit box, including any decimal fraction.

Complex and double complex values are read the same way, except that your application must separate the Edit box character string into the real part and imaginary part. You can do this with two separate Edit boxes, one for the real and one for the imaginary part, or by requiring the user to enter a separator between the two parts and parsing the string for the separator before converting. If the separator is a comma (,) you can read the string with two real edit descriptors without having to parse the string.

To write numbers to an Edit box, do an internal write to a string, then send the string to the Edit box with **DLGSET**. For example:

```

INTEGER  j
LOGICAL  retlog
CHARACTER(256) text
WRITE (text, '(I4)') j
retlog = DLGSET (dlg, IDC_EDITBOX1, text)

```

Visual Fortran does not support multiline Edit boxes.

Using Group Boxes

A Group box visually organizes a collection of controls as a group. When you select Group box in Resource Editor, you create an expanding (or shrinking) box around the controls you want to group and give the group a title. You can add a hotkey to your group title with an ampersand (&). For example, consider the following group title:

```
&Temperature
```

This causes the "T" to be underlined in the title and makes it a hotkey. When the user presses the key combination ALT+T, the focus of the dialog box shifts to the next control after the Group box in the tab order. This control should be a control in the group. (You can view and change the tab order from the Layout/Tab Order menu option in the Resource Editor.)

Disabling the Group box disables the hotkey, but does not disable any of the controls within the group. As a matter of style, you should generally disable the controls in a group when you disable the Group box.

Using Check Boxes and Radio Buttons

Check boxes and Radio Buttons present the user with an either-or choice. A Radio Button is pushed or not, and a Check box is checked or not. You use **DLGGET** or **DLGGETLOG** to check the state of these controls. Their state is a logical value that is **.TRUE.** if they are pushed or checked, and **.FALSE.** if they are not. For example:

```
LOGICAL pushed_state, checked_state, retlog
retlog = DLGGET (dlg, IDC_RADIOBUTTON1, pushed_state)
retlog = DLGGET (dlg, IDC_CHECKBOX1, checked_state)
```

If you need to change the state of the button, for initialization or in response to other user input, you use **DLGSET** or **DLGSETLOG**. For example:

```
LOGICAL retlog
retlog = DLGSET (dlg, IDC_RADIOBUTTON1, .FALSE.)
retlog = DLGSET (dlg, IDC_CHECKBOX1, .TRUE.)
```

Using Buttons

Unlike Check Boxes and Radio Buttons, Buttons do not have a state. They do not hold the value of being pushed or not pushed. When the user clicks on a Button with the mouse, the Button's callback routine is called. Thus, the purpose of a Button is to initiate an action. The external procedure you assign as a callback determines the action initiated. For example:

```
LOGICAL retlog
EXTERNAL DisplayTime
retlog = DlgSetSub( dlg, IDC_BUTTON_TIME, DisplayTime)
```

Visual Fortran does not support user-drawn Buttons.

Using List Boxes and Combo Boxes

List boxes and Combo boxes are used when the user needs to select a value from a set of many values. They are similar to a set of Radio buttons except that List boxes and Combo boxes are scrollable and can contain more items than a set of Radio buttons which are limited by the screen display area. Also, unlike Radio buttons, the number of entries in a List box or Combo box can change at run-time.

The difference between a List box and a Combo box is that a List box is simply a list of items, while a Combo box is a combination of a List box and an Edit box. A List box allows the user to choose multiple selections from the list at one time, while a Combo box allows only a single selection, but a Combo box allows the user to edit the selected value while a List box only allows the user to choose from the given list.

A Drop-down list box looks like a Combo box since it has a drop-down arrow to display the list. Like a Combo box, only one selection can be made at a time in a Drop-down list box, but, like a List box, the selected value cannot be edited. A Drop-down list box serves the same function as a List box except for the disadvantage that the user can choose only a single selection, and the advantage that it takes up less dialog screen space.

Visual Fortran does not support user-drawn List boxes or user-drawn Combo boxes. You must create List boxes and Combo boxes with the Resource Editor.

The following sections describe how to use List boxes and Combo boxes:

- [Using List boxes](#)
- [Using Combo boxes](#)
- [Using Drop-down List boxes](#)

Using List Boxes

For both List boxes and Combo boxes, the control index `DLG_NUMITEMS` determines how many items are in the box. Once this value is set, you set the text of List box items by specifying a character string for each item index. Indexes run from 1 to the total number of list items set with `DLG_NUMITEMS`. For example:

```
LOGICAL retlog
retlog = DlgSet ( dlg, IDC_LISTBOX1, 3, DLG_NUMITEMS )
retlog = DlgSet ( dlg, IDC_LISTBOX1, "Moe", 1 )
retlog = DlgSet ( dlg, IDC_LISTBOX1, "Larry", 2 )
retlog = DlgSet ( dlg, IDC_LISTBOX1, "Curly", 3 )
```

These statements puts three items in the List box. The initial value of each List box entry is a blank string and the value becomes nonblank after it has been set.

You can change the list length and item values at any time, including from within callback routines. If the list is shortened, the set of entries is truncated. If the list is lengthened, blank entries are added. In the preceding example, you could extend the list length and define the new item with the following:

```
retlog = DLGSET ( dlg, IDC_LISTBOX1, 4)
retlog = DLGSET ( dlg, IDC_LISTBOX1, "Shemp", 4)
```

Since List boxes allow selection of multiple entries, you need a way to determine which entries are selected. When the user selects a List box item, it is assigned an integer index equal to the order in which the item was selected. You can test which list items are selected by reading the selection indexes in order until a zero value is read. For example, if in the previous List box the user selected Moe and then Curly, the List box selection indexes would have the following values:

Selection index	Value
1	1 (for Moe)
2	3 (for Curly)
3	0 (no more selections)

If Larry alone had been selected, the List box selection index values would be:

Selection index	Value
1	2 (for Larry)
2	0 (no more selections)

To determine the items selected, the List box values can be read with **DLGGET** until a zero is encountered. For example:

```
INTEGER j, num, test
INTEGER, ALLOCATABLE :: values(:)
LOGICAL retlog

retlog = DLGGET (dlg, IDC_LISTBOX1, num, DLG_NUMITEMS)
ALLOCATE (values(num))
j = 1
test = -1
DO WHILE (test .NE. 0)
    retlog = DLGGET (dlg, IDC_LISTBOX1, values(j), j)
```

```

    test = values(j)
    j = j + 1
END DO

```

In this example, `j` is the selection index and `values(j)` holds the list numbers, in order, of the items selected by the user, if any.

To read a single selection, or the first selected item in a set, you can use `DLG_STATE`, since for a List Box `DLG_STATE` holds the character string of the first selected item (if any). For example:

```

! Get the string for the first selected item.
retlog = DLGGET (dlg, IDC_LISTBOX1, str, DLG_STATE)

```

Alternatively, you can first retrieve the list number of the selected item, and then get the string associated with that item:

```

INTEGER value
CHARACTER(256) str
! Get the list number of the first selected item.
retlog = DLGGET (dlg, IDC_LISTBOX1, value, 1)
! Get the string for that item.
retlog = DLGGET (dlg, IDC_LISTBOX1, str, value)

```

In these examples, if no selection has been made by the user, `str` will be a blank string.

In the Properties/Styles box in the Resource Editor, List boxes can be specified as sorted or unsorted. The default is sorted, which causes List box items to be sorted alphabetically starting with A. If a List box is sorted, before each callback is called or when **DLGMODAL** returns, the items in the List box are sorted in alphabetical order.

The alphabetical sorting follows the ASCII collating sequence, and uppercase letters come before lowercase letters. For example, if the List box in the example above with the list "Moe," "Larry," "Curly," and "Shemp" were sorted, before a callback or after **DLGMODAL** returned, index 1 would refer to "Curly," index 2 to "Larry," index 3 to "Moe," and index 4 to "Shemp." For this reason, when using sorted List boxes, indexes should not be counted on to be the same before, during, and after a call to **DLGMODAL**.

Using Combo Boxes

A Combo box is a combination of a List box and an Edit box. The user can make a selection from the list that is then displayed in the Edit box part of the control, or enter text directly into the Edit box.

All dialog values a user enters are character strings, and your application must interpret these strings as the data they represent. For example, numbers entered by the user are returned to your application as character strings.

Because user input can be given in two ways, selection from the List box portion or typing into the Edit box portion directly, you need to register two callback types with **DLGSETSUB** for a Combo box. These callback types are `dlg_selchange` to handle a new list selection by the user, and `dlg_update` to handle text entered by the user directly into the Edit box portion. For example:

```

retlog = DlgSetSub( dlg, IDC_COMBO1, UpdateCombo, dlg_selchange )
retlog = DlgSetSub( dlg, IDC_COMBO1, UpdateCombo, dlg_update )

```

A Combo box list is created the same way a List box list is created, as described in the previous section, but the user can select only one item from a Combo box at a time. When the user selects an item from the list, Windows automatically puts the item into the Edit box portion of the Combo box. Thus, there is no need, and no mechanism, to retrieve the item list number of a selected item.

If the user is typing an entry directly into the Edit box part of the Combo box, again Windows automatically displays it and you do not need to. You can retrieve the character string of the selected item or Edit box entry with the following statement:

```
! Returns the character string of the selected item or Edit box entry as str.
retlog = DLGGET (dlg, IDC_COMBO1, str)
```

You have three choices for Combo box Type in the Styles tab of Combo box Properties: Simple, Drop list, and Drop-down. Simple and Drop list are the same, except that a simple Combo box always displays the Combo box choices in a list, while a drop list Combo box has a drop-down button and displays the choices in a drop-down list, conserving screen space. The Drop-down type is halfway between a Combo box and a List box and is described below.

Using Drop-Down List Boxes

To create a Drop-down list box, choose a Combo box from the control toolbar place it in your dialog. Double-click the left mouse button on the Combo box to open the Properties box. On the Styles Tab choose Drop-down as the control type.

A Drop-down list box has a drop-down arrow to display the list. Like a Combo box, only one selection can be made at a time in the list, but like a List Box, the selected value cannot be edited. A Drop-down list box serves the same function as a List box except for the disadvantage that the user can choose only a single selection, and the advantage that it takes up less dialog screen space.

A Drop-down list box has the same control indexes as a Combo box with the addition of another INTEGER index to set or return the list number of the item selected in the list. For example:

```
INTEGER num
! Returns index of the selected item.
retlog = DLGGET (dlg, IDC_DROPDOWN1, num, DLG_STATE)
```

Using Scroll Bars

With a Scroll bar, the user determines input by manipulating the slide up and down or right and left. Your application sets the range for the Scroll bar, and thus can interpret a position of the slide as a number. If you want to display this number to the user, you need to send the number (as a character string) to a Static text or Edit Box control.

The Scroll bar range always starts at 1. You set the upper limit of the range by setting the control index `DLG_RANGE` with DLGSET or DLGSETINT. The default value is 100. For example:

```
LOGICAL retlog
retlog = DLGSET (dlg, IDC_SCROLLBAR1, 212, DLG_RANGE)
```

You get the slide position by retrieving the control index `DLG_POSITION` with DLGGET or DLGGETINT. For example:

```
INTEGER slide_position
```

```
retlog = DLGGET (dlg, IDC_SCROLLBAR1, slide_position, DLG_POSITION)
```

You can also set the increment taken when the user clicks on the arrow buttons of the Scroll bar by setting the control index `DLG_SMALLSTEP`. You set the increment taken when the user clicks in the blank area above or below the slide in a vertical Scroll bar, or to the left or right of the slide in a horizontal Scroll bar, by setting the control index `DLG_BIGSTEP`. For example:

```
retlog = DLGSET (dlg, IDC_SCROLLBAR1, 4, DLG_SMALLSTEP)
retlog = DLGSET (dlg, IDC_SCROLLBAR1, 20, DLG_BIGSTEP)
```

Setting Return Values and Exiting

When the user selects the dialog's OK or CANCEL button, your dialog procedure is exited and the dialog box is closed. `DLGMODAL` returns the control name (associated with an integer identifier in your include (.FD) file) of the control that caused it to exit; for example, `IDOK` or `IDCANCEL`. If you want to exit your dialog box on a condition other than the user selecting the OK or CANCEL button, you need to include a call to the dialog subroutine `DLGEXIT` from within your callback routine. For example:

```
SUBROUTINE EXITSUB (dlg, exit_button_id, callbacktype)
USE DFLOGM
TYPE (DIALOG) dlg
INTEGER exit_button_id, callbacktype
...
  CALL DLGEXIT (dlg)
```

The only argument for `DLGEXIT` is the **dialog** derived type. The dialog box is exited after `DLGEXIT` returns control back to the dialog manager, not immediately after calling `DLGEXIT`. That is, if there are other statements following `DLGEXIT` within the callback routine that contains it, those statements are executed and the callback routine returns before the dialog box is exited.

If you want `DLGMODAL` to return with a value other than the control name of the control that caused the exit, (or -1 if `DLGMODAL` fails to open the dialog box), you can specify your own return value with the subroutine `DLGSETRETURN`. For example:

```
TYPE (DIALOG) dlg
INTEGER altreturn
...
altreturn = 485
CALL DLGSETRETURN (dlg, altreturn)
CALL DLGEXIT(dlg)
```

To avoid confusion with the default failure condition, use return values other than -1.

If you want the user to be able to close the dialog from the system menu or by pressing the ESC key, you need a control that has the ID of `IDCANCEL`. When a system escape or close is performed, it simulates pressing the dialog button with the ID `IDCANCEL`. If no control in the dialog has the ID `IDCANCEL`, then the close command will be ignored (and the dialog can not be closed in this way).

If you want to enable system close or ESC to close a dialog, but don't want a cancel button, you can add a button with the ID `IDCANCEL` to your dialog and then remove the visible property in the button's Properties box. Pressing ESC will then activate the default click callback of the cancel button and close the dialog.

Drawing Graphics Elements

The graphics routines provided with Visual Fortran set points, draw lines, draw text, change colors, and draw shapes such as circles, rectangles, and arcs. This section assumes you have read the overview in [Using QuickWin](#).

This section uses the following terms:

- The *origin* (point 0, 0) is the upper-left corner of the screen or the client area (defined user area) of the child window being written to. The x-axis and y-axis start at the origin. You can change the origin in some coordinate systems.
- The horizontal direction is represented by the *x-axis*, increasing to the right.
- The vertical direction is represented by the *y-axis*, increasing down.
- Some graphics adapters offer a *color palette* that can be changed.
- Some graphics adapters (VGA and SVGA) allow you to change the color that a color index refers to by providing a *color value* that describes a new color. The color value indicates the mix of red, green, and blue in a screen color. A color value is always an INTEGER(4) number.

The sections on drawing graphics are organized as follows:

- [Working with Graphics Modes](#)
- [Adding Color](#)
- [Understanding Coordinate Systems](#)

Working with Graphics Modes

To display graphics, you need to set the desired graphics mode using [SETWINDOWCONFIG](#), and then call the routines needed to create the graphics.

These sections explain each step:

- [Checking the Current Graphics Mode](#)
- [Setting the Graphics Mode](#)
- [Writing a Graphics Program](#)

Checking the Current Graphics Mode

Call [GETWINDOWCONFIG](#) to get the child window settings. The DFLIB.F90 module in the \DF\INCLUDE subdirectory defines a derived type, **windowconfig**, that [GETWINDOWCONFIG](#) uses as a parameter:

```

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
  CHARACTER(80) title       ! window title
  INTEGER(2) bitsperpixel   ! Number of bits per pixel
END TYPE windowconfig

```

By default, a QuickWin child window is a scrollable text window 640x480 pixels, has 30 lines and 80 columns, and a font size of 8x16. Also by default, a Standard Graphics window is Full Screen. You can change the values of window properties at any time with **SETWINDOWCONFIG**, and retrieve the current values at any time with **GETWINDOWCONFIG**.

Setting the Graphics Mode

Use **SETWINDOWCONFIG** to configure the window for the properties you want. By assigning a -1 value for *numxpixels*, *numypixels*, *numtextcols*, and *numtextrows* in the `windowconfig` derived type, you set the highest possible resolution available with your graphics driver. This causes Standard Graphics applications to start in Full Screen mode.

If you specify less than the largest graphics area, the application starts in a window. You can use ALT+ENTER to toggle between Full Screen and windowed views. If your application is a QuickWin application and you do not call **SETWINDOWCONFIG**, the child window defaults to a scrollable text window with the dimensions of 640x480 pixels, 30 lines, 80 columns, and a font size of 8x16. The number of colors depends on the video driver used.

If **SETWINDOWCONFIG** returns **.FALSE.**, the video driver does not support the options specified. The function then adjusts the values in the `windowconfig` derived type to ones that will work and are as close as possible to the requested configuration. You can then call **SETWINDOWCONFIG** again with the adjusted values, which will succeed. For example:

```
LOGICAL statusmode
TYPE (windowconfig) wc
wc.numxpixels = 1000
wc.numypixels = 300
wc.numtextcols = -1
wc.numtextrows = -1
wc.numcolors = -1
wc.title = "Opening Title"C
wc.fontsize = #000A000C ! 10 X 12
statusmode = SETWINDOWCONFIG(wc)
IF (.NOT. statusmode) THEN statusmode = SETWINDOWCONFIG(wc)
```

If you use **SETWINDOWCONFIG**, you should specify a value for each field (-1 or your own number for numeric fields, and a C string for the title). Calling **SETWINDOWCONFIG** with only some of the fields specified can result in useless values for the other fields.

Writing a Graphics Program

Like many programs, graphics programs work well when written in small units. Using discrete routines aids debugging by isolating the functional components of the program. The following example program and its associated subroutines show the steps involved in initializing, drawing, and closing a graphics program.

The SINE program draws a sine wave. Its procedures call many of the common graphics routines. The main program, following, calls five subroutines that carry out the actual graphics commands (also located in the SINE.F90 file).

```
! SINE.F90 - Illustrates basic graphics commands.
!
```

```

USE DFLIB
CALL graphicsmode( )
CALL drawlines( )
CALL sinewave( )
CALL drawshapes( )
END
.
.
.

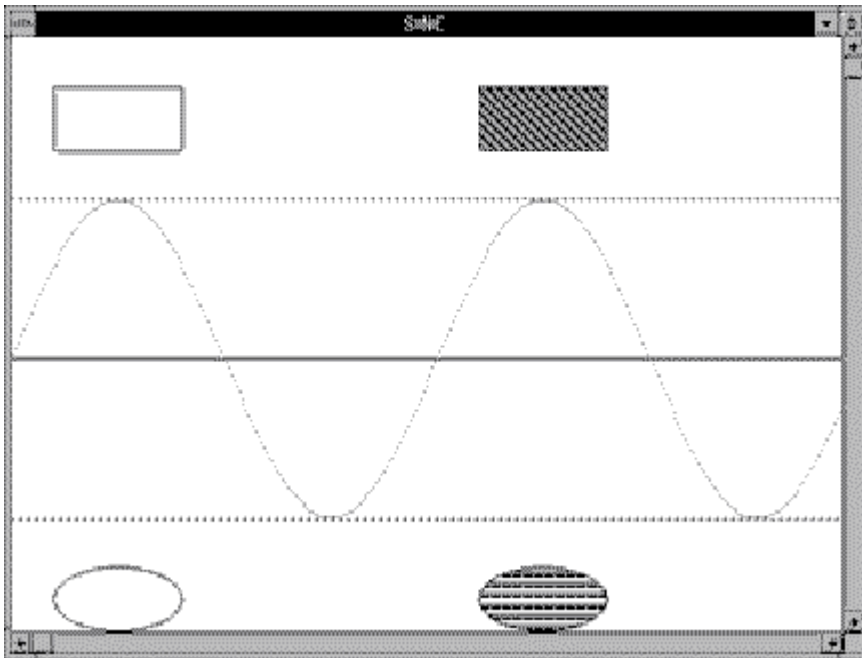
```

For information on the subroutines in the SINE program, see:

- [graphicsmode](#) in section [Activating a Graphics Mode](#)
- [drawlines](#) in section [Drawing Lines on the Screen](#)
- [sinewave](#) in section [Drawing a Sine Curve](#)
- [drawshapes](#) in section [Adding Shapes](#)

The SINE program's output appears in the following figure. The SINE routines are in the \DF\SAMPLES\TUTORIAL subdirectory. The project is built as a Standard Graphics application.

Figure: Sine Program Output



Activating a Graphics Mode

If you call a graphics routine without setting a graphics mode with [SETWINDOWCONFIG](#), QuickWin automatically sets the graphics mode with default values.

[SINE](#) selects and sets the graphics mode in the subroutine `graphicsmode`, which selects the highest possible resolution for the current video driver:

```

SUBROUTINE graphicsmode( )
  USE DFLIB
  LOGICAL          modestatus
  INTEGER(2)       maxx, maxy
  TYPE (windowconfig) myscreen

```

```

COMMON                maxx, maxy

! Set highest resolution graphics mode.

myscreen.numxpixels=-1
myscreen.numypixels=-1
myscreen.numtextcols=-1
myscreen.numtextrows=-1
myscreen.numcolors=-1
myscreen.fontsize=-1
myscreen.title = " "C ! blank

modestatus=SETWINDOWCONFIG(myscreen)

! Determine the maximum dimensions.

modestatus=GETWINDOWCONFIG(myscreen)
maxx=myscreen.numxpixels - 1
maxy=myscreen.numypixels - 1
END

```

Pixel coordinates start at zero, so, for example, a screen with a resolution of 640 horizontal pixels has a maximum x-coordinate of 639. Thus, maxx (the highest available x-pixel coordinate) must be 1 less than the total number of pixels. The same applies to maxy.

To remain independent of the video mode set by `graphicsmode`, two short functions convert an arbitrary screen size of 1000x1000 pixels to whatever video mode is in effect. From now on, the program assumes it has 1000 pixels in each direction. To draw the points on the screen, `newx` and `newy` map each point to their physical (pixel) coordinates:

```

! NEWX - This function finds new x-coordinates.

INTEGER(2) FUNCTION newx( xcoord )

INTEGER(2) xcoord, maxx, maxy
REAL(4) tempx
COMMON maxx, maxy

tempx = maxx / 1000.0
tempx = xcoord * tempx + 0.5
newx = tempx
END

! NEWY - This function finds new y-coordinates.
!
INTEGER(2) FUNCTION newy( ycoord )

INTEGER(2) ycoord, maxx, maxy
REAL(4) tempy
COMMON maxx, maxy

tempy = maxy / 1000.0
tempy = ycoord * tempy + 0.5
newy = tempy
END

```

You can set up a similar independent coordinate system with *window coordinates*, described in [Understanding Coordinate Systems](#).

Drawing Lines on the Screen

SINE next calls the subroutine `drawlines`, which draws a rectangle around the outer edges of the screen and three horizontal lines that divide the screen into quarters. (See [Figure: Sine Program Output](#).)

```
!   DRAWLINES - This subroutine draws a box and
!   several lines.

SUBROUTINE drawlines( )

USE DFLIB

EXTERNAL      newx, newy
INTEGER(2)    status, newx, newy, maxx, maxy
TYPE (xycoord) xy
COMMON        maxx, maxy

!
!   Draw the box.

status = RECTANGLE( $GBORDER, INT2(0), INT2(0), maxx, maxy )
CALL SETVIEWORG( INT2(0), newy( INT2( 500 ) ), xy ) ! This sets
!           the new origin to 0 for x and 500 for y. See comment after subroutine

!   Draw the lines.

CALL MOVETO( INT2(0), INT2(0), xy )
status = LINETO( newx( INT2( 1000 ) ), INT2(0) )
CALL SETLINESTYLE( INT2( #AA3C ) )
CALL MOVETO( INT2(0), newy( INT2( -250 ) ), xy )
status = LINETO(newx( INT2( 1000 ) ), newy( INT2( -250 ) ))
CALL SETLINESTYLE( INT2( #8888 ) )
CALL MOVETO( INT2(0), newy( INT2( 250 ) ), xy )
status = LINETO( newx( INT2( 1000 ) ), newy( INT2( 250 ) ) )
END
```

The first argument to RECTANGLE is the *fill flag*, which can be either `$GBORDER` or `$GFILLINTERIOR`. Choose `$GBORDER` if you want a rectangle of four lines (a border only, in the current line style), or `$GFILLINTERIOR` if you want a solid rectangle (filled in with the current color and fill pattern). Choosing the color and fill pattern is discussed in [Adding Color](#) and [Adding Shapes](#).

The second and third **RECTANGLE** arguments are the x- and y-coordinates of the upper-left corner of the rectangle. The fourth and fifth arguments are the coordinates for the lower-right corner. Because the coordinates for the two corners are (0, 0) and (maxx, maxy), the call to **RECTANGLE** frames the entire screen.

The program calls SETVIEWORG to change the location of the viewport origin. By resetting the origin to (0, 500) in a 1000x1000 viewport, you effectively make the viewport run from (0, -500) at the top left of the screen to (1000, 500) at the bottom right of the screen:

```
CALL SETVIEWORG( INT2(0), newy( INT2( 500 ) ), xy )
```

Changing the coordinates illustrates the ability to alter the viewport coordinates to whatever dimensions you prefer. (Viewports and the **SETVIEWORG** routine are explained in more detail in [Understanding Coordinate Systems](#).)

The call to SETLINESTYLE changes the line style from a solid line to a dashed line. A series of 16 bits tells the routine which pattern to follow. A "1" indicates a solid pixel and "0" an empty pixel.

Therefore, 1111 1111 1111 1111 represents a solid line. A dashed line might look like 1111 1111 0000 0000 (long dashes) or 1111 0000 1111 0000 (short dashes). You can choose any combination of ones and zeros. Any INTEGER(2) number in any base is an acceptable input, but binary and hexadecimal numbers are easier to envision as line-style patterns.

In the example, the hexadecimal constant #AA3C equals the binary value 1010 1010 0011 1100. You can use the decimal value 43580 just as effectively.

When drawing lines, first set an appropriate line style. Then, move to where you want the line to begin and call LINETO, passing to it the point where you want the line to end. The `drawlines` subroutine uses the following code:

```
CALL SETLINESTYLE(INT2( #AA3C ) )
CALL MOVETO( INT2(0), newy( INT2( -250 ) ), xy )
dummy = LINETO( newx( INT2( 1000 ) ), newy( INT2( -250 ) ) )
```

MOVETO positions an imaginary pixel cursor at a point on the screen (nothing appears on the screen), and LINETO draws a line. When the program called SETVIEWORG, it changed the viewport origin, and the initial y-axis range of 0 to 1000 now corresponds to a range of -500 to +500. Therefore, the negative value -250 is used as the y-coordinate of LINETO to draw a horizontal line across the center of the top half of the screen, and the value of 250 is used as the y-coordinate to draw a horizontal line across the center of the bottom half of the screen.

Drawing a Sine Curve

With the axes and frame in place, SINE is ready to draw the sine curve. The `sinewave` routine calculates the x and y positions for two cycles and plots them on the screen:

```
! SINEWAVE - This subroutine calculates and plots a sine
!           wave.
!
SUBROUTINE sinewave( )
USE DFLIB

INTEGER(2)    dummy, newx, newy, locx, locy, i
INTEGER(4)    color
REAL          rad
EXTERNAL      newx, newy

PARAMETER    ( PI = 3.14159 )

!
! Calculate each position and display it on the screen.
color = #0000FF ! red
!
DO i = 0, 999, 3
  rad = -SIN( PI * i / 250.0 )
  locx = newx( i )
  locy = newy( INT2( rad * 250.0 ) )
  dummy = SETPIXELRGB( locx, locy, color )
END DO
END
```

SETPIXELRGB takes the two location parameters, `locx` and `locy`, and sets the pixel at that position with the specified color value (red).

Adding Shapes

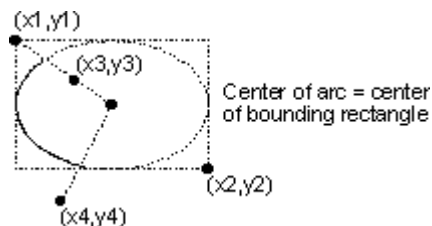
After drawing the sine curve, **SINE** calls `drawshapes` to put two rectangles and two ellipses on the screen. The fill flag alternates between `$GBORDER` and `$GFILLINTERIOR`:

```
!  DRAWSHAPES - Draws two boxes and two ellipses.
!
!      SUBROUTINE drawshapes( )
!
!      USE DFLIB
!
!      EXTERNAL    newx, newy
!      INTEGER(2)  dummy, newx, newy
!
!      Create a masking (fill) pattern.
!
!      INTEGER(1)  diagmask(8), horzmask(8)
!      DATA diagmask / #93, #C9, #64, #B2, #59, #2C, #96, #4B /
!      DATA horzmask / #FF, #00, #7F, #FE, #00, #00, #00, #CC /
!
!      Draw the rectangles.
!
!      CALL SETLINESTYLE( INT2(#FFFF) )
!      CALL SETFILLMASK( diagmask )
!      dummy = RECTANGLE( $GBORDER,newx(INT2(50)),newy(INT2(-325)), &
! & newx(INT2(200)),newy(INT2(-425)))
!      dummy = RECTANGLE( $GFILLINTERIOR,newx(INT2(550)), &
! & newy(INT2(-325)),newx(INT2(700)),newy(INT2(-425)))
!
!      Draw the ellipses.
!
!      CALL SETFILLMASK( horzmask )
!      dummy = ELLIPSE( $GBORDER,newx(INT2(50)),newy(INT2(325)), &
! & newx(INT2(200)),newy(INT2(425)))
!      dummy = ELLIPSE( $GFILLINTERIOR,newx(INT2(550)), &
! & znewy(INT2(325)),newx(INT2(700)),newy(INT2(425)))
!      END
```

The call to **SETLINESTYLE** resets the line pattern to a solid line. Omitting this routine causes the first rectangle to appear with a dashed border, because the `drawlines` subroutine called earlier changed the line style to a dashed line.

ELLIPSE draws an ellipse using parameters similar to those for **RECTANGLE**. It, too, requires a fill flag and two corners of a bounding rectangle. The following figure shows how an ellipse uses a bounding rectangle:

Figure: Bounding Rectangle



The `$GFILLINTERIOR` constant fills the shape with the current fill pattern. To create a pattern, pass the address of an 8-byte array to **SETFILLMASK**. In `drawshapes`, the `diagmask` array is initialized with the pattern shown in the following table:

Table: Fill Patterns

Bit pattern	Value in diagmask
Bit No. 7 6 5 4 3 2 1 0	
x o o x o o x x	diagmask(1) = #93
x x o o x o o x	diagmask(2) = #C9
o x x o o x o o	diagmask(3) = #64
x o x x o o x o	diagmask(4) = #B2
o x o x x o o x	diagmask(5) = #59
o o x o x x o o	diagmask(6) = #2C
x o o x o x x o	diagmask(7) = #96
o x o o x o x x	diagmask(8) = #4B

Adding Color

The Visual Fortran QuickWin Library supports color graphics. The number of total available colors depends on the current video driver and video adapter you are using. The number of available colors you use depends on the graphics functions you choose. The different color modes and color functions are discussed and demonstrated in the following sections:

- [Color Mixing](#)
- [VGA Color Palette](#)
- [Using Text Colors](#)

Color Mixing

If you have a VGA machine, you are restricted to displaying at most 256 colors at a time. These 256 colors are held in a palette. You can choose the palette colors from a range of 262,144 colors (256K), but only 256 at a time. Some display adapters (most SVGAs) are capable of displaying all of the 256K colors and some (true color display adapters) of displaying $256 * 256 * 256 = 16.7$ million colors.

If you use a palette, you are restricted to the colors available in the palette. In order to access all colors available on your system, you need to specify an explicit Red-Green-Blue (RGB) value, not a palette index.

When you select a color index, you specify one of the colors in the system's predefined palette. SETCOLOR, SETBKCOLOR, and SETTEXTCOLOR set the current color, background color, and text color to a palette index. SETCOLORRRGB, SETBKCOLORRRGB, and SETTEXTCOLORRRGB set the colors to a color value chosen from the entire available range. When you select a color value, you specify a level of intensity with a range of 0 - 255 for each of the red, green, and blue color values. The long integer that defines a color value consists of 3 bytes (24 bits) as follows:

```

MSB                                     LSB
BBBBBBBB GGGGGGGG RRRRRRRR
    
```

where R, G, and B represent the bit values for red, green, and blue intensities. To mix a light red (pink), turn red all the way up and mix in some green and blue:

```
10000000 10000000 11111111
```

In hexadecimal notation, this number equals #8080FF . You can use the function:


```
i = SETCOLORRGB (#8080FF)
```

to set the current color to this value.

You can also pass decimal values to this function. Keep in mind that 1 (binary 00000001, hex 01) represents a low color intensity and that 255 (binary 11111111, hex FF) equals full color intensity. To create pure yellow (100-percent red plus 100-percent green) use this line:

```
i = SETCOLORRGB( #00FFFF )
```

For white, turn all of the colors on:

```
i = SETCOLORRGB( #FFFFFF )
```

For black, set all of the colors to 0:

```
i = SETCOLORRGB( #000000 )
```

RGB values for example colors are in the following table.

Table: RGB Color Values			
Color	RGB Value	Color	RGB Value
Black	#000000	Bright White	#FFFFFF
Dull Red	#000080	Bright Red	#0000FF
Dull Green	#008000	Bright Green	#00FF00
Dull Yellow	#008080	Bright Yellow	#00FFFF
Dull Blue	#800000	Bright Blue	#FF0000
Dull Magenta	#800080	Bright Magenta	#FF00FF
Dull Turquoise	#808000	Bright Turquoise	#FFFF00
Dark Gray	#808080	Light Gray	#C0C0C0

If you have a 64K-color machine and you set an RGB color value that is not equal to one of the 64K preset RGB color values, the system approximates the requested RGB color to the closest available RGB value. The same thing happens on a VGA machine when you set an RGB color that is not in the palette. (You can remap your VGA color palette to different RGB values. See [VGA Color Palette](#).)

However, although your graphics are drawn with an approximated color, if you retrieve the color with [GETCOLORRGB](#), [GETBKCOLORRGB](#), or [GETTEXTCOLORRGB](#), the color you specified is returned, not the actual color used. This is because the **SETCOLORRGB** functions do not execute any graphics, they simply set the color and the approximation is made when the drawing is made (by **ELLIPSE** or **ARC**, for example). [GETPIXELRGB](#) and [GETPIXELSRGB](#) do return the approximated color actually used, because [SETPIXELRGB](#) and [SETPIXELSRGB](#) actually set a pixel to a color on the screen and the approximation, if any, is made at the time they are called.

VGA Color Palette

A VGA machine is capable of displaying at most 256 colors at a time. In QuickWin, VGA can display 256 colors with a resolution of 320x200 pixels, but can display 2 or 16 colors at higher resolutions (up to 640x480 pixels). The number of colors you select for your VGA palette depends

on your application, and is set by setting the *wc.numcolors* variable in the `windowconfig` derived type to 2, 16, or 256 with SETWINDOWCONFIG.

An RGB color value must be in the palette to be accessible to your VGA graphic displays. You can change the default colors and customize your color palette by using REMAPPALLETTERGB to change a palette color index to any RGB color value. The following example remaps the color index 1 (default blue color) to the pure red color value given by the RGB value #0000FF. After this is executed, whatever was displayed as blue will appear as red:

```
USE DFLIB
INTEGER(4) status
status = REMAPPALLETTERGB( 1, #0000FF )    ! Reassign color index 1
                                           ! to RGB red
```

REMAPALLPALETTERGB remaps one or more color indexes simultaneously. Its argument is an array of RGB color values that are mapped into the palette. The first color number in the array becomes the new color associated with color index 0, the second with color index 1, and so on. At most 236 indexes can be mapped, because 20 indexes are reserved for system use.

If you request an RGB color that is not in the palette, the color selected from the palette is the closest approximation to the RGB color requested. If the RGB color was previously placed in the palette with REMAPPALLETTERGB or **REMAPALLPALETTERGB**, then that exact RGB color is available.

Remapping the palette has no effect on 64K-color machines, SVGA, or true-color machines, unless you limit yourself to a palette by using color index functions such as SETCOLOR. On a VGA machine, if you remap all the colors in your palette and display that palette in graphics, you cannot then remap and simultaneously display a second palette.

For instance, in VGA 256-color mode, if you remap all 256 palette colors and display graphics in one child window, then open another child window, remap the palette and display graphics in the second child window, you are attempting to display more than 256 colors at one time. The machine cannot do this, so whichever child window has the focus will appear correct, while the one without the focus will change color.

Note: Machines that support more than 256 colors will not be able to do animation by remapping the palette. Windows 95 and Windows NT create a logical palette that maps to the video hardware palette. On video hardware that supports a palette of 256 colors or less, remapping the palette maps over the current palette and redraws the screen in the new colors.

On large hardware palettes that support more than 256 colors, remapping is done into the unused portion of the palette. It does not map over the current colors nor redraw the screen. So, on machines with large palettes (more than 256 colors), the technique of changing the screen through remapping, called palette animation, cannot be used. See the *Win32 SDK Manual* for more information.

Symbolic constants (names) for the default color numbers are supplied in the graphics modules. The names are self-descriptive; for example, the color numbers for black, yellow, and red are represented by the symbolic constants \$BLACK, \$YELLOW, and \$RED.

All of the VGA display modes operate with any VGA (analog) monitor. Colors appear as shades of

gray on compatible analog monochrome monitors.

Using Text Colors

SETTEXTCOLORRGB (or SETTEXTCOLOR) and SETBKCOLORRGB (or SETBKCOLOR) set the foreground and background colors for text output. All use a single argument specifying the color value (or color index) for text displayed with OUTTEXT and WRITE. For the color index functions, colors are represented by the range 0-31. Index values in the range of 16-31 access the same colors as those in the range of 0-15.

You can retrieve the current foreground and background color values with GETTEXTCOLORRGB and GETBKCOLORRGB or the color indexes with GETTEXTCOLOR and GETBKCOLOR. Use SETTEXTPOSITION to move the cursor to a particular row and column. **OUTTEXT** and **WRITE** print the text at the current cursor location.

For more information on these routines, see the *Reference*.

Understanding Coordinate Systems

Several different coordinate systems are supported by the Visual Fortran QuickWin Library. Text coordinates work in rows and columns; physical coordinates serve as an absolute reference and as a starting place for creating custom window and viewport coordinates. Conversion routines make it simple to convert between different coordinate systems.

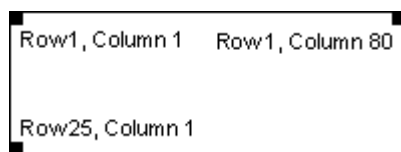
The coordinate systems are demonstrated and discussed in the following sections:

- Text Coordinates
- Graphics Coordinates
- Real Coordinates Sample Program

Text Coordinates

The text modes use a coordinate system that divides the screen into rows and columns as shown in the following figure:

Figure: Text Screen Coordinates



Text coordinates use the following conventions:

- Numbering starts at 1. An 80-column screen contains columns 1-80.
- The row is always listed before the column.

If the screen displays 25 rows and 80 columns (as in the above Figure 14.3), the rows are numbered 1-25 and the columns are numbered 1-80. The text-positioning routines, such as SETTEXTPOSITION and SCROLLTEXTWINDOW, use row and column coordinates.

Graphics Coordinates

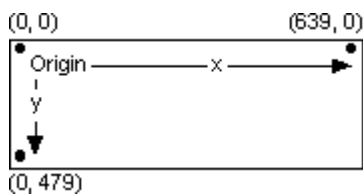
Three coordinate systems describe the location of pixels on the screen: physical coordinates, viewport coordinates, and window coordinates. In all three coordinate systems, the x-coordinate is listed before the y-coordinate.

Physical Coordinates

Physical coordinates are integers that refer to pixels in a window's client area. By default, numbering starts at 0, not 1. If there are 640 pixels, they are numbered 0-639.

Suppose your program calls SETWINDOWCONFIG to set up a client area containing 640 horizontal pixels and 480 vertical pixels. Each individual pixel is referred to by its location relative to the x-axis and y-axis, as shown in the following figure:

Figure: Physical Coordinates



The upper-left corner is the *origin*. The x- and y-coordinates for the origin are always (0, 0).

Physical coordinates refer to each pixel directly and are therefore integers (that is, the window's client area cannot display a fractional pixel). If you use variables to refer to pixel locations, declare them as integers or use type-conversion routines when passing them to graphics functions. For example:

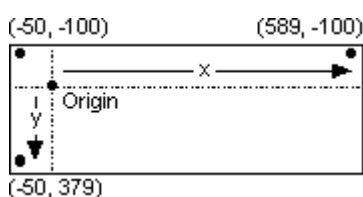
```
ISTATUS = LINETO( INT2(REAL_x), INT2(REAL_y))
```

If a program uses the default dimension of a window, the *viewport* (drawing area) is equal to 640x480. SETVIEWORG changes the location of the viewport's origin. You pass it two integers, which represent the x and y physical screen coordinates for the new origin. You also pass it an *xycoord* type that the routine fills with the physical coordinates of the previous origin. For example, the following line moves the viewport origin to the physical screen location (50, 100):

```
TYPE (xycoord) origin
CALL SETVIEWORG(INT2(50), INT2(100), origin)
```

The effect on the screen is illustrated in the following figure:

Figure: Origin Coordinates Changed by SETVIEWORG



The number of pixels hasn't changed, but the coordinates used to refer to the points have changed. The x-axis now ranges from -50 to +589 instead of 0 to 639. The y-axis now covers the values -100 to +379.

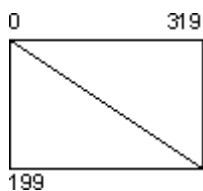
All graphics routines that use viewport coordinates are affected by the new origin, including **MOVETO**, **LINETO**, **RECTANGLE**, **ELLIPSE**, **POLYGON**, **ARC**, and **PIE**. For example, if you call **RECTANGLE** after relocating the viewport origin and pass it the values (0, 0) and (40, 40), the upper-left corner of the rectangle would appear 50 pixels from the left edge of the screen and 100 pixels from the top. It would not appear in the upper-left corner of the screen.

SETCLIPRGN creates an invisible rectangular area on the screen called a *clipping region*. You can draw inside the clipping region, but attempts to draw outside the region fail (nothing appears outside the clipping region).

The default clipping region occupies the entire screen. The QuickWin Library ignores any attempts to draw outside the screen.

You can change the clipping region by calling **SETCLIPRGN**. For example, suppose you entered a screen resolution of 320x200 pixels. If you draw a diagonal line from (0, 0) to (319, 199), the upper-left to the lower-right corner, the screen looks like the following figure:

Figure: Line Drawn on a Full Screen

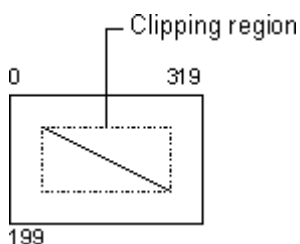


You could create a clipping region by entering:

```
CALL SETCLIPRGN(INT2(10), INT2(10), INT2(309), INT2(189))
```

With the clipping region in effect, the same **LINETO** command would put the line shown in the following figure on the screen. The dashed lines indicate the outer bounds of the clipping region and do not actually print on the screen.

Figure: Line Drawn Within a Clipping Region



Viewport Coordinates

The viewport is the area of the screen displayed, which may be only a portion of the window's client area. Viewport coordinates represent the pixels within the current viewport. **SETVIEWPORT** establishes a new viewport within the boundaries of the physical client area. A standard viewport has two distinguishing features:

- The origin of a viewport is in the upper-left corner.
- The default clipping region matches the outer boundaries of the viewport.

SETVIEWPORT has the same effect as **SETVIEWORG** and **SETCLIPRGN** combined. It specifies a limited area of the screen in the same manner as **SETCLIPRGN**, then sets the viewport origin to the upper-left corner of the area.

Window Coordinates

Functions that refer to coordinates on the client-area screen and within the viewport require integer values. However, many applications need floating-point values--for frequency, viscosity, mass, and so on. **SETWINDOW** allows you to scale the screen to almost any size. In addition, window-related functions accept double-precision values.

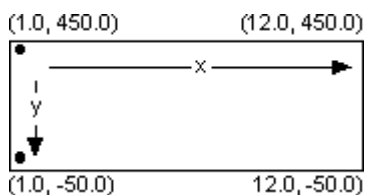
Window coordinates use the current viewport as their boundaries. A window overlays the current viewport. Graphics drawn at window coordinates beyond the boundaries of the window--the same as being outside the viewport--are clipped.

For example, to graph 12 months of average temperatures on the planet Venus that range from -50 to +450, add the following line to your program:

```
status = SETWINDOW(.TRUE., 1.0D0, -50.0D0, 12.0D0, 450.0D0)
```

The first argument is the invert flag, which puts the lowest y value in the lower-left corner. The minimum and maximum x- and y-coordinates follow; the decimal point marks them as floating-point values. The new organization of the screen is shown in the following figure:

Figure: Window Coordinates



January and December plot on the left and right edges of the screen. In an application like this, numbering the x-axis from 0.0 to 13.0 provides some padding space on the sides and improves appearance.

If you next plot a point with **SETPIXEL_W** or draw a line with **LINETO_W**, the values are automatically scaled to the established window.

► To use window coordinates with floating-point values:

1. Set a graphics mode with **SETWINDOWCONFIG**.
2. Use **SETVIEWPORT** to create a viewport area. This step is not necessary if you plan to use the entire screen.
3. Create a real-coordinate window with **SETWINDOW**, passing a **LOGICAL** invert flag and four **DOUBLE PRECISION** x- and y-coordinates for the minimum and maximum values.
4. Draw graphics shapes with **RECTANGLE_W** and similar routines. Do not confuse **RECTANGLE** (the viewport routine) with **RECTANGLE_W** (the window routine for

drawing rectangles). All window function names end with an underscore and the letter **W** (**_W**).

Real-coordinate graphics give you flexibility and device independence. For example, you can fit an axis into a small range (such as 151.25 to 151.45) or into a large range (-50000.0 to +80000.0), depending on the type of data you graph. In addition, by changing the window coordinates, you can create the effects of zooming in or panning across a figure. The window coordinates also make your drawings independent of the computer's hardware. Output to the viewport is independent of the actual screen resolution.

Real Coordinates Sample Program

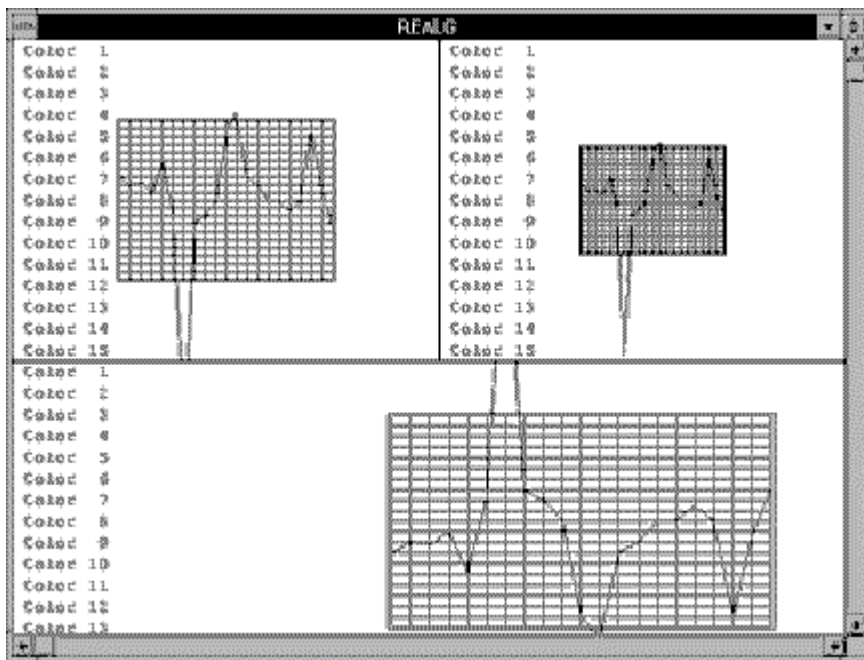
The program REALG.F90 shows how to create multiple window-coordinate sets, each in a separate viewport, on a single screen. REALG.F90 is in the \DF\SAMPLES\TUTORIAL subdirectory.

```
! REALG.F90 (main program) - Illustrates coordinate graphics.
!
  USE DFLIB
  LOGICAL          statusmode
  TYPE (windowconfig) myscreen
  COMMON          myscreen
!
! Set the screen to the best resolution and maximum number of
! available colors.
myscreen.numxpixels = -1
myscreen.numypixels = -1
myscreen.numtextcols = -1
myscreen.numtextrows = -1
myscreen.numcolors = -1
myscreen.fontsize = -1
myscreen.title = " "C
statusmode = SETWINDOWCONFIG(myscreen)
IF(.NOT. statusmode) statusmode = SETWINDOWCONFIG(myscreen)

statusmode = GETWINDOWCONFIG( myscreen )
CALL threegraphs( )
END
.
.
.
```

The main body of the program is very short. It sets the window to the best resolution of the graphics driver (by setting the first four fields to -1) and the maximum number of colors (by setting `numcolors` to -1). The program then calls the [threegraphs](#) subroutine that draws three graphs. The program output is shown in the following figure:

Figure: REALG Program Output



The `gridshape` subroutine, which draws the graphs, uses the same data in each case. However, the program uses three different coordinate windows. The two viewports in the top half are the same size in physical coordinates, but have different window sizes. Each window uses different maximum and minimum values. In all three cases, the graph area is two units wide. The window in the upper-left corner has a range in the x-axis of four units (4 units wide); the window in the upper-right corner has a range in the x-axis of six units, which makes the graph on the right appear smaller.

In two of the three graphs, one of the lines goes off the edge, outside the clipping region. The lines do not intrude into the other viewports, because defining a viewport creates a clipping region.

Finally, the graph on the bottom inverts the data with respect to the two graphs above it.

The next section describes and discusses the subroutine invoked by `REALG.F90`:

Drawing the Graphs

The main program calls `threegraphs`, which prints the three graphs.

```

SUBROUTINE threegraphs()
  USE DFLIB
  INTEGER(2)      status, halfx, halfy
  INTEGER(2)      xwidth, yheight, cols, rows
  TYPE (windowconfig) myscreen
  COMMON          myscreen

  CALL CLEARSCREEN( $GCLEARSCREEN )
  xwidth  = myscreen.numxpixels
  yheight = myscreen.numypixels
  cols    = myscreen.numtextcols
  rows    = myscreen.numtextrows
  halfx   = xwidth / 2
  halfy   = (yheight / rows) * ( rows / 2 )
  !
  ! First window
  !
  CALL SETVIEWPORT( INT2(0), INT2(0), halfx - 1, halfy - 1 )

```



```

CALL SETTEXTWINDOW( INT2(1), INT2(1), rows / 2, cols / 2 )
status = SETWINDOW( .FALSE., -2.0_8, -2.0_8, 2.0_8, 2.0_8)
! The 2.0_8 notation makes these constants REAL(8)

CALL gridshape( rows / 2 )
status = RECTANGLE( $GBORDER,INT2(0),INT2(0),halfx-1,halfy-1)
!
! Second window
!
CALL SETVIEWPORT( halfx, INT2(0), xwidth - 1, halfy - 1 )
CALL SETTEXTWINDOW( INT2(1), (cols/2) + 1, rows/2, cols)
status = SETWINDOW( .FALSE., -3.0D0, -3.0D0, 3.0D0, 3.0D0)
! The 3.0D0 notation makes these constants REAL(8)

CALL gridshape( rows / 2 )
status = RECTANGLE_W( $GBORDER, -3.0_8,-3.0_8,3.0_8, 3.0_8)

!
! Third window
!
CALL SETVIEWPORT( 0, halfy, xwidth - 1, yheight - 1 )
CALL SETTEXTWINDOW( (rows / 2) + 1, 1_2, rows, cols )
status = SETWINDOW( .TRUE., -3.0_8, -1.5_8, 1.5_8, 1.5_8)
CALL gridshape( INT2( (rows / 2) + MOD( rows, INT2(2))))
status = RECTANGLE_W( $GBORDER, -3.0_8, -1.5_8, 1.5_8, 1.5_8)
END

```

Although the screen is initially clear, `threegraps` makes sure by calling the CLEARSCREEN routine to clear the window:

```
CALL CLEARSCREEN( $GCLEARSCREEN )
```

The `$GCLEARSCREEN` constant clears the entire window. Other options include `$GVIEWPORT` and `$GWINDOW`, which clear the current viewport and the current text window, respectively.

After assigning values to some variables, `threegraps` creates the first window:

```

CALL SETVIEWPORT( INT2(0), INT2(0), halfx - 1, halfy - 1)
CALL SETTEXTWINDOW( INT2(1), INT2(1), rows / 2, cols / 2)
status = SETWINDOW( .FALSE., -2.0_8, -2.0_8, 2.0_8, 2.0_8)

```

The first instruction defines a viewport that covers the upper-left quarter of the screen. The next instruction defines a text window within the boundaries of that border. Finally, the third instruction creates a window with both x and y values ranging from -2.0 to 2.0. The **.FALSE.** constant causes the y-axis to increase from top to bottom, which is the default. The `_8` notation identifies the constants as `REAL(8)`.

Next, the function `gridshape` inserts the grid and plots the data, and a border is added to the window:

```

CALL gridshape( rows / 2 )
status = RECTANGLE( $GBORDER,INT2(0),INT2(0),halfx-1,halfy-1)

```

This is the standard **RECTANGLE** routine, which takes coordinates relative to the viewport, not the window.

The `gridshape` subroutine plots the data on the screen.

```

! GRIDSHAPE - This subroutine plots data for REALG.F90
!
!   SUBROUTINE gridshape( numc )
!
!   USE DFLIB
!   INTEGER(2)          numc, i, status
!   INTEGER(4)          rgbcolor, oldcolor
!   CHARACTER(8)       str
!   REAL(8)             bananas(21), x
!   TYPE (windowconfig) myscreen
!   TYPE (wxycoord)     wxy
!   TYPE (rccoord)      curpos
!   COMMON              myscreen
!
!   Data for the graph:
!
!   DATA bananas / -0.3, -0.2, -0.224, -0.1, -0.5, 0.21, 2.9, &
& 0.3, 0.2, 0.0, -0.885, -1.1, -0.3, -0.2, &
& 0.001, 0.005, 0.14, 0.0, -0.9, -0.13, 0.31 /
!
!   Print colored words on the screen.
!
!   IF(myscreen.numcolors .LT. numc) numc = myscreen.numcolors-1
!   DO i = 1, numc
!     CALL SETTEXTPOSITION( i, INT2(2), curpos )
!     rgbcolor = 12**i -1
!     rgbcolor = MODULO(rgbcolor, #FFFFFF)
!     oldcolor = SETTEXTCOLORRGB( rgbcolor )
!     WRITE ( str, '(I8)' ) rgbcolor
!     CALL OUTTEXT( 'Color ' // str )
!   END DO
!
!   Draw a double rectangle around the graph.
!
!   oldcolor = SETCOLORRGB( #0000FF ) ! full red
!   status = RECTANGLE_W( $GBORDER, -1.00_8, -1.00_8, 1.00_8,1.00_8)
! constants made REAL(8) by appending _8
!   status = RECTANGLE_W( $GBORDER, -1.02_8, -1.02_8, 1.02_8, 1.02_8)
!
!   Plot the points.
!
!   x = -0.90
!   DO i = 1, 19
!     oldcolor = SETCOLORRGB( #00FF00 ) ! full green
!     CALL MOVETO_W( x, -1.0_8, wxy )
!     status = LINETO_W( x, 1.0_8 )
!     CALL MOVETO_W( -1.0_8, x, wxy )
!     status = LINETO_W( 1.0_8, x )
!     oldcolor = SETCOLORRGB( #FF0000 ) ! full blue
!     CALL MOVETO_W( x - 0.1_8, bananas( i ), wxy )
!     status = LINETO_W( x, bananas( i + 1 ) )
!     x = x + 0.1
!   END DO
!
!   CALL MOVETO_W( 0.9_8, bananas( i ), wxy )
!   status = LINETO_W( 1.0_8, bananas( i + 1 ) )
!   oldcolor = SETCOLORRGB( #00FFFF ) ! yellow
!   END

```

The routine names that end with **_W** work in the same way as their viewport equivalents, except that you pass double-precision floating-point values instead of integers. For example, you pass **INTEGER(2)** to **LINETO**, but **REAL(8)** values to **LINETO_W**.

The two other windows are similar to the first. All three call the `gridshape` function, which draws a grid from location $(-1.0, -1.0)$ to $(1.0, 1.0)$. The grid appears in different sizes because the coordinates in the windows vary. The second window ranges from $(-3.0, -3.0)$ to $(3.0, 3.0)$, and the third from $(-3.0, -1.5)$ to $(1.5, 1.5)$, so the sizes change accordingly.

The third window also contains a **.TRUE.** inversion argument. This causes the y-axis to increase from bottom to top, instead of top to bottom. As a result, this graph appears upside down with respect to the other two.

After calling `gridshape`, the program frames each window, using a statement such as the following:

```
status = RECTANGLE_W( $GBORDER, -3.0_8, -1.5_8, 1.5_8, 1.5_8)
```

The first argument is a fill flag indicating whether to fill the rectangle's interior or just to draw its outline. The remaining arguments are the x and y coordinates for the upper-left corner followed by the x and y coordinates for the lower-right corner. **RECTANGLE** takes integer arguments that refer to the viewport coordinates. **RECTANGLE_W** takes four double-precision floating-point values referring to window coordinates.

After you create various graphics elements, you can use the font-oriented routines to polish the appearance of titles, headings, comments, or labels. [Using Fonts from the Graphics Library](#) describes in more detail how to print text in various fonts with font routines.

Using Fonts from the Graphics Library

The Visual Fortran Graphics Library includes routines that print text in various sizes and type styles. These routines provide control over the appearance of your text and add visual interest to your screen displays.

This section assumes you have read [Drawing Graphics Elements](#) and that you understand the general terminology it introduces. You should also be familiar with the basic properties of both the [SETWINDOWCONFIG](#) and [MOVETO](#) routines. Also, remember that graphics programs containing graphics routines must be built as QuickWin or Standard Graphics applications.

The project type is set in Developer Studio when you select New from the File menu, then click on the Projects tab, and select either QuickWin Application or Standard Graphics Application from the application types listed. Graphics applications can also be built with the [/libs:qwin](#) or [/libs:qwins](#) compiler option.

Font types and the use of fonts are described in the following sections:

- [Available Typefaces](#)
- [Using Fonts](#)
- [SHOWFONT.F90 Example](#)

Available Typefaces

A *font* is a set of text characters of a particular size and style. A *typeface* (or *type style*) refers to the style of the displayed text --Arial, for example, or Times New Roman.

Type size measures the screen area occupied by individual characters. The term comes from the printer's lexicon, but uses screen pixels as the unit of measure rather than the traditional points. For example, "Courier 12 9" denotes the Courier typeface, with each character occupying a screen area of 12 vertical pixels by 9 horizontal pixels. The word "font", therefore implies both a typeface and a type size.

The QuickWin Library's font routines use all Windows operating system installed fonts. The first type of font used is a *bitmap* (or *raster-map*) font. Bitmap fonts have each character in a binary data map. Each bit in the map corresponds to a screen pixel. If the bit equals 1, its associated pixel is set to the current screen color. Bit values of 0 appear in the current background color.

The second type of font is called a TrueType font. Some screen fonts look different on a printer, but TrueType fonts print exactly as they appear on the screen. TrueType fonts may be bitmaps or soft fonts (fonts that are downloaded to your printer before printing), depending on the capabilities of your printer. TrueType fonts are scalable and can be sized to any height. It is recommended that you use TrueType fonts in your graphics programs.

Each type of font has advantages and disadvantages. Bitmapped characters appear smoother on the screen because of the predetermined pixel mapping. However, they cannot be scaled. You can scale TrueType text to any size, but the characters sometimes don't look quite as solid as the bitmapped characters on the screen. Usually this screen effect is hardly noticeable, and when printed, TrueType fonts are as smooth or smoother than bitmapped fonts.

The bitmapped typefaces come in preset sizes measured in pixels. The exact size of any font depends on screen resolution and display type.

Using Fonts

QuickWin's font routines can use all the Windows operating system installed fonts. To use fonts in your program, you must:

1. Initialize the fonts.
2. Select a current font from the initialized fonts.
3. Display font text with **OUTGTEXT**

Initializing Fonts

A program that uses fonts must first organize the fonts into a list in memory, a process called initializing. The list gives the computer information about the available fonts.

Initialize the fonts by calling the INITIALIZEFONTS routine:

```
USE DFLIB
INTEGER(2) numfonts
numfonts = INITIALIZEFONTS( )
```

If the computer successfully initializes one or more fonts, **INITIALIZEFONTS** returns the number of fonts initialized. If the function fails, it returns a negative error code.

Setting the Font and Displaying Text

Before a program can display text in a particular font, it must know which of the initialized fonts to use. SETFONT makes one of the initialized fonts the current (or "active") font. **SETFONT** has the following syntax:

SETFONT(*options*)

The function's argument consists of letter codes that describe the desired font: typeface, character height and width in pixels, fixed or proportional, and attributes such as bold or italic. These options are discussed in detail in the **SETFONT** entry in the *Reference*. For example:

```
USE DFLIB
INTEGER(2) index, numfonts
numfonts = INITIALIZEFONTS ( )
index = SETFONT('t' 'Cottage' 'h18w10')
```

This sets the typeface to Cottage, the character height to 18 pixels and the width to 10 pixels.

The following example sets the typeface to Arial, the character height to 14, with proportional spacing and italics (the pi codes):

```
index = SETFONT('t' 'Arial' 'h14pi')
```

If **SETFONT** successfully sets the font, it returns the font's index number. If the function fails, it returns a negative integer. Call **GRSTATUS** to find the source of the problem; its return value indicates why the function failed. If you call **SETFONT** before initializing fonts, a run-time error occurs.

SETFONT updates the font information when it is used to select a font. **GETFONTINFO** can be used to obtain information about the currently selected font. **SETFONT** sets the user fields in the `fontinfo` type (a derived type defined in `DFLIB.MOD`), and **GETFONTINFO** returns the user-selected values. The following user fields are contained in `fontinfo`:

```
TYPE fontinfo
  INTEGER(2) type ! 1 = truetype, 0 = bit map
  INTEGER(2) ascent ! Pixel distance from top to baseline
  INTEGER(2) pixwidth ! Character width in pixels, 0=prop
  INTEGER(2) pixheight ! Character height in pixels
  INTEGER(2) avgwidth ! Average character width in pixels
  CHARACTER(32) facename ! Font name
END TYPE fontinfo
```

To find the parameters of the current font, call **GETFONTINFO**. For example:

```
USE DFLIB
TYPE (fontinfo) font
INTEGER(2) i, numfonts
numfonts = INITIALIZEFONTS()
i = SETFONT ( ' t ' 'Arial ' )
i = GETFONTINFO(font)
WRITE (*,*) font.avgwidth, font.pixheight, font.pixwidth
```

After you initialize the fonts and make one the active font, you can display the text on the screen.

▶ To display text on the screen after selecting a font

1. Select a starting position for the text with **MOVETO**.
2. Optionally, set a text display angle with **SETGTEXTROTATION**.
3. Send the text to the screen (in the current font) with **OUTGTEXT**.

MOVETO moves the current graphics point to the pixel coordinates passed to it when it is invoked. This becomes the starting position of the upper-left corner of the first character in the text.

SETGTEXTROTATION can set the text's orientation in one-degree increments.

SHOWFONT.F90 Example

The program `SHOWFONT.F90` in the `\DF\SAMPLES\TUTORIAL` subdirectory displays text in the fonts available on your system. (Once the screen fills with text, press `ENTER` to display the next screen.) An abbreviated version follows. **SHOWFONT** calls **SETFONT** to specify the typeface. **MOVETO** then establishes the starting point for each text string. The program sends a message of sample text to the screen for each font initialized:

```
! Abbreviated version of SHOWFONT.F90.
USE DFLIB

INTEGER(2) grstat, numfonts,indx, curr_height
TYPE (xycoord) xyt
TYPE (fontinfo) f
CHARACTER(6) str      ! 5 chars for font num
```

```

! (max. is 32767), 1 for 'n'

! Initialization.
  numfonts=INITIALIZEFONTS( )
  IF (numfonts.LE.0) PRINT *, "INITIALIZEFONTS error"
  IF (GRSTATUS().NE.$GROK) PRINT *, 'INITIALIZEFONTS GRSTATUS error.'
  CALL MOVETO (0,0,xyt)
  grstat=SETCOLORRGB(#FF0000)
  grstat=FLOODFILLRGB(0, 0, #00FF00)
  grstat=SETCOLORRGB(0)
! Get default font height for comparison later.
  grstat = SETFONT('n1')
  grstat = GETFONTINFO(f)
  curr_height = f.pixheight
! Done initializing, start displaying fonts.
  DO indx=1,numfonts
    WRITE(str,10)indx
    grstat=SETFONT(str)
    IF (grstat.LT.1) THEN
      CALL OUTGTEXT('SetFont error.')
    ELSE
      grstat=GETFONTINFO(f)
      grstat=SETFONT('n1')
      CALL OUTGTEXT(f.facename(:len_trim(f.facename)))
      CALL OUTGTEXT(' ')
! Display font.
      grstat=SETFONT(str)
      CALL OUTGTEXT('ABCDEFGGabcdefg12345!@#$$%')
    END IF
! Go to next line.
    IF (f.pixheight .GT. curr_height) curr_height=f.pixheight
    CALL GETCURRENTPOSITION(xyt)
    CALL MOVETO(0,INT2(xyt.ycoord+curr_height),xyt)
  END DO
10  FORMAT ('n',I5.5)
END

```

Writing New Code: Design Considerations

Before you can start to write new programs or port existing ones to Visual Fortran, you must decide what to build and how to build it. This section covers the following topics:

- [Choosing Your Development Environment](#) with Visual Fortran.
- [Selecting a Program Type](#) that you can build.
- [Structuring Your Program](#).
- [Special Design Considerations](#).
- [Using the Special Features of Microsoft Windows](#) with your programs.
- [Development Environments](#), which provides a graphic overview of the development process.

Choosing Your Development Environment

With Visual Fortran, you can build programs either from a console window (which allows you to enter text commands directly into a command prompt) or from Microsoft Developer Studio, the integrated development environment. For information on using Microsoft Developer Studio, see the [Developer Studio Environment User's Guide](#).

Developer Studio offers a number of ways to simplify the task of compiling and linking programs. For example, a dialog box presents compiler and linker options in logical groupings, with descriptive names and simple mouse or keyboard selection methods. (If you need assistance using this or any other dialog box, choose the Help button in the dialog box.)

Developer Studio also provides a default editor, which is integrated with Help, the debugger, and error tracking features. Developer Studio allows many types of editors, and is customizable. For example, Emacs is a preset type of editor, which is integrated with both Help and the debugger. You can use your favorite ASCII text editor within Developer Studio. If you do, however, you may not be able to use the integrated Help, debugger, and error tracking features.

Because software development is an iterative process, it is important to be able to move quickly and efficiently to various locations in your source code. If you use Developer Studio to compile and link your programs, you can call up both the description of the error message and the relevant source code directly from the error messages in the output window.

You also use the Developer Studio editor to view and control execution of your program with the integrated source level debugger. Finally, when you use the project browser to locate routines, data elements, and references to them, Developer Studio uses its editor to go directly to the source code.

When you build programs from the console, you are in complete control of the build tools. If you choose to, you can customize how your program is built by your selection of compiler and linker options. Compiler and linker options are described in [Compiler and Linker Options](#).

Even if you choose to edit and build your program from the command console, you can still use the Developer Studio debugger and browser after your program has compiled and linked cleanly. Finally, you can run the profiler to produce a text report of your program's execution statistics either from the command console or from Developer Studio.

Selecting a Program Type

You select the program type in the Project Type dialog box when you create a new project file. You can build four basic kinds of executable programs (as well as DLLs and static libraries):

- [Console applications \(.EXE\)](#)
- [Standard graphics applications \(.EXE\)](#)
- [QuickWin graphics applications \(.EXE\)](#)
- [Windows applications \(.EXE\)](#)

Code that works in one application may not work in others. For example, graphics calls are not appropriate in a console application.

Console applications are the most portable to other systems because they are text-only and do not support graphics. With standard graphics applications, you can add graphics to your text without the additional overhead of menus and other interface features of typical programs for Windows.

QuickWin graphics applications provide a simple way to use some features of Windows in a Visual Fortran program with graphics.

Windows applications give users full access to the Win32 Application Programming Interface (API), giving you a larger set of functions than QuickWin offers. With Windows applications, you can access low-level system services directly, or access higher level system services such as OpenGL.

None of the graphics functions in Visual Fortran, except for those in the OpenGL library, are directly portable to operating systems offered by other vendors. A graphical interface does, however, offer certain advantages to the application designer and to the person who will use the program. The choice of what kind of program to build is a trade-off between performance, portability, ease of coding, and ease of use. The advantages and disadvantages of each type of application are summarized in the following sections.

All four kinds of applications can be maximized, minimized, resized, and moved around the screen when displayed in a window. If the drawing area of a window in your application is larger than the window in which it is displayed, scroll bars are automatically added to the bottom and right edges of the window.

You can write any of the applications with one section of the program beginning execution before another has completed. These threads of execution run either concurrently on a computer with one processor or simultaneously on a computer with multiple processors. (See [Creating Multithread Applications](#).)

Console Applications

A console application (.EXE) is a character-based Visual Fortran program that runs in a command console. It looks similar to a program running on a UNIX workstation or a terminal connected to a mainframe computer.

Console applications can be faster than standard graphics or QuickWin graphics applications, because of the time required to display graphical output. Console applications are better suited to problems that require pure numerical processing rather than graphical output or a graphical user

interface. This type of application is also more transportable to other platforms than the other types of application.

As with all Windows command consoles, you can toggle between viewing the console in a window or in full-screen mode by using the ALT+ENTER key combination.

Standard Graphics Applications

A standard graphics application (.EXE) is a Visual Fortran program with graphics that runs in a single window. A standard graphics application looks similar to an MS-DOS program when manipulating the graphics hardware directly, without Windows. You can select displayed text either as a bitmap or as text. Windows provides APIs for loading and unloading bitmap files. Standard graphics applications can be written as either single-threaded or multithreaded applications under Windows NT or Windows 95. (For information about multithreaded programs, see [Creating Multithread Applications](#).)

Standard graphics applications are normally presented in full-screen mode. If the resolution selected matches the screen size, the application covers the entire screen; otherwise, it is a resizable window with scroll bars. You cannot open additional windows in a standard graphics application. Standard graphics applications have neither a menu bar at the top of the window, nor a status bar at the bottom.

Standard graphics applications are appropriate for problems that require numerical processing and graphics, and that do not require a sophisticated user interface.

QuickWin Graphics Applications

QuickWin is a library that lets you build applications with a simplified version of the Windows interface with Visual Fortran. The QuickWin library provides a rich set of Windows features, but it does not include the complete Windows Applications Programming Interface (API). If you need additional capabilities, you must set up a Windows application to call the Win32 API directly rather than using QuickWin to build your program. For more information on QuickWin programming, see [Using QuickWin](#).

QuickWin graphics applications (.EXE) have a multiple-document interface (MDI). Applications that use MDI have a menu bar at the top of the window and a status bar at the bottom. The QuickWin library provides a default set of menus and menu items that you can customize with the QuickWin APIs. An application that uses MDI creates many "child" windows within an outer application window. The user area in an MDI application is a child window that appears in the space between the menu bar and status bar of the application window. Your application can have more than one child window open at a time.

QuickWin applications can also use the DFLOGM.F90 module to access functions to control dialog boxes. These functions allow you to display, initialize, and communicate with special dialog boxes in your application. They are a subset of Win32 API functions, which Windows applications can call directly. For more information on using dialog boxes, see [Using Dialogs](#).

If you build a QuickWin application from the command console, you can use the `/libs:qwins` option to indicate standard graphics applications. A QuickWin application that uses the compiler option is

similar to a standard graphics application in that it has no menu bar or status bar. (In fact, a standard graphics application is a QuickWin application with a set of preset options. It is offered in the program types list for your convenience.) As with a standard graphics application, the application covers the entire screen if the resolution selected matches the screen size; otherwise, it is a resizable window with scroll bars.

Windows Applications

A windows application ("Application" project type; .EXE program type) calls the Windows APIs directly from Visual Fortran. The DFWIN.F90 module contains interfaces to the most common Win32 APIs. If you include the **USE DFWIN** statement in your program, all routines are available to you. The DFWIN.F90 module gives you access to a full range of routines including window management, graphic device interface, system services, multimedia, and remote procedure calls.

Window management gives your application the means to create and manage a user interface. You can create windows to display output or prompt for input. Graphics device interface functions provide ways for you to generate graphical output for displays, printers, and other devices. Win32 system functions allow you to manage and monitor resources such as memory, access to files, directories, and I/O devices. System service functions provide features that your application can use to handle special conditions such as errors, event logging, and exception handling.

Using multimedia functions, your application can create documents and presentations that incorporate music, sound effects, and video clips as well as text and graphics. Multimedia functions provide services for audio, video, file I/O, media control, joystick, and timers.

Remote Procedure Calls (RPC) gives you the means to carry out distributed computing, letting applications tap the resources of computers on a network. A distributed application runs as a process in one address space and makes procedure calls that execute in an address space on another computer. You can create distributed applications using RPC, each consisting of a client that presents information to the user and a server that stores, retrieves, and manipulates data as well as handling computing tasks. Shared databases and remote file servers are examples of distributed applications.

Advanced Applications includes information on how to create a windows application in the section entitled Creating Windows Applications.

If you are using the CD version of Visual Fortran, you can access the Windows API help file directly. You can also obtain information through the Microsoft Developer Network. Developer Network membership includes a development library and a quarterly CD containing technical information for Windows programming.

The full Win32 API set is documented in the *Win32 Application Programming Interface for Windows NT Programmer's Reference*, available from Microsoft Press and also distributed as part of the Windows NT Software Development Kit. For information on joining the Developer Network, see the Microsoft Support Network help file.

Note: Windows projects are much more complex than other kinds of Visual Fortran projects. Before attempting to use the full capabilities of Windows programming, you should be comfortable writing C applications and familiarize yourself with the Windows Software

Development Kit (SDK).

Structuring Your Program

There are several ways to organize your project and the applications that you build with Visual Fortran. This section introduces several of these options, and offers suggestions for when you might want to use them.

For more information, see:

- [Creating Fortran Executables](#)
- [Advantages of Modules](#)
- [Advantages of Internal Procedures](#)
- [Storing Object Code in Static Libraries](#)
- [Storing Routines in Dynamic-Link Libraries](#)

Creating Fortran Executables

The simplest way to build an application is to compile all of your Visual Fortran source files (.FOR) and then link the resulting object files (.OBJ) into a single executable file (.EXE). You can build single-file executables either with Microsoft Developer Studio or by using the DF (or FL32) command from the console command line.

The executable file you build with this method contains all of the code needed to execute the program, including the run-time library. Because the program resides in a single file, it is easy to copy or install. However, the project contains all of the source and object files for the routines that you used to build the application. If you need to use some of these routines in other projects, you must link all of them again.

Advantages of Modules

One way to reduce potential confusion when you use the same source code in several projects is to organize the routines into modules. There are two main uses for modules in Visual Fortran:

- Internal encapsulation--A single complex program can be made up of many modules. Each module can be a self-contained entity, incorporating all the procedures and data required for one of your program's tasks. When a task is encapsulated, it is easy to share the code between two different projects.

In this case, all the modules should be included in the main project directory. If many projects all share the same module, the module should reside in only one directory. All projects that use it should specify the /I compiler option to indicate the location of the module.

- External modules--If you use a module provided from an outside source, you need only the .MOD file at compile time, and the .OBJ file at link time. Use the `/[no]include[path]` (or `/Ipath`) command line option (or the INCLUDE environment variable) to specify the location of these files, which will probably not be the same as your project directory.

During the building of a project, the compiler scans the project files for dependencies. If you specify

the `/[no]include[path]` (or `/Ipath`) command line option or the `INCLUDE` environment variable, the compiler is able to find the external modules.

Store precompiled module files, with the extension `.MOD`, in a directory included in the path. When the compiler sees the `USE` statement in a program, it finds the module based on the name given in the `USE` statement, and there is no need to maintain several copies of the same source or object code.

Modules are excellent ways to organize programs. You can set up separate modules for:

- Commonly used routines
- Data definitions specific to certain operating systems
- System-dependent language extensions

Advantages of Internal Procedures

Functions or subroutines that are used in only one program can be organized as internal procedures, following the `CONTAINS` statement of a program or module.

Internal procedures have the advantage of host association, that is, variables declared and used in the main program are also available to any internal procedure it may contain. For more information on procedures and host association, see [Program Units and Procedures](#).

Internal procedures, like modules, provide a means of encapsulation. Where modules can be used to store routines commonly used by many programs, internal procedures separate functions and subroutines whose use is limited or temporary.

Storing Object Code in Static Libraries

Another way to organize source code used by several projects is to build a static library (`.LIB`) containing the object files for the reused procedures. You can create a static library either by building a project of that type from Microsoft Developer Studio or by using the `LIB` command from the console. After you have created a static library, you can use it to build any of the other types of Visual Fortran projects.

If you use a static library, only those routines actually needed by the program are incorporated into the executable image (`.EXE`). This means that your executable image will be smaller than if you included all the routines in the library in your executable image. Also, you do not have to worry about exactly which routines you need to include -- the link program takes care of that for you.

Because applications built with a static library all contain the same version of the routines in the library, you can use static libraries to help keep applications current. When you revise the routines in a static library, you can easily update all the applications that use it by relinking the applications.

Storing Routines in Dynamic-Link Libraries

Another method of organizing the code in your application involves storing the executable code for certain routines in a separate file called a Dynamic-Link Library (`DLL`), and building your applications so that they call these routines from the `DLL`. When routines in a `DLL` are called, the routines are loaded into memory at run-time, as they are needed. This is most useful when several

applications use a common group of routines. By storing these common routines in a DLL, you reduce the size of each application that calls the DLL. In addition, you can update the routines in the DLL without having to rebuild any of the applications that call the DLL.

With Visual Fortran, you can use DLLs in two ways. First, you can build a DLL with your own routines. In Microsoft Developer Studio, select dynamic-link library as your project type; from the command line use the /DLL option with DF. As stated earlier, Visual Fortran also lets you build applications with the run-time library stored in a separate DLL instead of in the main application file. DLL run-time routines can be selected with a compiler option.

For more information on compiler and linker options and how to build a project, see [Win32 Dynamic-Link Library Projects](#).

Special Design Considerations

You can write your code any way you want if you plan to run it on a single computer, use only one variation of one programming language, and never hand your code to anyone else. If any of these assumptions changes, there are several other issues to consider when you design your program.

For more information, see:

- [Porting Fortran Source Between Systems](#)
- [Mixed-Language Issues](#)
- [Porting Data Between Systems](#)

Porting Fortran Source Code Between Systems

In general, Visual Fortran is a portable language. One of the main advantages of the language is the availability of large and well-tested libraries of Fortran code. You also might have existing code addressing your problem that you want to reuse. Math and scientific code libraries from most vendors should port to Visual Fortran with virtually no problems.

You might also want to use Visual Fortran as a development platform for code that can later be ported to another system, such as mainframe-class Alpha systems running the DIGITAL UNIX® or the OpenVMS™ operating system.

Whether you are bringing code from another system or planning to export it to another system, you will need to do the following:

- Isolate system-dependent code into separate modules. Maintain distinct modules with similar functionality for each separate platform.
- In your main program, use only language extensions that will compile on both platforms, putting system-dependent code into modules.
- Place language extension subsets into modules.
- If you use Microsoft compiler directives, replace the older *\$directive* format with the **!DEC\$directive** format, because this will be ignored by other systems.
- Specify data precision, for integers and logicals as well as for floating-point numbers when the size matters. If you do not explicitly specify KIND for variables, this could be the source of problems if one system uses a default of (KIND=2) for integers, while your program assumes

(KIND=4).

- Conversely, if the size of a variable is not significant, avoid specifying data precision. Code that does specify precision will run slower on systems that do not use the same default integer and real sizes.
- Avoid using algorithms that exhibit floating-point instability. For information on handling floating-point numbers, see [The Floating-Point Environment](#).
- Specify equivalent floating-point precision on each platform.
- Specify the appropriate attributes when defining routines and data that will be interacting with code written in Microsoft® Visual C/C++® or assembly language.

For more information on porting code between systems, see [Portability](#).

Choosing a Language Extension Subset

The Visual Fortran compiler supports extensions used on a variety of platforms, plus some that are specific to Visual Fortran. Because there are Fortran compilers for many different computers, you might need to move your source code from one to another. If the trip is one-way and it is permanent, you can simply change the code to work on the second platform. But if you need to make sure you can move the code whenever needed, you must be aware of the extensions to Fortran that are supported on each platform.

You can use some of the Visual Fortran compiler options to help you write portable code. For example, by specifying ANSI syntax adherence in the Compiler Options dialog box or on the command line, you can have the compiler enforce Fortran 90 syntax. Code that compiles cleanly with this option set is very likely to compile cleanly on any other computer with a Fortran compiler that obeys strict Fortran syntax.

If you choose to use platform-specific extensions, you need to note whether there are any differences in how those extensions are implemented on each computer, and use only those features that are identical on both. (For more information, see [Portability](#).) The default is to compile with the full set of extensions available.

Because Visual Fortran compiler directives look like standard Fortran comments (**!DEC\$directive**), programs that use directives can compile on other systems. They will, however, lose their function as compiler directives.

Floating-Point Issues

Floating-point answers can differ from system to system, because different systems have different precisions and treat rounding errors in different ways.

One programming practice that can be a serious source of floating-point instability is performing an **IF** test (either obvious or implied) that takes some action if and only if a floating-point number exactly equals a particular value. If your program contains code like this, rewrite the code to a version that is stable in the presence of rounding error. For more details, see [The Floating-Point Environment](#) and [Portability](#).

Another source of floating-point instability is the use of mathematical algorithms that tend to diminish precision. Incorrect answers can result when the code is moved to a system with less precision. For more information, see [The Floating-Point Environment](#).

One way of making all REAL variables on one system DOUBLE PRECISION on another is to use modules to declare explicit data types for each system. Specify a different KIND parameter in each module. Another way is to add an include file that declares explicit data types on each system in all source files.

Mixed-Language Issues

You can combine object modules generated by Visual Fortran with object files from compilers for 32-bit Windows that compile other languages (such as Microsoft Visual C++, or Microsoft® MASM), so long as the compilers use the COFF object module format used by Microsoft.

You need to respect certain calling, naming, and argument-passing conventions when combining object modules from different languages. These conventions are discussed in [Programming with Mixed Languages](#).

Porting Data Between Systems

The easiest way to port data to or from the Visual Fortran environment is as a formatted, sequential, 8-bit ASCII character file that can be read using Fortran formatted input statements; if you do this, you should have no trouble.

If you try to transfer unformatted binary data between systems, you need to be aware of the different orders (low-order byte first or high-order byte first) in which different systems store bytes within words. If you need to transfer unformatted binary data, review [Portability](#) and [Converting Unformatted Numeric Data](#). You can avoid these problems by using a formatted ASCII file.

Using the Special Features of Microsoft Windows

One of the greatest advantages to building applications for Windows is the power and security provided by the operating system. By simply recompiling your old source code and building a (text-only) console application, you can run your program in a protected address space where it cannot damage other applications, hang the processor, or cause the computer to crash.

If you choose to take advantage of the power of Windows NT or Windows 95, your programs can run more efficiently on single-processor computers. Window NT also supports multi-processor computers.

For more information, see:

- [Built-in Benefits of Windows](#)
- [Single or Multithread Program Execution](#)
- [QuickWin and Windows Programs](#)

Built-in Benefits of Windows

Windows executes your application in a secure environment that includes the support services your application needs to execute efficiently and with a minimum of problems. This environment is a flat virtual address space that can be as large as 2 gigabytes, providing you have enough available disk

space. While executing, your program is protected by Windows from damaging other applications and from being damaged by other applications.

The operating system uses *preemptive multitasking* to control how much processor time each application uses. Instead of waiting for an application to voluntarily yield control of the computer back to the operating system, Windows allocates a period of processor time to the application and regains control when that period has expired. This prevents a program with an infinite loop from hanging the computer. If your program hangs, you can easily and safely stop it by using the Windows task manager. (For information about using this or any other feature of Windows, see the manuals that came with the operating system.)

Because you can use one application while another continues to execute, you can make better use of your own time. For example, you can use Microsoft Developer Studio to edit the source for one project while another project is building, or use Microsoft Excel to prepare a graph for data that your program is busy producing. And if your computer has multiple processors and you are using Windows NT, the computation-intensive program producing your data might be executing on an otherwise idle processor, making it less likely that your other work will slow it down.

Single or Multithread Program Execution

You can take further advantage of preemptive multitasking by designing your program so that portions of it, called *threads*, can be executed in parallel. For example, one thread can perform a lengthy input/output operation while another thread processes data. All of the threads in your application share the same virtual address space. Both Windows 95 and Windows NT support multithreading.

When running Windows NT on a symmetric multiprocessor machine (sometimes called an "SMP machine") you can achieve a substantial speedup on numerically intensive problems by dividing the work among different threads; the operating system will assign the different threads to different processors. Even if you have a single processor machine, multiple-window applications might benefit from multithreading because threads can be associated with different windows; one thread can be calculating while another is waiting for input.

Multithreaded code must be written so that the threads do not interfere with each other and overwrite each other's data. [Creating Multithread Applications](#) describes how to do this. If your multithreaded code calls functions from the run-time library or does input/output, you must also link your code to the multithreaded version of the run-time libraries instead of the regular single-threaded ones. This is described in in [Compiling and Linking Multithread Programs](#) and [Building Programs and Libraries](#).

While you might gain execution speed by having a program executed in multiple threads, there is overhead involved in managing the threads. You need to evaluate the requirements of your project to determine whether you should run it with more than one thread.

QuickWin and Windows Programs

One decision you must make when designing a program is how it will be used. If the person using your program must interact with it, the method of interaction can be important. For example, anytime the user must supply data, that data must be validated or it could cause errors. One way to minimize

data errors is to change how the data is provided. In this example, if the data is one of several values that are known when the program is executed, the user can select a menu item instead of typing on the keyboard.

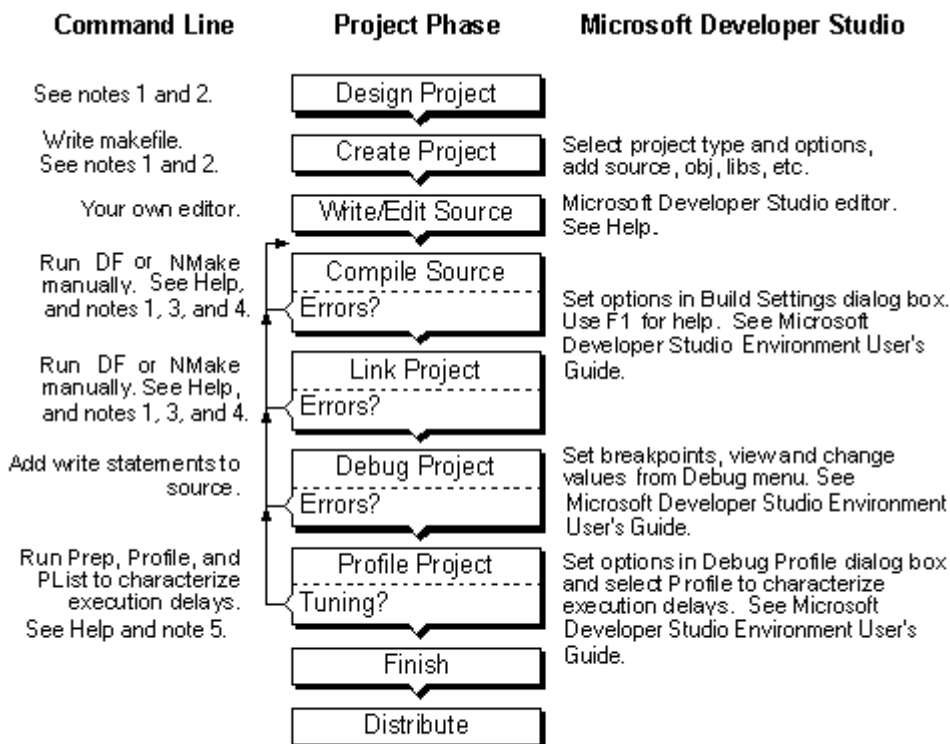
When you design programs to be interactive, you use a different structure than if you design them to be run in unattended batches. Interactive applications behave more like state machines than numerical algorithms, because they perform the actions you request when you request them. You may also find that once you can change what your program is doing while it runs, you will be more likely to experiment with it.

The QuickWin library lets you build simple Windows applications. Because QuickWin is a wrapper around a subset of the Windows API, there are limitations to what you can do, but it can fulfill the requirement of most users. If you need additional capabilities, you can call the Windows API directly rather than using QuickWin to build your program. (For more information, see [Using QuickWin](#)). You can also build a graphic user interface in either Microsoft®C or Visual Basic® that calls your Fortran code.

Development Environments

Whether you choose to use Microsoft Developer Studio or the console command line to build your programs, the process that you use is constant.

However, your choice of development environment determines what you can do at each stage. The following diagram illustrates the development process for both choices.



note 1: Building Programs and Libraries

note 2: Using the Compiler and Linker from the Command Line and Using Visual Fortran Tools

note 3: Advanced Applications

note 4: The Floating-Point Environment

note 5: Profiling Code from the Command Line

Building Programs and Libraries

Microsoft Developer Studio makes it easy for you to create, debug, and execute your programs. It includes a full-feature editor and interactive help. For complete details on how to use the development environment, see the [Developer Studio Environment User's Guide](#)

You can build your source code into several types of programs and libraries, either using Developer Studio or working from the command line. This section covers the general issues involved in building projects with Developer Studio:

- [Overview of Building Projects](#)
- [Types of Projects](#) you can build
- [Defining Your Project](#) and selecting project features with Developer Studio
- [Errors During the Build Process](#)
- [Running Fortran Applications](#)
- [Porting Projects Between x86 and Alpha Platforms](#)

For information on building programs and libraries at the command line, see:

- [Using the Compiler and Linker from the Command Line](#)
- [Using Visual Fortran Tools](#)

Overview of Building Projects

Microsoft Developer Studio organizes development into *projects*. A project consists of the source files required for your application, along with the specifications for building the project. The build process involves defining your project, setting options for it, and building the program or library.

Each project can specify one or more *configurations* to build from its source files. A *configuration* specifies such things as the type of application to build, the platform on which it is to run, and the tool settings to use when building. Having multiple configurations allows you to extend the scope of a project but still maintain a consistent source code base from which to work.

When you create a new project, Developer Studio automatically creates Debug and Release configurations for you. To specify the configuration, from the Build menu select Set Active Configuration.

The development environment includes a *FileView* pane, which displays the files contained in the project, and lets you examine visually the relationships among the files in your project. Modules, include files, or special libraries your program uses are automatically listed as *dependencies*. The output window displays information produced by the compiler, linker, Find in Files utility, and the profiler.

You can specify build options in the Project menu Settings dialog box, for one of the following:

- The entire project
- For certain configurations
- For certain files

For example, you can specify certain kinds of compiler optimizations for your project in general, but

turn them off for certain configurations or certain files.

Once you have specified the files in your project, the configurations that your project is to build, and the tool settings for those configurations, you can build the project with the commands on the Build menu.

For more information, see:

- [How Information Is Displayed](#)
- [Menu Options](#)
- [Using the Shortcut Menu](#)

How Information Is Displayed

Microsoft Developer Studio displays information in windows, panes, and folders. One window can contain several panes, and each pane can display one or more folders. A *pane* is a separate and distinct area of a window; a *folder* is a visual representation of files in a project. Folders show the order in which Visual Fortran compiles the files, and the relationship of source files to their dependent files, such as modules.

When you initially create a project, the Project Workspace window contains some default panes, accessible through tabs at the bottom of the window, to display information about the content of the project. You can also open an output window, which has panes that display build output, debug output, Find in Files output, and profiler output. In addition to the default panes, you can create customized panes to organize and display project information in ways most useful to you.

You can access information about components of the project from the panes in the project window. Double-clicking any item in a pane displays that item in an appropriate way: source files in a text editor, dialog boxes in the dialog editor, help topics in the information window, and so on.

Be sure to select the appropriate pane when using the menu commands, in particular the Save and Save As commands. Commands on the File menu affect only the window that currently has the focus.

Menu Options

Menu options that are available to you may look different, depending on which window or pane has current focus. The Debug menu, for example, is only visible when you are debugging. Microsoft Developer Studio has the following menu bars and toolbars:

- Standard menu bar
- Standard toolbar
- Build toolbar
- Build minibar
- Resource toolbar
- InfoViewer toolbar
- Edit toolbar
- Debug toolbar
- Browse toolbar
- Fortran toolbar (format editor)

You can select or deselect the menu bars and toolbars from the Tools menu Customize item, Toolbar tab.

Using the Shortcut Menu

The project window has a shortcut menu that lists commands appropriate for the current selection in the window. This is a quick method to display commands that are also available from the main menu bar.

► To display the shortcut menu:

- Move the mouse pointer into the project window and click the right mouse button.

You can now select project commands that are appropriate for your current selection in the project window.

Types of Projects

Each project has a type, which you choose when you create the project. You need to create a project for each binary executable file to be created. For example, the main Fortran program and a Fortran dynamic-link library (DLL) would each reside in the same workspace as separate projects.

The project type specifies what to generate and determines some of the options that Microsoft Developer Studio sets by default for the project. It determines, for instance, the options that the compiler uses to compile the source files, the static libraries that the linker uses to build the project, the default locations for output files, defined constants, and so on.

You can build six kinds of projects with Visual Fortran. You specify the project type when you create a new project. They are summarized in the following table:

Project type	Key features
<u>Win32 Console Application</u> (.EXE)	Single window main projects without graphics (resembles character-cell applications). Requires no special programming expertise. For a sample of a Console Application, see \MYPROJECTS\CELSIUS, as described in Opening an Existing Project .
<u>Standard Graphics Application</u> (.EXE)	Single window main projects with graphics. The programming complexity is simple to moderate, depending on the graphics and user interaction used. Samples of Standard Graphics Application (QuickWin single window), resemble those for QuickWin Applications.
<u>QuickWin Application</u> (.EXE)	Multiple window main projects with graphics. The programming complexity is simple to moderate, depending on the graphics and user interaction used. Samples of QuickWin Applications (QuickWin multiple window) are in \DF\SAMPLES\GENERAL, such as QWPiANO and QWPiAINT.
<u>Win32 (Windows) Application</u> (.EXE)	Multiple window main projects with full graphical interface and Win32 API functions. Requires advanced programming expertise and knowledge of the Win32 API. Samples of Win32 Applications are in \DF\SAMPLES\ADVANCED\WIN32, such as PLATFORM or POLYDRAW.

<u>Win32 Static library (.LIB)</u>	Library routines to link into .EXE files.
<u>Win32 Dynamic-Link Library (.DLL)</u>	Library routines to associate during execution.

The first four projects listed in the preceding table are main project types, requiring main programs. The next two are library projects, without main programs. The project types are discussed in detail in:

- [Win32 Console Application](#)
- [Standard Graphics Application](#)
- [QuickWin Application](#)
- [Win32 \(Windows\) Application](#)
- [Win32 Static library](#)
- [Win32 Dynamic-Link Library](#)

Other sections related to project types include:

- [Selecting a Program Type](#)
- [Advanced Applications](#)
- [Specifying Project Types with DF Command Options](#)

Win32 Console Application Projects

Console projects are suitable for character-based applications not requiring screen graphics output. These projects operate in a single window, and allow you to interact with your program through normal read and write commands.

Any graphics routine that your program calls will produce no output, but will return error codes. A program will not automatically exit if such an error occurs, so your code should be written to handle this condition.

With a console project, you can use static libraries, DLLs, and dialog boxes, but you cannot use the QuickWin functions. You can select the multithreaded libraries with this and all of the other project types.

Standard Graphics Application Projects

Standard graphics applications (.EXE) operate in a single window that allows graphics output (such as drawing lines and basic shapes) and other screen functions, such as clearing the screen. Standard Graphics is a subset of Quickwin, sometimes called *Quickwin single window*. You can use all of the QuickWin graphics functions in these projects. You can use dialog boxes with these and all other project types (see [Using Dialogs](#)).

When you select the standard graphics project type, Microsoft Developer Studio includes the QuickWin library automatically, enabling you to use the graphics functions. When building from the command line, you must specify the `/libs:qwins` option. You cannot use the run-time functions meant for multiple-window projects if you are building a standard graphics project. You cannot make a Standard Graphics application a DLL.

The single window can be either full-screen or have window borders and controls available. You can change between these two modes by using ALT+ENTER.

For more information about Standard Graphics (QuickWin single window) applications, see [Using Quickwin](#).

QuickWin Application Projects

QuickWin graphics applications (.EXE) are more versatile than standard graphics applications because you can open multiple windows while your project is executing. For example, you might want to generate several graphic plots and be able to switch between them while also having a window for controlling the execution of your program. These windows can be full screen or reduced in size and placed in various parts of the screen.

When you select the QuickWin graphics project type, Microsoft Developer Studio includes the QuickWin library automatically, enabling you to use the graphics functions. When building from the command line, you must specify the `/libs:qwin` compiler option. You cannot make a QuickWin application a DLL.

For information on how to use QuickWin functions, including how to open and control multiple windows, see [Using Quickwin](#).

Win32 (Windows) Application Projects

Win32 (Windows) applications (.EXE) are main programs selected by choosing the Win32 Application project type. This type of project lets you have full access to the Win32 APIs, giving you a larger (and different) set of functions to work with than QuickWin. You can call some of the Win32 APIs from the other project types. The full set of Win32 functions available for Win32 (Windows) applications allows use of certain system features not available for the other project types.

For more information, see [Creating Windows Applications](#).

Note: Windows projects are much more complex than other kinds of Visual Fortran projects. Before attempting to use the full capabilities of Windows programming, you should be comfortable with writing C applications and should familiarize yourself with the Windows Software Development Kit (SDK).

Win32 Static Library Projects

Static libraries (.LIB) are blocks of code compiled and kept separate from the main part of your program; you would usually keep them in their own directories. They offer important advantages in organizing large programs and in sharing routines between several programs. These libraries contain only subprograms, not main programs. When you associate a static library with a program, any necessary routines are linked from the library into your executable program when it is built.

A static library is a collection of source and object code defined in the FileView pane. The source code is compiled when you build the project. The object code is assembled into a .LIB file without

going through a linking process. The name of the project is used as the name of the library file by default.

If you have a library of substantial size, you should maintain it in a dedicated directory. Projects using the library access it at linking time.

When you link a project that uses the library, selected object code from the library is linked into that project's executable code to satisfy calls to external procedures. Unnecessary object files are not included.

When compiling a static library from the command line, include the `/c` option to suppress linking. Without this option, the compiler generates an error because the library does not contain a main program.

To debug a static library, you must use a main program that calls the library routines. Both the main program and the static library should have been compiled using the debug option. After compiling and linking is completed, open the Debug menu and choose Go to reach breakpoints, use Step to Cursor to reach the cursor position, or use the step controls on the Debug toolbar.

Using Static Libraries

You add static libraries to a main project in Microsoft Developer Studio with the Insert Files into Project dialog box. Enter the path and library name in the Insert Files into Project box with a `.LIB` extension on the name. If you are using a foreign makefile, you must add the library by editing the makefile for the main project. If you are building your project from the command line, add the library name with a `.LIB` extension and include the path specification if necessary.

For more information about static libraries, see:

- [Storing Object Code in Static Libraries](#)

Win32 Dynamic-Link Library Projects

A dynamic-link library (`.DLL`) is a source-code library that is compiled and linked to a unit independently of the applications that use it. A DLL shares its code and data address space with a calling application. A DLL contains only subprograms, not main programs.

A DLL offers the organizational advantages of a static library, but with the advantage of a smaller executable file at the expense of a slightly more complex interface. Object code from a DLL is not included in your program's executable file, but is associated as needed in a dynamic manner while the program is executing. More than one program can access a DLL at a time.

For more information about DLLs, see:

- [Storing Routines in Dynamic-Link Libraries](#)
- [Building Dynamic-Link Library Projects](#)
- [DLLs in Advanced Applications](#)

Defining Your Project

To create a new project, use the File menu and select New. A dialog box opens that has the following tabs:

- Files
- Projects
- Workspaces
- Other Documents

The Projects tab displays various project types. Specify the project name and location. Click the type of Fortran project to be created. If you have other Visual tools installed, make sure you select a Fortran project type (see [Types of Projects](#)).

You can set the Create New Workspace check box to create a new Workspace.

Click OK to create the new project.

You can add files to the project by selecting Add To Project from the Project menu:

- To add an existing file to the project:
 1. Select Files... from the submenu.
 2. The Insert Files into Project dialog box appears. Use this dialog box to select the Fortran files to be added to the Project. To add more than one file to the project, hold down the CTRL key as you select each file name.
- To add a new file to the project:
 1. Select Add to Project New...
 2. The New dialog box appears. Specify the file name and its location.
 3. Click the type of file (Fortran Fixed Format Source or Fortran Free Format Source).
 4. Click OK. The editor appears allowing you to type in source code. The file name appears in the FileView pane.

For information on:

- How to use icon files, see [Using QuickWin](#).
- Using binary files, see [Files, Devices, and I/O Hardware](#).
- Creating and adding to a new project, see the "How Do I ..." section of [Working With Projects in the Developer Studio Environment User's Guide](#).
- Using the Resource Editor, Dialog Editor, or Graphics Editor, see the [Developer Studio Environment User's Guide](#).

You need to add these kinds of files to your project:

- Program files with .FOR, .F or .F90 extension
- Resource files with .RC extension

Include files (extension .FI, or any file your program refers to with an **INCLUDE** statement) do not need to be added to the list of files.

To define a project from a set of existing or new source files:

- On the File menu, click New...
- Click the Projects tab.
- Select the type of project and name it.
- To add an existing file to the project:
 1. Select Files... from the submenu.
 2. The Insert Files into Project dialog box appears. Use this dialog box to select the

Fortran files to be added to the Project. To add more than one file to the project, hold down the CTRL key as you select each file name.

- To add each new file to the project:
 1. Select Add to Project New...
 2. The New dialog box appears. Specify the file name and its location.
 3. Click the type of file (Fortran Fixed Format Source or Fortran Free Format Source).
 4. Click OK. The editor appears allowing you to type in source code. The file name appears in the FileView pane.
- You can now select "Build *filename*" from the Build Menu to build your application.

For more information on projects, see:

- [Files in a Project](#)
- [Selecting Project Features](#)
- [Selecting a Configuration](#)
- [Setting Build Options](#)
- [Creating the Executable Program](#)
- [Building Dynamic-Link Library Projects](#)

Files in a Project

When you create a project, Microsoft Developer Studio always creates the following files:

- Project workspace file - Has the extension .DSW. It stores project workspace information.
- Project file - Has the extension .DSP. It is used to build a single project or subproject.
- Workspace options file - Has the extension .OPT. It contains environment settings for Visual Fortran, such as window sizes and positions, insertion point locations, state of project breakpoints, contents of the Watch window, and so on.

Directly modifying the DSW and DSP files with a text editor is not supported.

For information on creating (exporting) a makefile, see [The Project Makefile](#).

When you create a project, you also identify a project subdirectory. If the subdirectory does not exist, Microsoft Developer Studio creates it. Project files that Developer Studio creates are put into this directory.

When you create a project, Developer Studio also specifies subdirectories for intermediate and final output files for the various configurations that you specify. These subdirectories allow you to build configurations without overwriting intermediate and final output files with the same names. The General tab in the Project Settings dialog box allows you to modify the subdirectories, if you choose.

If you have existing source code, you should organize it into directories before building a project, although it is easy to move files and edit your project definitions if you should later decide to reorganize your files.

If your program uses modules, you do not need to explicitly add them to your project, they appear as dependencies. Developer Studio scans the file list for modules and compiles them before program units that use them. Developer Studio automatically scans the added project files recursively for modules specified in **USE** statements, as well as any **INCLUDEs**. It scans both source files (.FOR, .F, .F90) and resource files (.RC), and adds all the files it finds to a Dependencies folder. You cannot

directly add or delete the files listed in this folder.

The Project Makefile

Developer Studio speeds and simplifies the task of building programs and libraries outside of Developer Studio by allowing you to export a makefile, which is a set of build instructions for each project. Makefiles contain the names of the source, object, and library files needed to build a program or library, plus the compiler and linker options selected in the Project Settings dialog boxes.

Developer Studio updates the build instructions in internal makefiles when you add or remove project files in the project window, and when you make changes to the compiler or linker options in the Project Settings dialog boxes. To get an updated version of a makefile, from the Project menu, select Export Makefile. The makefile is used by the external program maintenance utility, NMAKE.EXE.

You can edit the makefile generated by Developer Studio if you need to perform unusual or exceptional builds. Remember, however, that once you have edited a makefile, exporting the makefile again from Developer Studio will overwrite your changes.

If you use a foreign makefile for a project, Developer Studio calls NMAKE to perform the build. You can run NMAKE from the console command line to perform builds either with makefiles exported by Developer Studio or with foreign makefiles that you have edited. For more about the the external program maintenance utility, see [Building Projects with NMAKE](#).

Note: When you use a foreign makefile, the project is considered to be foreign. You cannot use the Project Settings dialog box to make changes to the build options, or use the Add to Project dialog box to add files.

Selecting Project Features

Before you can build a project, you must:

- [Select a configuration type](#)
- [Select build settings](#), such as dependencies, compile and link options

Selecting a Configuration

A configuration defines the final binary output file that you create within a project. A configuration has the following characteristics:

- [Project type](#)--Specifies the type of application to build, such as a static library, console application, QuickWin application, and so on. The word *Win32 Application* alone means a Windows application.
- [Build options](#)--Specifies the build options.

When you create a new project, Developer Studio creates the following configurations:

- Debug configuration

By default, the debug configuration sets project options to include the debugging information in the debug configuration. It also turns off optimizations. Before you can debug an

application, you must build a debug configuration for the project.

- Release configuration

The release configuration does *not* include the debugging information, and it uses any optimizations that you have chosen.

Select the configuration in the Build menu, Set Active Configuration item.

You can define new configurations within your project. These configurations can use the existing source files in your project, the existing project settings, or other characteristics of existing configurations. A new configuration does not have to share any of the characteristics or content of existing configurations, however.

You could, for instance, create an initial project with debug and release configurations specifying an application for the Win32 environment, and add source files to the project. Later, within the project, you could create debug and release configurations specifying a DLL for the Win32 environment, add an entirely disjoint set of files to this configuration, and make these configurations dependencies of the application configurations.

Platform Types

The platform type specifies the operating environment for a project. The platform type sets options required specifically for a given platform, such as options that the compiler uses for the source files, the static libraries that the linker uses for the platform, the default locations for output files, defined constants, and so on. Visual Fortran supports the Win32 platform type.

Setting Build Options

When you create a new configuration, you specify options for file creation and build settings by selecting the Settings item in the Project menu. These options can include compile and link options, optimization, or browse information.

Configurations have a hierarchical structure of options. The options set at the configuration level apply to all files within the configuration. Setting options at the configuration level is sufficient for most configurations. For instance, if you set default optimizations for the configuration, all files contained within the configuration use default optimizations.

However, you can set different options for files within a configuration, such as specific optimization options -- or no optimization at all -- for any individual files in the configuration. The options that you set at the file level in the configuration override options set at the configuration level.

You can set some types of options, such as linking, only at the configuration level.

You can set options at the following levels within a configuration:

- Configuration level--Options set at this level apply to all actions. Any options set for the configuration apply to every file in the configuration unless overridden at the file level.
- File level--Options set at this level apply to file-level actions, such as compiling. Any options set for the file apply only to that file and override any options set at the configuration level.

You can insert both source (.FOR, .F90, .F, .FI, .FD) and object (.OBJ) files by using the Project menu Add to Project, Files item. It is usually better to insert all source files for your programs. That way, if you update a source file, Developer Studio can create a new object file and links it into your project. Also insert the names of any necessary static libraries and DLLs with .LIB extensions to be linked with your project. Use only the library names, not the names of any files within the libraries.

You can include C source code files only if you have Microsoft Visual C++ Version 5.0 installed, but you can include C object code without Visual C++ being present.

Compile and Link Options

You can set any of the compiler or linker options described in [Compiler and Linker Options](#) in the Project menu, Settings dialog box. The Fortran tab of this dialog box presents several categories of options to set. The options are grouped under different categories. Select the category from the Category drop-down list (see [Categories of Compiler Options](#)). You can choose compiler and linker options through the various dialog boxes. If a compiler option is not available in the dialog boxes, you can enter the option in the lower part of the window just as you would at the command line.

You can also compile, link, and set options at the command line if you choose. For information on how to do this, see [Using the Compiler and Linker from the Command Line](#).

The linker builds an executable program (.EXE), static library (.LIB), or dynamic-link library (.DLL) file from Common Object File Format (COFF) object files and other libraries identified in the linker options. You direct the linker by setting linker options either in Microsoft Developer Studio, in a build instructions file, or on the console command line. For example, you can use a linker option to specify what kind of debug information to include in the program or library.

For more information on compiler and linker options, see [Compiler and Linker Options](#).

Source Browser Information

The Source Browser generates a listing of all symbols in your program; information that can be useful when you need to debug it, or simply to maintain large volumes of unwieldy code. It keeps track of locations in your source code where your program declares, defines, and uses names. You can find references to variables or procedures in your main program and all subprograms it calls by selecting one of the files in your project, then using the Go to Definition or Go to Reference button on the Browse toolbar. Source Browser information is available only after you achieve a successful build.

Browser information is off by default for projects, but you can turn it on if you wish. You can set the browse option:

- In Developer Studio:
 1. In the General category of the Fortran tab, set the Generate Source Browse Information check box.
 2. Click the BrowseInfo tab and set the Build Browse info check box.
- On the command line:
 1. Specify the `/browser` option.
 2. Use the Browse Information File Maintenance Utility (BSCMAKE) utility to generate a browse information file (.BSC) that can be examined in browse windows in Developer

Studio.

When the browse option is on, the compiler creates intermediate .SBR files when it creates the .OBJ files; at link time, all .SBR files in a project are combined into one .BSC file. These files are binary, not readable, but they are used when you access them through the Browser menu.

Creating the Executable Program

Once you are ready to create an executable image of your application, select the Build menu. You can:

- Compile a file without linking
- Build a project
- Rebuild all parts of a project
- Batch build several configurations of a project
- Clean extra files created by project builds
- Execute the program, either in debug mode or not
- Update program dependencies
- Select the active project and configuration
- Edit the project configuration
- Define and perform profiling

Once you have completed your project definition, you can build the executable program.

When you select Build *projectname* from the Build menu (or one of the Build toolbars), Developer Studio automatically updates dependencies, compiles and links all files in your project. When you build a project, Developer Studio processes only the files in the project that have changed since the last build.

The Rebuild All mode forces a new compilation of all source files listed for the project.

You either can choose to build a single project, the current project, or you can choose multiple projects (requires batch build) to build in one operation.

You can execute your program from Microsoft Developer Studio using Ctrl+F5 or Execute from the Build menu (or Build toolbar), or from the command prompt.

Compiling Files In a Project

You can select and compile individual files in any project in your project workspace. To do this, select the file in the project workspace window (FileView tab). Then, do one of the following:

- Press Ctrl+F7.
- or-
- Choose Compile from the Build menu (or Build toolbar).
- or-
- Click the right mouse button to display the pop-up menu and select Compile.

You can also use the Ctrl+F7 or Compile from the Build menu (or Build toolbar) options when the source window is active (input focus).

Building Dynamic-Link Library Projects

A dynamic-link library is a collection of source and object code in the same manner as a static library. The differences between the two libraries are that the DLL requires an interface specification and it is associated with a main project during execution, not during linking.

When a DLL is built, two library files are created. One is an import library (.LIB), which the linker uses to associate a main program with the DLL. The other is the .DLL file containing the library's executable code. Both files have the same basename as the library project by default.

Data is by default not shared between instances of the same DLL, which is a change from the 16-bit applications, but there are ways to share data between instances. For more details, see the DLL samples in the ...\\DF\\SAMPLES\\ subdirectories.

The DLL must reside either in the same directory as the program using it, or in a directory listed in the PATH environment variable.

For more information, see:

- [Using Microsoft Developer Studio to Build DLLs](#)
- [Organization and Behavior of DLLs](#)
- [Importing and Exporting Data with DLLs](#)
- [Building the DLL](#)
- [The DLL Build Output](#)
- [Using DLLs](#)
- [QuickWin Restrictions](#)

Using Microsoft Developer Studio to Build DLLs

To build a DLL with Microsoft Developer Studio, select Dynamic-Link Library as the project type when you create a new project.

To debug a DLL, you must use a main program that calls the library routines. From the Project Settings menu, choose the Debug tab. A dialog box is available for you to specify the executable for a debug session.

Organization and Behavior of DLLs

Windows calls the **DLLMain** entry-point function in a DLL to initialize and to terminate execution of the DLL. If you don't include a **DLLMain** function in your source, Visual Fortran provides one that returns **.TRUE.** when it is called. Windows also makes calls to **DLLMain** on both a per-process and per-thread basis, so several initialization calls can be made to the DLL if a process is multithreaded. Fortran code written for a DLL must be re-entrant if you want to allow multiple threads of execution to use the DLL simultaneously. (For information about writing reentrant code, see [Creating Multithread Applications](#).)

DLLMain returns **.TRUE.** to indicate success. If the function returns **.FALSE.** during per-process initialization, the system cancels the process. **DLLMain** is called with a parameter that indicates the reason it was called: initialization or termination, for a process or a thread. The following table describes the meaning of the four possible values:

Value of passed parameter	Description
DLL_PROCESS_ATTACH	A new process is attempting to access the DLL; one thread is assumed.
DLL_THREAD_ATTACH	A new thread of an existing process is attempting to access the DLL; this call is made beginning with the <i>second thread</i> of a process attaching to the DLL.
DLL_PROCESS_DETACH	A process is detaching from the DLL.
DLL_THREAD_DETACH	One of the additional threads (not the first thread) of a process is detaching from the DLL.

Each time a new process attempts to use the DLL, the operating system performs what is called a *process attach*, which means it creates a separate copy of the DLL's data. The run-time library code for the DLL then calls the **DLLMain** function with process attach selected. The opposite situation is process detach: the run-time library code calls **DLLMain** with process detach selected. Note that the order of events in process detach is the reverse of that in process attach.

Importing and Exporting Data with DLLs

Data and code in a dynamic-link library is loaded into the same address space as the data and code of the program that calls it. However, variables and routines declared in the program and in the DLL are not shared unless you use the `DLLIMPORT` and `DLLEXPORT` compiler directives. These directives enable the compiler and linker to map to the correct portions of the address space so that the data and routines can be shared.

You can use `DLLEXPORT` to declare that a common block in a DLL is owned by one routine and is being exported to a program or another DLL. Similarly, you can use `DLLIMPORT` within a calling routine to tell the compiler that a common block is being imported from the routine in a program or DLL that owns it. One `DLLEXPORT`-`DLLIMPORT` link must be established for each common block shared between a program and a DLL or between DLLs.

The `DLLEXP2` sample shows how to use `DLLIMPORT` and `DLLEXPORT` with shared `COMMON` data. To build the DLL Sample `DLLEXP2` from the command line, see the makefile.

To find out how to declare the `DLLEXPORT` and `DLLIMPORT` attributes, see [DLLs](#) and `cDEC$ ATTRIBUTES`.

Building the DLL

When you first create a DLL, you follow the general steps described in [Defining Your Project](#). For the project type, choose Dynamic-Link Library (.DLL). Microsoft Developer Studio automatically selects the correct linker instructions for loading the proper run-time library routines (located in a DLL themselves). Your DLL is created as a multithread-enabled library.

Developer Studio takes the following actions in building a DLL:

- Looks for the **DLLMain** entry-point function in the source code
- Links with DLL start-up code that performs some initialization for you
- Produces an import library, *projectname.LIB*, to be linked to applications that call your DLL
- If you do not specify `/dll`, it interprets `/exe:file` (or `/Fe`) or `/link /out:file` as naming a .DLL rather than an .EXE file; the default file extension becomes *projectname.DLL* instead of

projectname.EXE

- Selects as the default the DLL run-time libraries to support multithreaded operation

If you build a DLL from the console command line or using a foreign makefile, you must include the /dll option.

The DLL Build Output

Your library routines are contained in the file *projectname*.DLL located in the default directory for your project, unless you specified another name and location. Your import library file is *projectname*.LIB, located in the default directory for your project.

Using DLLs

Add the import .LIB file with its path and library name to your main project. The file contains information that your program needs to work with the DLL.

For an application to access your dynamic-link library, *projectname*.DLL, it must be located in a directory on the search path or in the same directory as the main project. If you have more than one program accessing your DLL, you can keep it in a convenient directory identified in the environment path. If you have several DLLs, you can place them all in the same directory to avoid numerous revisions to the path specification.

Remember, when changing your path specification in Windows 95, you must restart the operating system for the change to take effect. In the Windows NT system, you should log out and back in after modifying the system path.

QuickWin Restrictions

You cannot make a QuickWin application into a DLL (see [Using QuickWin](#)) and QuickWin applications cannot be used with Fortran run-time routines in a DLL.

Errors During the Build Process

Compiler and linker errors are displayed in the Build pane of the output window. To quickly locate the source line causing the error, follow these steps:

1. Select (click) the error message text in the Build pane of the output window.
2. Press F4.

The editor window appears with a marker in the left margin that identifies the line causing the error.

If you need to set different compiler options for some of your source files, you can highlight the source file name and select the Project menu, Settings item. Options set in this manner are valid only for the file you selected.

After you have corrected any compiler errors reported during the previous build, choose Build from the Build menu. The build engine recompiles only those files that have changed, or which refer to changed include or module files. If all files in your project compile without errors, the build engine links the object files and libraries to create your program or library.

You can force the build engine to recompile all source files in the project by selecting Rebuild All from the Build menu. This is useful to verify that all of your source code is clean, especially if you are using a foreign makefile, or if you use a new set of compiler options for all of the files in your project.

▶ **To view the include file and library directory paths in Developer Studio:**

- In the Tools menu, click Options.
- Click the Directories tab.
- In the drop-down list for Show Directories For, select Include files and view the include file paths.
- In the drop-down list for Show Directories For, select Library files and view the library paths.
- Click OK if you have changed any information.

▶ **To view the libraries being passed to the linker in Developer Studio:**

- If not already open, open your Project Workspace (File menu, Open Workspace).
- In the Project menu, click on Settings.
- Click on the Link tab to view the list of Object/Library modules (General category).
- Click OK if you have changed any information.

With the Professional Edition, if you have trouble linking IMSL libraries, see also [Using the Libraries from Visual Fortran](#).

Running Fortran Applications

You can execute programs built with this version of Visual Fortran only on a computer running the Microsoft Windows 95® or Windows NT™ operating system (see the release notes for the Windows NT version number). You can run the programs from the command console, Start ... Program ... group, Windows Explorer, and Microsoft Developer Studio. Each program is treated as a protected user application with a private address space and environment variables. Because of this, your program cannot accidentally damage the address space of any other program running on the computer at the same time.

Environment variables defined for the current user are part of the environment that Windows sets up when you open the command console. You can change these variables and set others within the console session, but they are only valid during the current session.

If you run a program from the console, the operating system searches directories listed in the PATH user environment variable to find the executable file you have requested. You can also run your program by specifying the complete path of the .EXE file. If you are also using DLLs, they must be in the same directory as the .EXE file or in one specified in the path.

You can easily recover from most problems that may arise while executing your program. You can use the just-in-time debugging feature to debug your programs as they run outside of Developer Studio, if both of the following items have been set:

- In the Tools menu Options item, the Debug tab has the checkbox for Just-In Time debugging set.
- The FOR_IGNORE_EXCEPTIONS environment variable is set to TRUE.

If your program is multithreaded, Windows NT starts each thread on whichever processor is available at the time. On a computer with one processor, the threads all run in parallel, but not simultaneously; the single processor switches among them. On a computer with more than one processor, the threads can run simultaneously.

If you specified the `/fpscomp:filesfromcmd` option (Compatibility category in Project Settings, Fortran tab), the command line that executes the program can also include additional filenames to satisfy **OPEN** statements in your program in which the filename field has been left blank. The first filename on the command line is used for the first such **OPEN** statement executed, the second filename for the second **OPEN** statement, and so on. In Developer Studio, you can provide these filenames in the Project menu Settings item, Debug tab, in the Program Arguments text box.

Each filename on the command line (or in a Developer Studio dialog box) must be separated from the names around it by one or more spaces or tab characters. You can enclose each name in double quotation marks ("*filename*"), but this is not required unless the argument contains spaces or tabs. A null argument consists of an empty set of double quotation marks with no filename enclosed ("").

The following example runs the program MYPROG.EXE from the console:

```
MYPROG " " OUTPUT.DAT
```

Because the first filename argument is null, the first **OPEN** statement with a blank filename field produces the following message:

```
File name missing or blank - please enter file name<R>
UNIT number  ?
```

The *number* is the unit number specified in the **OPEN** statement. The filename OUTPUT.DAT is used for the second such **OPEN** statement executed. If additional **OPEN** statements with blank filename fields are executed, you will be prompted for more filenames. Programs built with the QuickWin library prompt for a file to open by presenting a dialog box in which you can browse for the file or type in the name of a new file to be created.

Run-time error messages are displayed in the console or in a dialog box depending upon the type of application you build. If you need to capture these messages, you can redirect *stderr* to a file. For example, to redirect run-time error messages from a program called BUGGY.EXE to a file called BUGLIST.TXT, you would use the following syntax:

```
BUGGY.EXE > BUGLIST.TXT
```

The redirection portion of the syntax must appear last on the command line. You can append the output to an existing file by using two greater-than signs (>>) instead of one. If the file does not exist, one is created.

For more information about:

- Locating the source of exceptions, see [Locating Run-Time Errors](#)
- Handling run-time errors with source changes, see [Methods of Handling Errors](#)
- Environment variables recognized during run-time, see [Run-Time Environment Variables](#)
- Each Visual Fortran run-time message, see [Run-Time Errors](#)
- Debugging, see "Debugger" in the [Developer Studio Environment User's Guide](#).

Porting Projects Between x86 and Alpha Platforms

► **To move an existing Visual Fortran project to another platform:**

1. Copy all project files to the new platform

Keep the folder/directory hierarchy intact by copying the entire project tree to the new computer. For example, if a project resides in the folder `\MyProjects\Projapp` on one computer, you can copy the contents of that directory, and all subdirectories, to the `\MyProjects\Projapp` directory on another computer. After copying all of the files, delete any `*.opt` files. These files are computer specific and should not be copied.

2. Specify new configurations

After you copy the files, opening the project reveals that the target platform is still set to the original platform. Although this is not obvious, you can tell this is so because the Build, Compile, and Execute options are grayed out in the Build menu. Before you can build the application on the new platform, you must first specify one or more new configurations for the project on the new platform.

To create Debug and Release targets for this project, you create a new configuration while running Visual Fortran on the new platform. The platform for a new configuration is assumed to be the current platform. For example, if you copy an x86 project to an Alpha system, and create a new configuration, the target platform can only be Alpha. You cannot specify another platform. This same behavior applies when moving projects between any two platforms.

To create a new project configuration:

- a. In the Configurations dialog box, click the Add button. The Add Project Configuration dialog box appears.
- b. In the Configuration box, type a new configuration name. The names do not matter, as long as they differ from existing configuration names.
- c. Select the configuration from which to copy the settings for this configuration and click OK. Usually, you will want to copy the settings from a similar configuration. For example, if this new configuration is a release configuration, you will usually copy settings from an existing release configuration.
- d. The Projects dialog box appears with the new project configuration.
Repeat the process as necessary to create as many configurations as you need.

3. Reset project options

Because not all settings are transportable across platforms, you should verify your project settings on the new platform. To verify your project settings:

- a. From the Project menu, choose Settings. The Project Settings dialog box appears.
- b. Review the tabs and categories to ensure that the project settings you want are selected. Pay special attention to the following items:
 - General Tab: Review the directories for intermediate and output files.
 - Custom Build Tab: Review for any custom commands that might change between platforms.
 - Fortran and Linker tabs - Nonstandard options in the original configurations must be replicated (as applicable) in the new configurations. As listed in Compiler

Options, certain options are supported only on *x86* or Alpha systems.

- Pre-link and Post-build Step tabs -- Review for any custom commands that might change between platforms.

Advanced Applications

With Visual Fortran, you can create full Windows applications. You can create the familiar Windows interface for your programs, complete with tool bars, pull-down menus, dialog boxes, and other features. You can include data entry and mouse control, and interaction with programs written in other languages or commercial programs such as Microsoft® Excel.

This section describes how you turn your Fortran programs into Windows applications. It covers the following topics:

- [Creating Windows Applications](#)
- [Dialog Boxes](#)
- [OpenGL Graphics](#)
- [DLLs](#)

Creating Windows Applications

With Visual Fortran, you can build Fortran applications that are also fully-featured Windows applications. The Win32 Application Programming Interface (API) provides sophisticated window management, memory management, graphics support, threading, security and networking. QuickWin allows you to easily build Windows applications, but accesses only a small subset of the available Win32 API features.

With full Windows programming you can:

- Package Fortran applications with a Windows Graphical User Interface (GUI).
- Access all available Windows Graphic Device Interface (GDI) calls with your Fortran applications. Win32 GDI uses a 32-bit coordinate system, allowing coordinates in the +/-2 GB range, and performs skewing, reflection, rotation and shearing.
- Access low-level system services, such as the registry, and virtual and shared memory functions, and access high-level system services, such as network functions, mailslots, and the MIDI Mapper.

When you access the Windows module, DFWIN.F90 with the **USE DFWIN** statement, all parameters and interfaces to Windows routines are made available to your Visual Fortran program. Specify the `/winapp` option to search the commonly used link libraries. If unresolved link references occur when using `/winapp`, consider adding the `\DF\INCLUDE\FULLAPI.F90` file to your project. This file contains search directives for almost all of the libraries needed.

To build your application as a Windows application in Microsoft Developer Studio, choose Win32 Application from the list of Project types when you open a new project. However, you can access Win32 APIs from any Fortran application, including console and QuickWin.

The following Windows application topics are discussed:

- [The Visual Fortran Windows Module](#)
- [Writing a Windows GDI Program](#)
- [Using Win32 with QuickWin](#)
- [Sample Fortran Windows Applications](#)

- Getting Help with Windows Programming

The Visual Fortran Windows Module

DFWIN.F90 is a Fortran version (a subset) of the Win32 WINDOWS.H header file. The correspondence of data types is given in the following table:

Win32 Data Type	Equivalent Fortran Data Type
BOOL, BOOLEAN	LOGICAL(4)
BYTE	BYTE
CHAR, CCHAR, UCHAR	CHARACTER
COLORREF	INTEGER(4)
DWORD, INT, LONG, ULONG	INTEGER(4)
SHORT, USHORT, WORD	INTEGER(2)
FLOAT	REAL(4)
All Handles	INTEGER(4)
All Pointers (LP*, P*)	INTEGER(4) (Integer Pointers)

The structures in WINDOWS.H have been converted to derived types in DFWIN.F90. Unions in structures are converted to union/maps within the derived type. Names of components are unchanged. Bit fields are converted to Fortran's INTEGER(4). Functions accessing bit fields are contained in the DFWIN.F90 module with names of the form:

structurename\$bitfieldname

These functions take an integer argument and return an integer. All bit fields are unsigned integers. The following is an example of the translation from Win32 structures to Fortran derived types.

WINDOWS.H Definition

```
typedef struct _LDT_ENTRY {
    WORD LimitLow;
    WORD BaseLow;
    union {
        struct {
            BYTE BaseMid;
            BYTE Flags1;
            BYTE Flags2;
            BYTE BaseHi;
        } Bytes;
        struct {
            DWORD BaseMid : 8;
            DWORD Type : 5;
            DWORD Opl : 2;
            DWORD Pres : 1;
            DWORD LimitHi : 4;
            DWORD Sys : 1;
            DWORD Reserved_0 : 1;
            DWORD Default_Big : 1;
            DWORD Granularity : 1;
            DWORD BaseHi : 8;
        } Bits;
    } HighWord;
} LDT_ENTRY, *PLDT_ENTRY;
```

Fortran Definition

```
type LDT_ENTRY$HIGHWORD_BYTES
    BYTE BaseMid
    BYTE Flags1
    BYTE Flags2
    BYTE BaseHi
end type

type LDT_ENTRY$HIGHWORD
    union
        map
            type( LDT_ENTRY$HIGHWORD_BYTES ) Bytes
        end map
        map
            INTEGER(4) Bits
        end map
    end union
end type

type LDT_ENTRY
    INTEGER(2) LimitLow
    INTEGER(2) BaseLow
    type(LDT_ENTRY$HIGHWORD) HighWord
end type

INTEGER(4) function LDT_ENTRY$BaseMid( Bits )
```



```

INTEGER(4) Bits
LDT_ENTRY$BaseMid = IAND( Bits, #ff
end
INTEGER(4) function LDT_ENTRY$Type( Bits )
INTEGER(4) Bits
LDT_ENTRY$Type = IAND( ISHFT( Bits, -8 ), #1f )
end
...

```

Note that `_LDT_ENTRY` and `PLDT_ENTRY` do not exist in the Fortran definition. Also note that `Bits.xxx` is not the same as the C version. In the Fortran case, the bit field functions must be used. For example, the C variable:

```
yyy.HighWord.Bits.BaseHi
```

is replaced with the Fortran variable:

```
LDT_ENTRY$BaseHi( yyy.HighWord.Bits )
```

All macros in the `WINDOWS.H` file are converted to functions in the `DFWIN.F90` module. The object modules that this conversion creates are in `DFWIN.LIB` in the `LIB` directory.

Writing a Windows GDI Program

To write a Windows GDI (Graphic Device Interface) subsystem program, the following function must be defined by the user:

```

      INTEGER(4) function WinMain ( hInstance, hPrevInstance,
&      lpszCmdLine, nCmdShow )
!DEC$ ATTRIBUTES STDCALL, ALIAS:'_WinMain@16' :: WinMain
      INTEGER(4), INTENT(IN) :: hInstance, hPrevInstance
      INTEGER(4), INTENT(IN) :: lpszCmdLine
      INTEGER(4), INTENT(IN) :: nCmdShow

```

In a program that includes a `WinMain` function, no program unit can be identified as the main program with the **PROGRAM** statement.

Using Win32 with QuickWin

You can convert the unit numbers of QuickWin windows to Win32 handles with the [GETHWNDQQ](#) QuickWin function. You should not use Windows GDI to draw on QuickWin windows because QuickWin keeps a window buffer and the altered window would be destroyed on redraw. You can use Windows subclassing to intercept graphics messages bound for QuickWin before QuickWin receives them.

See the sample program `POKER` in the `\DF\SAMPLES\GENERAL\POKER` subdirectory for a demonstration of this technique.

Sample Fortran Windows Applications

The `\DF\SAMPLES` subdirectory contains many Fortran Windows applications that demonstrate Windows functionality or a particular Win32 function. Each sample application is in separate folder.

Users unfamiliar with Windows programming should start by looking at the programs in \DF\SAMPLES.

Getting Help with Windows Programming

In InfoViewer, you can access the folder "Platform, SDK, and DDK Documentation." For information about using InfoViewer, see [Using InfoViewer](#).

The full Win32 API set is documented in the *Win32 Application Programming Interface for Windows NT Programmer's Reference*, available from Microsoft Press and also distributed as part of the Windows NT Software Development Kit.

Dialog Boxes

Visual Fortran gives you an easy way to create simple dialog boxes that can be used for data entry and application control. Dialogs are a user-friendly way to get and process input. As your application executes, you can make a dialog box appear on the screen and the user can click on a button or scroll bar to enter data or choose what happens next. You can add dialog boxes to any Fortran application, including Windows, QuickWin, and console applications.

You design your dialog with the Resource Editor, and drive them with a combination of the dialog functions, such as **DLGSET**, and your own subroutines. A complete discussion of how to design and use dialog boxes is given in [Using Dialogs](#).

OpenGL Graphics

OpenGL is a library of graphic functions that create sophisticated graphic displays such as 3-D images and animation. OpenGL is commonly available on workstations. Writing to this standard allows your program to be ported easily.

OpenGL windows are used independently of and in addition to any console, QuickWin and regular Windows windows your application uses. Every window in OpenGL uses a pixel format, and the pixels carry, among other things, RGB values, opacity values, and depth values so that pixels with a small depth (shallow) overwrite deeper pixels. The basic steps in creating OpenGL applications are:

- Specify the pixel format
- Specify how the pixels will be rendered on the video device
- Call OpenGL commands

OpenGL programming is straightforward, but requires a particular initialization and order, like other software tools. References to get you started are:

- *OpenGL Programming Guide, The Official Guide to Learning OpenGL, Release 1*, OpenGL Architecture Review Board, Addison Wesley, 1992, ISBN 0201-63274-8.
- OpenGL documentation in the Windows NT and Windows 95 Platform SDK in InfoViewer.
- The OpenGL description from Microsoft Visual C++ manuals.

Visual Fortran provides an OpenGL module, DFOPNGL.MOD, invoked with the **USE** statement.

When you use this module, all constants and interfaces that bind Fortran to the OpenGL routines become available. Any link libraries required to link with an OpenGL program are automatically searched if **USE DFOPNGL** is present in your Fortran program.

An OpenGL window can be opened from a console, Windows, or QuickWin application. The OpenGL window uses OpenGL calls exclusively, not normal Graphic Device Interface (GDI) calls. Likewise, OpenGL calls cannot be made within an ordinary Windows window or QuickWin child window, because special initialization is required for OpenGL calls.

The Fortran OpenGL identifiers are the same as the C identifiers, except that the gl prefix is changed to fgl, and the GL prefix is changed to FGL. The data types in the OpenGL C binding are translated to Fortran types as shown in the following table:

OpenGL/C Type	Fortran Data Type
GLbyte	INTEGER(1)
GLshort	INTEGER(2)
GLint, GLsizei	INTEGER(4)
GLfloat, GLclampf	REAL(4)
GLdouble, GLclampd	REAL(8)
GLubyte	INTEGER(1)
GLboolean	LOGICAL
GLushort	INTEGER(2)
GLuint, GLenum, GLbitfield	INTEGER(4)
GLvoid	not needed
pointers	INTEGER

OpenGL sample programs are available in the \DF\SAMPLES\ADVANCED subdirectory.

DLLs

A dynamic-link library (DLL) is an executable file, but is usually used as a library for applications. A DLL contains one or more functions that are compiled, linked and stored separately from the applications using them. The advantages of DLLs include:

- Multiple applications can access the same DLL--This reduces the overall amount of memory needed in the system, which results in fewer memory swaps to disk and improves performance.
- When general functions are placed in DLLs, the applications that share the DLLs can have very small executables.
- You can change the functions in a DLL without recompiling or relinking the applications that use them, as long as the functions' arguments and return types do not change. This allows you to upgrade your applications easily. For example, a display driver DLL can be modified to support a display that was not available when your application was created.
- Programs written in different languages can call the same DLL functions, as long as each program follows the functions' calling conventions.

Within your Fortran DLL, you use cDEC\$ ATTRIBUTES DLLEXPORT to declare that a function or

data is being exported to other applications. Within your Fortran application, you use `cDEC$ ATTRIBUTES DLLIMPORT` to declare that the function or data is being imported from a DLL.

For more information, see:

- [DLLEXPORT and DLLIMPORT Compiler Directive Options](#)
- [DLLEXPORT and DLLIMPORT in Modules](#)

DLLEXPORT and DLLIMPORT Compiler Directive Options

The DLLEXPORT and DLLIMPORT options (for the `ATTRIBUTES` directive) define a DLL's interface in the process that uses them. Declaring functions as DLLEXPORT eliminates the need for a module-definition (.DEF) file. You can also apply the DLLEXPORT and DLLIMPORT properties to data and to objects in modules (see [DLLEXPORT and DLLIMPORT in Modules](#)).

The DLLEXPORT property declares that functions or data are being exported to other applications or DLLs. The compiler produces the most efficient code when this option is used, and there is no need for module definition (.DEF) file to export symbols. If you declare a function or data with the DLLEXPORT property, the definition must appear in the same module of the same program. Otherwise, a linker error occurs.

A program that uses symbols defined in a DLL imports them. The DLL user needs to link with the import LIB and use the DLLIMPORT property inside the application that imports the symbol. The DLLIMPORT directive option is used in a declaration, not a definition, because you do not define the symbol you are importing.

Building a DLL is described in detail in [Dynamic-Link Library Projects in Building Programs and Libraries](#).

Fortran and C applications can call Fortran and C DLLs provided the calling conventions are consistent. Visual Basic applications can also call Fortran functions and subroutines in the form of DLLs. For example, the following Visual Basic code calls the Fortran subroutine ARRAYTEST:

```
Static arr(1 To 3, 1 To 7) As Single
Call ARRAYTEST(arr(1, 1))
```

The subroutine ARRAYTEST is defined in the following Fortran code:

```
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
```

If the Fortran code in the example is saved in the file `f90vb4.f90`, you can build a DLL in by selecting New from the File menu, choosing Projects from the New File list, then choosing Win32 Dynamic-Link Library as the Project type. Or you can compile from the command line with the following:

```
DF /dll f90vb4.f90
```

This code creates a DLL named `f90vb4.dll`. The DLL is declared in the Visual Basic `.BAS` file as follows:

```
Declare Sub ARRAYTEST Lib "f90vb4.dll" (Myarray As Single)
```

For details on how to call Fortran DLLs from Visual Basic, see [Programming with Mixed Languages](#).

Your Fortran application can access the same subroutine from the same DLL, as follows:

```
PROGRAM FORAPP
REAL r1, r2, xarray(3, 7)
...
INTERFACE
  SUBROUTINE ARRAYTEST (rarray)
    !DEC$ ATTRIBUTES DLLIMPORT :: ARRAYTEST
    REAL rarray(3, 7)
  END SUBROUTINE ARRAYTEST
END INTERFACE
...
CALL ARRAYTEST(xarray)
...
END
```

DLLEXPORT and DLLIMPORT in Modules

You can give objects in a module the `DLLEXPORT` property, in which case the object is exported from a DLL. When a module is used in other program units, through the `USE` statement, any objects in the module with the `DLLEXPORT` property are treated in the program using the module as if they were declared with the `DLLIMPORT` property. So, a main program that uses a module contained in a DLL has the correct import attributes for all objects exported from the DLL.

You can also give some objects in a module the `DLLIMPORT` property. Only procedure declarations in `INTERFACE` blocks and objects declared `EXTERNAL` or with `cDEC$ ATTRIBUTES EXTERN` can have the `DLLIMPORT` property. In this case, the objects are imported by any program unit using the module.

If you use a module that is part of a DLL and you use an object from that module that does not have the `DLLEXPORT` or `DLLIMPORT` property, the results are undefined.

Using COM and Automation Objects

Visual Fortran provides a wizard to simplify the use of Component Object Model (COM) and Automation (formerly called OLE Automation) objects. The Visual Fortran Module Wizard generates Fortran 90 modules that simplify calling COM and Automation services from Fortran programs. This Fortran code allows you to invoke routines in a dynamic link library, methods of an Automation object, and member functions of a Component Object Model (COM) object.

The following sections describe the use of COM and Automation objects with Visual Fortran:

- [The Role of the Module Wizard](#)
- [Using the Module Wizard to Generate Code](#)
- [Calling the Routines Generated by the Module Wizard](#)
- [Additional Information About COM and Automation Objects](#)

The Role of the Module Wizard

To use COM and Automation objects from a Fortran program, the following steps need to occur:

1. Find or install the object on the system. COM and Automation objects can be registered:
 - By other programs you install.
 - By creating the object yourself, for example, by using Visual C++ or Visual Basic.

For example, Developer Studio registers certain objects during installation (see the Developer Studio documentation on the Developer Studio object model).

Creating an object involves deciding what type of object and what type of interfaces or methods should be available. The object's server must be designed, coded, and tested like any other application. For general information about object creation and related information, see [Additional Information About COM and Automation Objects](#).

2. Determine:
 - Whether the object has a COM interface, Automation interface, or both.
 - Where the object's type information is located.

You should be able to obtain this information from the object's documentation. You can use the OLE-COM Object Viewer tool (provided in the Visual Fortran program folder) to determine the characteristics of an object on your system.

3. Use the Visual Fortran module wizard to generate code.

The Visual Fortran module wizard is a application that interactively asks certain questions about the object, including its name, type, and other information. The information collected by the module wizard is used in the generated code. To learn about using the Visual Fortran module wizard, see [Using the Module Wizard to Generate Code](#).

4. Write a Fortran 90 program to invoke the code generated by the Visual Fortran module

wizard.

To understand more about calling the interfaces and jacket routines created by the module wizard, see [Additional Information About COM and Automation Objects](#).

Using the Module Wizard to Generate Code

To run the Visual Fortran Module Wizard, choose the Tools menu item Fortran Module Wizard. The module wizard asks a series of questions, including the name and type of the object as well as certain characteristics. If you have not already obtained the object's characteristics, see [The Role of the Module Wizard](#).

The Visual Fortran Module Wizard presents a series of dialog boxes that allow you to select the type of information needed.

An object's type information contains programming language independent descriptions of the object's interfaces. Depending on the implementation of the object, type information can be obtained from the running object (see Automation Object below) or from a type library.

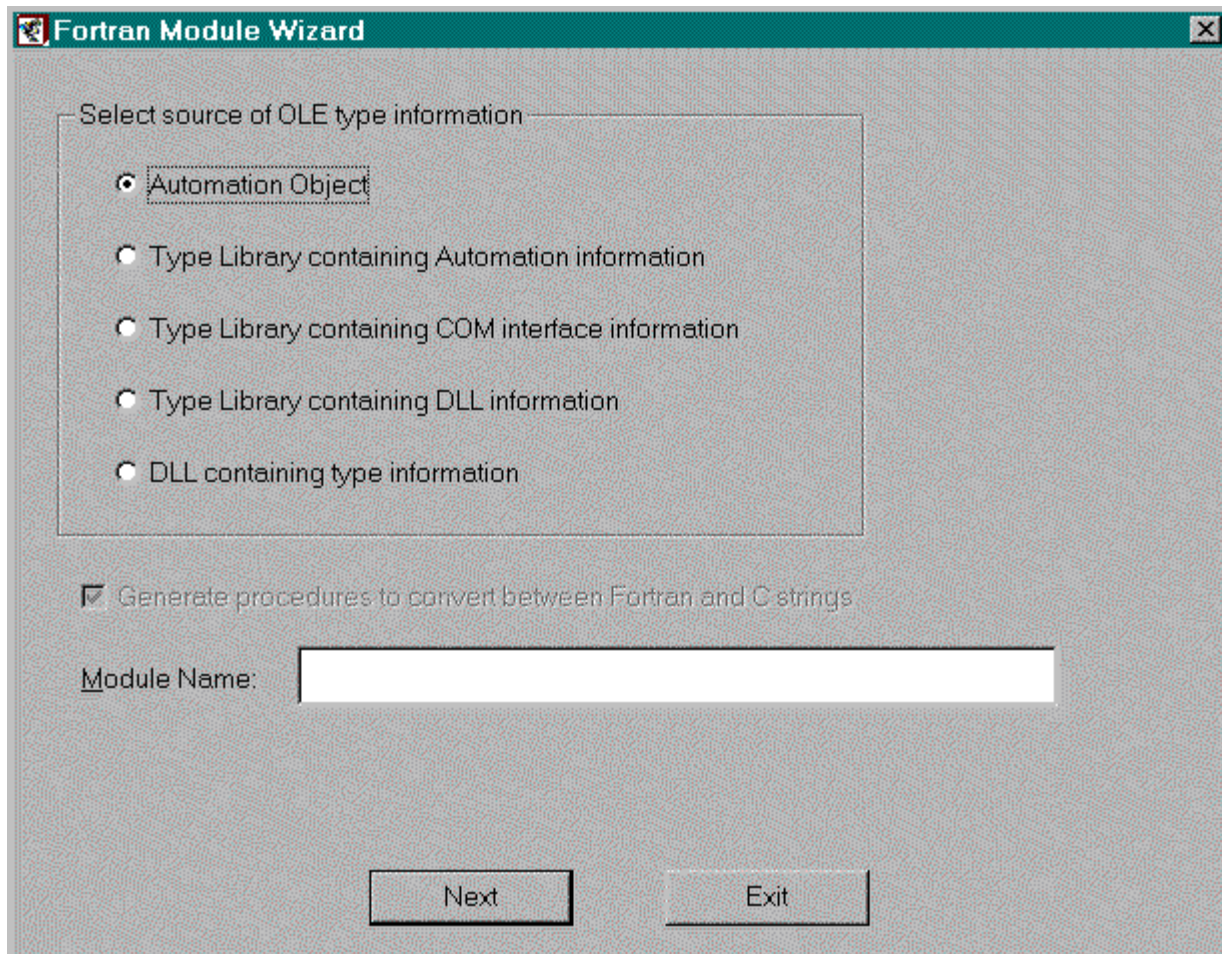
A type library is a collection of type information for any number of object classes, interfaces, and so on. A type library can also be used to describe the routines in a DLL. You can store a type library in a file of its own (usually with an extension of .TLB) or it can be part of another file. For example, the type library that describes a DLL can be stored in the DLL itself.

After you start the Module Wizard (Tools menu, Fortran Module Wizard), a dialog box requests the source of the type information that describes the object you need to use. You need to determine what type of object it is (or DLL) and how it makes its type information available. The choices are:

- Automation Object
- Type Library Containing Automation Information
- Type Library Containing COM Interface Information
- Type Library Containing DLL Information
- DLL Containing Type Information

The following initial screen appears after you select the Visual Fortran Module Wizard:

Figure: Initial Module Wizard Screen

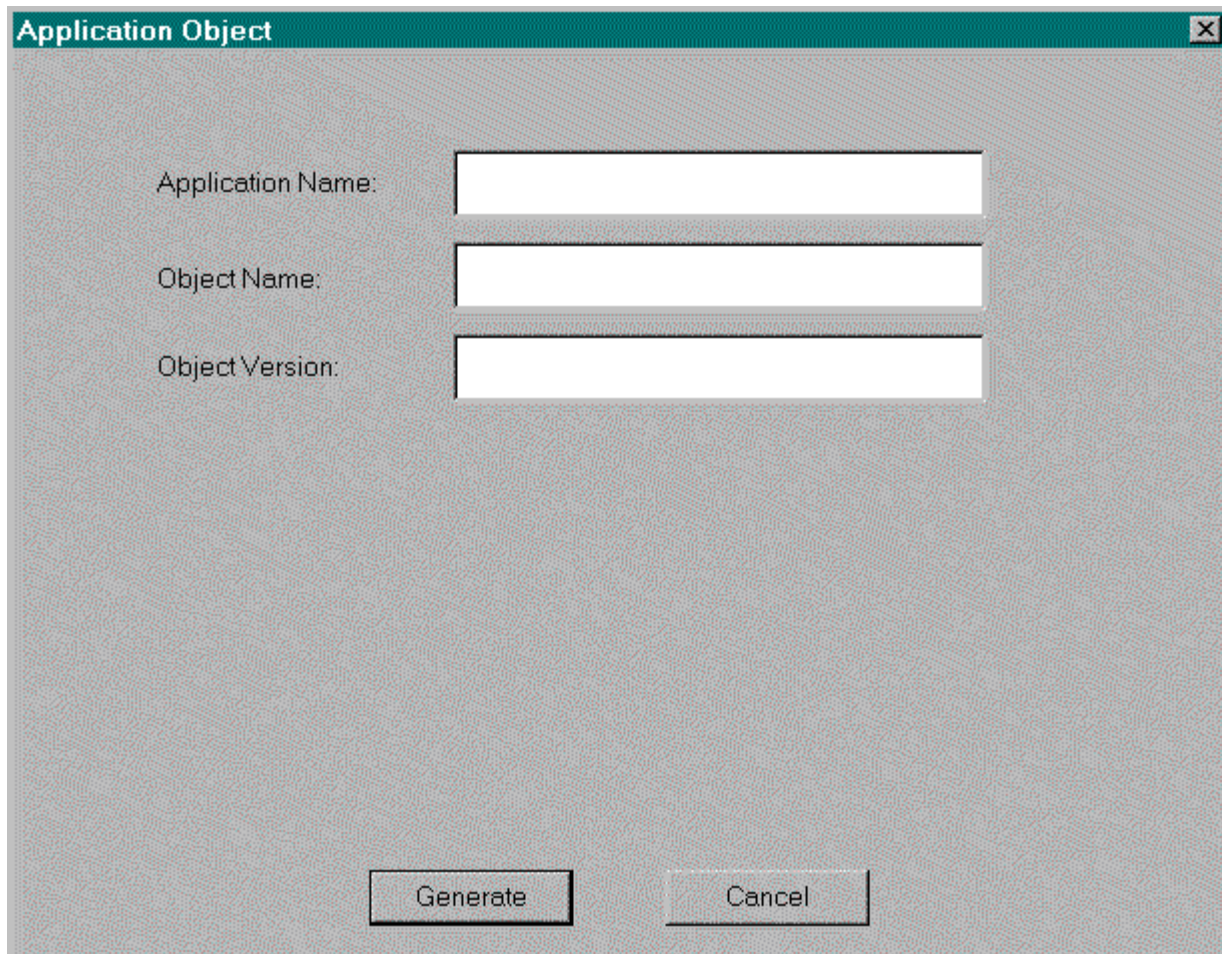


After you select one of the five choices, one of two different screens will appear depending on the selection made. The **Module Name** in the initial Module Wizard screen is used as the name of the Fortran module being generated. It is also used as the default file name of the generated source file.

If You Select Automation Object

If you select Automation Object, the following screen appears:

Figure: Application Object Screen



The image shows a dialog box titled "Application Object". It has a standard Windows-style title bar with a close button (X) in the top right corner. The main area of the dialog is light gray and contains three text input fields stacked vertically. The first field is labeled "Application Name:", the second is labeled "Object Name:", and the third is labeled "Object Version:". At the bottom of the dialog, there are two buttons: "Generate" on the left and "Cancel" on the right.

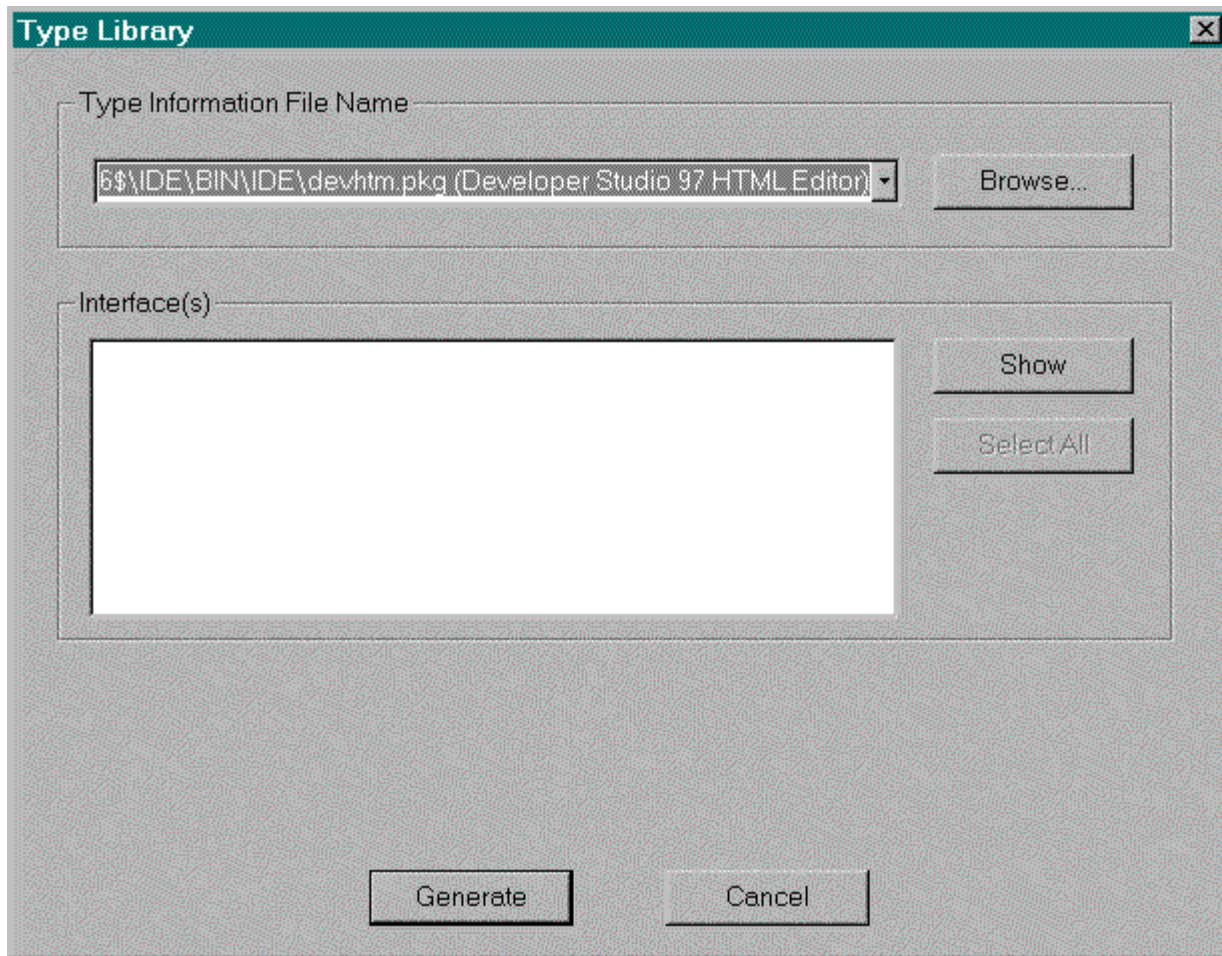
Microsoft recommends that object servers provide a type library. However some applications do not, but do provide type information dynamically when running. Use this option for such an application. Enter the name of the application, name of the object, and version number. The version number is optional. If you do not specify it, you will get the latest version of the object. Note that this method only works for objects that provide a programmatic identifier (ProgID). ProgIDs are entered into the system registry and identify, among other things, the executable program that is the object's server.

After entering the information and pressing the "Generate" button, the Fortran Module Wizard asks you for the name of the source file to be generated. It then asks COM to create an instance of the object identified by the ProgID that the wizard constructs using the supplied information. COM starts the object's server if it needs to do so. The wizard then asks the object for its type information and generates a file containing Fortran modules.

If You Select Other Options

After selecting any of the remaining options in the initial screen and press the "Next" button, the Module Wizard displays the following screen:

Figure: Type Library Screen



Choose the type library (or file containing the type library), and optionally specific components of the type library.

At the top of the dialog box is a combo box that lists all of the type libraries that have been registered with the system. You will notice a number of different file extensions, for example, .OLB (object libraries) and .OCX (ActiveX controls). Select a type library from the list or press "Browse" to find the file using the standard "Open" dialog box. Once you have selected a type library press the "Show" button to list the interfaces described in the type library. By default, the Fortran Module Wizard will use all of the interfaces. Optionally, you can select the ones desired from the list.

After entering the information and pressing the "Generate" button, the Fortran Module Wizard asks you for the name of the source file to be generated. It then asks COM to open the type library and generates a file containing Fortran modules.

Calling the Routines Generated by the Module Wizard

Although Fortran 90 does not support objects, it does provide Fortran 90 *modules*. A module is a set of declarations that are grouped together under a global name, and are made available to other program units by using the USE statement.

The Fortran Module Wizard generates a source file containing one or more modules. The types of information placed in the modules include:

- *Derived-type definitions* are Fortran equivalents of data structures that are found in the type information.
- *Procedure interface definitions* are Fortran interface blocks that describe the procedures found in the type information.
- *Procedure definitions* are Fortran functions and subroutines that are jacket routines for the procedures found in the type information.

The jacket routines make the external procedures easier to call from Fortran by handling data conversion and low-level invocation details.

The use of modules allows the Visual Fortran Module Wizard to encapsulate the data structures and procedures exposed by an object or DLL in a single place. You can then share these definitions in multiple Fortran programs.

The appropriate USE statement needs to be added in your program, as well as function invocations or subroutine calls.

The routines generated by the Visual Fortran Module Wizard are designed to be called from Fortran. These routines in turn call the appropriate system routines (not designed to be called from Fortran), thereby simplifying the coding needed to use COM and Automation objects.

Visual Fortran provides a set of run-time routines that present to the Fortran programmer a higher level abstraction of the COM and Automation functionality. The Fortran interfaces that the Wizard generates hide most of the differences between Automation objects and COM objects.

Depending on the options specified, the following routines can be present in the generated code, allowing you to call them to use COM or Automation objects:

DFCOM Routines (COMxxxx)	
<u>COMAddObjectReference</u>	Adds a reference to an object's interface.
<u>COMCLSIDFromProgID</u>	Passes a programmatic identifier and returns the corresponding class identifier.
<u>COMCLSIDFromString</u>	Passes a class identifier string and returns the corresponding class identifier.
<u>COMCreateObjectByGUID</u>	Passes a class identifier and creates an instance of an object. It returns a pointer to the object's interface.
<u>COMCreateObjectByProgID</u>	Passes a programmatic identifier and creates an instance of an object. It returns a pointer to the object's IDispatch interface.
<u>COMGetActiveObjectByGUID</u>	Pass a class identifier and returns a pointer to the interface of a currently active object.
<u>COMGetActiveObjectByProgID</u>	Passes a programmatic identifier and returns a pointer to the IDispatch interface of a currently active object.
<u>COMInitialize</u>	Initializes the COM library. You must initialize the library before calling any other COM or AUTO routine.
<u>COMGetFileObject</u>	Passes a file name and returns a pointer to the IDispatch interface of an Automation object that can manipulate the file.
<u>COMQueryInterface</u>	Passes an interface identifier and it returns a pointer to an object's interface.
<u>COMReleaseObject</u>	Indicates that the program is done with a reference to an object's

<u>COMReleaseObject</u>	interface.
<u>COMUninitialize</u>	Uninitializes the COM library. This must be the last COM routine that you call.
DFAUTO Automation Routines (AUTOxxxx)	
<u>AUTOAddArg</u>	Passes an argument name and value and adds the argument to the argument list data structure.
<u>AUTOAllocateInvokeArgs</u>	Allocates an argument list data structure that holds the arguments that you will pass to AUTOInvoke.
<u>AUTODeallocateInvokeArgs</u>	Deallocates an argument list data structure.
<u>AUTOGetExceptInfo</u>	Retrieves the exception information when a method has returned an exception status.
<u>AUTOGetProperty</u>	Passes the name or identifier of the property and gets the value of the Automation object's property.
<u>AUTOGetPropertyByID</u>	Passes the member ID of the property and gets the value of the Automation object's property into the argument list's first argument.
<u>AUTOGetPropertyInvokeArgs</u>	Passes an argument list data structure and gets the value of the Automation object's property specified in the argument list's first argument.
<u>AUTOInvoke</u>	Passes the name or identifier of an object's method and an argument list data structure. It invokes the method with the passed arguments.
<u>AUTOSetProperty</u>	Passes the name or identifier of the property and a value. It sets the value of the Automation object's property.
<u>AUTOSetPropertyByID</u>	Passes the member ID of the property and sets the value of the Automation object's property using the argument list's first argument.
<u>AUTOSetPropertyInvokeArgs</u>	Passes an argument list data structure and sets the value of the Automation object's property specified in the argument list's first argument.

Visual Fortran provides two sample applications in the Advanced folder of Samples (. . . \DF\SAMPLES\ADVANCED\COM\) that demonstrate the use of the Fortran Module Wizard. They are:

- DSLINES uses COM to drive Microsoft Developer Studio to edit a Fortran source file and convert Debug lines (column 1) to IFDEF directives.
- DSBUILD uses OLE Automation to drive Microsoft Developer Studio to rebuild a project configuration.

Example of Generated Code Used by the DSLINES Sample

The DLINES Sample contains the code that invokes this and other Microsoft Developer Studio functionality using COM interfaces.

The following code shows an annotated version of the code generated by the Fortran Module Wizard from the COM type information in . . . \DevStudio\SharedIDE\Bin\devsh1.dll. This type information describes the top-level objects in the Microsoft Developer Studio object model.

```

INTERFACE
! Saves the document to disk. 1

INTEGER*4 FUNCTION IGenericDocument_Save($OBJECT, vFilename, &
                                         vBoolPrompt, pSaved) 2
USE DFCOMTY
INTEGER*4, INTENT(IN)  :: $OBJECT      ! Object Pointer

!DEC$ ATTRIBUTES VALUE :: $OBJECT 3

TYPE (VARIANT), INTENT(IN)  :: vFilename ! (Optional Arg) 4
!DEC$ ATTRIBUTES VALUE     :: vFilename
TYPE (VARIANT), INTENT(IN)  :: vBoolPrompt ! (Optional Arg)
!DEC$ ATTRIBUTES VALUE     :: vBoolPrompt

INTEGER*4, INTENT(OUT)      :: pSaved      ! Void 5
!DEC$ ATTRIBUTES REFERENCE :: pSaved
!DEC$ ATTRIBUTES STDCALL   :: IGenericDocument_Save
END FUNCTION IGenericDocument_Save
END INTERFACE

POINTER(IGenericDocument_Save_PTR, IGenericDocument_Save) ! routine pointer 6

```

Notes for this example:

- 1**If the type information provides a comment that describes the member function, then the comment is placed before the beginning of the procedure.
- 2**The first argument to the procedure is always \$OBJECT. It is a pointer to the object's interface. The remaining argument names are determined from the type information.
- 3**This is an example of an **ATTRIBUTE** directive statement used to specify the calling convention of an argument.
- 4**A **VARIANT** is a data structure that can contain any type of Automation data. It contains a field that identifies the type of data and a union that holds the data value. The use of a **VARIANT** argument allows the caller to use any data type that can be converted into the data type expected by the member function.
- 5**Nearly every COM member function returns a status of type **HRESULT**. Because of this, if a COM member function produces output it uses output arguments to return the values. In this example, the "pSaved" argument returns a routine specific status value.
- 6**The interface of a COM member function looks very similar to the interface for a dynamic link library function with one major exception. Unlike a DLL function, the address of a COM member function is never known at program link time. You must get a pointer to an object's interface at run-time, and the address of a particular member function is computed from that.

The following code shows an annotated version of the wrapper generated by the Fortran Module Wizard for the "Save" function. The name of a wrapper is the same as the name of the corresponding member function, prefixed with a "\$" character.

```
! Saves the document to disk.
```

```

INTEGER*4 FUNCTION $IGenericDocument_Save($OBJECT, vFilename, & 1
                                vBoolPrompt, pSaved)
!DEC$ ATTRIBUTES DLLEXPORT      :: $IGenericDocument_Save
IMPLICIT NONE

INTEGER*4, INTENT(IN) :: $OBJECT      ! Object Pointer
!DEC$ ATTRIBUTES VALUE        :: $OBJECT
TYPE (VARIANT), INTENT(IN), OPTIONAL :: vFilename
!DEC$ ATTRIBUTES REFERENCE    :: vFilename
TYPE (VARIANT), INTENT(IN), OPTIONAL :: vBoolPrompt
!DEC$ ATTRIBUTES REFERENCE    :: vBoolPrompt
INTEGER*4, INTENT(OUT)        :: pSaved ! Void
!DEC$ ATTRIBUTES REFERENCE    :: pSaved

INTEGER*4 $RETURN
INTEGER*4 $VTBL ! Interface Function Table 2
POINTER($VPtr, $VTBL)
TYPE (VARIANT) :: $VAR_vFilename
TYPE (VARIANT) :: $VAR_vBoolPrompt
IF (PRESENT(vFilename)) THEN 3
    $VAR_vFilename = vFilename
ELSE
    $VAR_vFilename = OPTIONAL_VARIANT
END IF
IF (PRESENT(vBoolPrompt)) THEN
    $VAR_vBoolPrompt = vBoolPrompt
ELSE
    $VAR_vBoolPrompt = OPTIONAL_VARIANT
END IF
$VPtr = $OBJECT ! Interface Function Table 4
$VPtr = $VTBL + 84 ! Add routine table offset
IGenericDocument_Save_PTR = $VTBL
$RETURN = IGenericDocument_Save($OBJECT, $VAR_vFilename, &
    $VAR_vBoolPrompt, pSaved)
$IGenericDocument_Save = $RETURN
END FUNCTION $IGenericDocument_Save

```

Notes for this example:

- 1** The wrapper takes the same argument names as the member function interface.
- 2** The wrapper computes the address of the member function from the interface pointer and an offset found in the interface's type information. In implementation terms, an interface pointer is a pointer to a pointer to an array of function pointers called an "Interface Function Table".
- 3** Arguments to a COM or Automation routine can be optional. The wrapper handles the invocation details for specifying an optional argument that is not present in the call.
- 4** The offset of the "Save" member function is 84. The code assigns the computed address to the function pointer IGenericDocument_Save_PTR, which was declared in the previous example, and then calls the function.

The DLINES Sample contains the code that invokes this and other Microsoft Developer Studio functionality using COM interfaces.

Example of Generated Code Used by the DSBUILD Sample

The DSBUILD example contains the code that invokes this and other Microsoft Developer Studio functionality using Automation interfaces.

The following code shows an annotated version of the code generated by the Fortran Module Wizard from the Automation type information in ... \DevStudio\SharedIDE\Bin\devshl.dll.

```

! Rebuilds all files in a specified configuration.
SUBROUTINE IApplication_RebuildAll($OBJECT, Configuration, $STATUS) 1
!DEC$ ATTRIBUTES DLLEXPORT :: IApplication_RebuildAll
IMPLICIT NONE

INTEGER*4, INTENT(IN)                :: $OBJECT          ! Object Pointer
!DEC$ ATTRIBUTES VALUE                :: $OBJECT
TYPE (VARIANT), INTENT(IN), OPTIONAL :: Configuration
!DEC$ ATTRIBUTES REFERENCE            :: Configuration
INTEGER*4, INTENT(OUT), OPTIONAL     :: $STATUS         ! Method status
!DEC$ ATTRIBUTES REFERENCE            :: $STATUS
INTEGER*4 $$$STATUS
INTEGER*4 invokeargs
invokeargs = AUTOALLOCATEINVOKEARGS() 2
IF (PRESENT(Configuration)) CALL AUTOADDARG(invokeargs, '$ARG1', &
                                           Configuration, .FALSE.)
$$$STATUS = AUTOINVOKE($OBJECT, 28, invokeargs) 3
IF (PRESENT($STATUS)) $STATUS = $$$STATUS 4
CALL AUTODEALLOCATEINVOKEARGS (invokeargs) 5
END SUBROUTINE IApplication_RebuildAll

```

Notes for this example:

1 The first argument to the procedure is always \$OBJECT. It is a pointer to an Automation object's IDispatch interface. The last argument to the procedure is always \$STATUS. It is an optional argument that you can specify if you wish to examine the return status of the method. The IDispatch Invoke member function returns a status of type HRESULT. An HRESULT is a 32-bit value. It has the same structure as a Win32 error code. In between the \$OBJECT and \$STATUS arguments are the method arguments' names determined from the type information. Sometimes, the type information does not provide a name for an argument. The Fortran Module Wizard creates a "\$ARGn" name in this case.

2 AUTOAllocateInvokeArgs allocates a data structure that is used to collect the arguments that you will pass to the method. AUTOAddArg adds an argument to this data structure.

3 AUTOInvoke invokes the named method passing the argument list. This returns a status result.

4 If the caller supplied a status argument, the code copies the status result to it.

5 AUTODEallocateInvokeArgs deallocates the memory used by the argument list data structure.

The DSBUILD example contains the code that invokes this and other Microsoft Developer Studio functionality using Automation interfaces.

Additional Information About COM and Automation Objects

This section provides some information about COM and Automation objects.

COM Objects

The Component Object Model (COM) provides mechanisms for creating reusable software components. COM is an object-based programming model designed to promote software interoperability; that is, to allow two or more applications or "components" to easily cooperate with one another, even if they were written by different vendors at different times, in different programming languages, or if they are running on different machines running different operating systems.

With COM, components interact with each other and with the system through collections of function calls, also known as methods or member functions or requests, called *interfaces*. An interface is a semantically related set of member functions. The interface as a whole represents a features of an object. The member functions of an interface represent the operations that make up the feature. In general, an object can support multiple interfaces and you can use COMQueryInterface to get a pointer to any of them.

The Visual Fortran COM routines provide a Fortran interface to basic COM functions.

Automation Objects

The capabilities of an *Automation object* resemble those of a COM object. An Automation object is in fact a COM object. An Automation object exposes:

- *Methods*, which are functions that perform an action on an object. These are very similar to the member functions of COM objects.
- *Properties*, which hold information about the state of an object. A property can be represented by a pair of methods; one for getting the property's current value, and one for setting the property's value.

The Visual Fortran AUTO routines provide a Fortran interface to invoking an automation object's methods and setting and getting its properties.

Object Identification

Object identification enables the use of COM objects created by disparate groups of developers. To provide a method of uniquely identifying an object class regardless of where it came from, COM uses *globally unique identifiers* (GUIDs). A GUID is a 16-byte integer value that is guaranteed (for all practical purposes) to be unique across space and time. COM uses GUIDs to identify object classes, interfaces, and other things that require unique identification.

To create an instance of an object, you need to tell COM what the GUID of the object is. While using 16-byte integers for identification is fine for computers, it poses a challenge for the typical developer. So, COM also supports the use of a less precise, textual name called a *programmatic identifier* (ProgID). A ProgID takes the form:

```
application_name.object_name.object_version
```

Additional Resources

There have been a number of published books and articles about COM and Automation. DIGITAL

lists these additional resources for the sole purpose of assisting customers who want to learn more about the subject matter. This list does not comment--either negatively or positively--on any documents listed or not yet listed. Books and related resources include:

- *How OLE and COM Solve the Problems of Component Software Design* by K. Brockschmidt. Microsoft Systems Journal, vol. 11, no. 5 (May 1996): 63-80
- *Inside OLE, Second Edition* by K. Brockschmidt. Published by Microsoft Press (Redmond, Washington) 1995
- Microsoft Developer Studio Environment User's Guide (InfoViewer, provided with Visual Fortran)
- *OLE 2 Programmer's Reference, Volume Two*. Published by Microsoft Press (Redmond, Washington) 1994
- *Understanding ActiveX and OLE* by David Chappell. Published by Microsoft Press (Redmond, Washington) 1996
- *Win 32 SDK, OLE Programmer's Reference* online version
- *Win 32 SDK, Automation* online version

Microsoft Press has a URL of <http://mspress.microsoft.com/>.

Using the Compiler and Linker from the Command Line

The DF command is used to compile and link your programs. In most cases, you will use a single DF command to invoke the compiler and linker. The DF command invokes a *driver program* that is the actual user interface to the compiler and linker. It accepts a list of command options and file names and causes one or more of the processors to process each file.

If you will be using the DF command from the command line, you can set the appropriate environment variables for the terminal window environment by executing the file DFVARS.BAT or by using the supplied F90 terminal window (in the Visual Fortran program folder).

The driver program does the following:

- Calls the DIGITAL Visual Fortran compiler to process Fortran files
- Passes the linker options to the linker
- Passes object files created by the compiler to the linker
- Passes libraries to the linker

The DF command automatically references the appropriate Visual Fortran Run-Time Libraries when it invokes the linker. Therefore, to link one or more object files created by the Visual Fortran compiler, you should use the DF command instead of the linker command.

Because the DF driver calls other software components, error messages may be returned by these other components. For instance, the linker may return a message if it cannot resolve a global reference. Using the [/watch:cmd](#) option on the DF command line can help clarify which component is generating the error.

The following sections discuss these topics:

- [The Format of the DF Command](#)
- [Examples of the DF Command Format](#)
- [Input and Output Files](#)
- [Environment Variables Used with the DF Command](#)
- [Specifying Project Types with DF Command Options](#)
- [Using the DF Command to Compile and Link](#)
- [DF Indirect Command File Use](#)
- [Compiler Limits and Messages](#)

The Format of the DF Command

This section describes the format of the DF command. It also provides an alphabetical list of DF command options.

The DF command accepts the following types of options:

- Compiler options
- Linker options

The command driver requires that the following rules be observed when specifying the DF command:

- Except for the linker options, options can be specified in any order.
- Linker options must be preceded by the keyword `/link` and must be specified at the end of the command line, following all other options.

The DF command has the following form:

```
DF options [/link options]
```

options

A list of compiler or linker options. These lists of options take the following form:

```
[/option:[arg]] [filename.ext]...
```

Where:

/option[:arg]

Indicates either special actions to be performed by the compiler or linker, or special properties of input or output files.

The following rules apply to options and their names:

- Options begin with a slash (/). You can use a dash (-) instead of a slash.
- Visual Fortran options are *not* case-sensitive. Certain options provided for compatibility with Microsoft Fortran Powerstation options *are* case-sensitive, such as `/FA` and `/Fafile`.
- You can abbreviate option names. You need only enter as many characters as are needed to uniquely identify the option.

Certain options accept one or more keyword arguments following the option name. For example, the `/warn` option accepts several keywords, including `argument_checking` and `declarations`.

To specify only a single keyword, specify the keyword after the colon (:). For example, the following specifies the `/warn` option `declarations` keyword:

```
DF /warn:declarations test.f90
```

To specify multiple keywords, specify the option name once, and place each keyword in a comma-separated list enclosed within parentheses with no spaces between keywords, as follows:

```
DF /warn:(argument_checking,declarations) test.f90
```

Instead of the colon, you can use an equal sign (=):

```
DF /warn=(argument_checking,declarations) test.f90
```

filename.ext

Specifies the files to be processed. You can use wildcard characters (such as `*.f90`) to indicate multiple files or you can specify each file name.

The file extension identifies the type of the file. With Fortran source files, certain file extensions indicate whether that source file contains source code in free (such as `.f90`) or fixed (such as `.for`) source form. You can also specify compiler options to indicate fixed or free source form (see

/[no]free).

The file extension determines whether that file gets passed to the compiler or to the linker. For example, files myfile.for and projfile.f are passed to the compiler and file myobj.obj is passed to the linker.

The following table lists the DF command options:

Table: Options List, Alphabetic Order

<u>/[no]alignment</u>	<u>/[no]altparam</u>
<u>/architecture (Alpha only)</u>	<u>/[no]asmattributes</u>
<u>/[no]asmfile</u>	<u>/assume</u>
<u>/[no]automatic</u>	<u>/bintext</u>
<u>/[no]browser</u>	<u>/[no]check</u>
<u>/[no]comments</u>	<u>/[no]compile_only</u>
<u>/convert</u>	<u>/[no]d_lines</u>
<u>/[no]dbglibs</u>	<u>/[no]debug</u>
<u>/define</u>	<u>/[no]dll</u>
<u>/[no]error_limit</u>	<u>/[no]exe</u>
<u>/[no]extend_source</u>	<u>/extfor</u>
<u>/extfpp</u>	<u>/extlnk</u>
<u>/[no]f66</u>	<u>/[no]f77rtl</u>
<u>/fast</u>	<u>/[no]fixed</u>
<u>/[no]fltconsistency (x86 only)</u>	<u>/[no]fpconstant</u>
<u>/fpe</u>	<u>/fpp</u>
<u>/[no]fpscomp</u>	<u>/[no]free</u>
<u>/granularity (Alpha only)</u>	<u>/help or /?</u>
<u>/iface</u>	<u>/[no]include</u>
<u>/[no]inline</u>	<u>/[no]intconstant</u>
<u>/integer_size</u>	<u>/[no]keep</u>
<u>/[no]libdir</u>	<u>/libs</u>
<u>/[no]link</u>	<u>/[no]list</u>
<u>/[no]logo</u>	<u>/[no]machine_code</u>
<u>/[no]map</u>	<u>/math_library</u>
<u>/[no]module</u>	<u>/names</u>
<u>/[no]object</u>	<u>/[no]optimize</u>
<u>/[no]pad_source</u>	<u>/[no]pdbfile</u>
<u>/[no]pipeline (Alpha only)</u>	<u>/preprocess_only</u>
<u>/real_size</u>	<u>/[no]recursive</u>
<u>/[no]reentrancy</u>	<u>/rounding_mode (Alpha only)</u>
<u>/[no]show</u>	<u>/source</u>
<u>/[no]static</u>	<u>/[no]stand</u>
<u>/[no]synchronous_exceptions (Alpha only)</u>	<u>/[no]syntax_only</u>
<u>/[no]threads</u>	<u>/[no]transform_loops (Alpha only)</u>

<u>/tune</u> (Alpha only)	<u>/undefine</u>
<u>/unroll</u>	<u>/[no]vms</u>
<u>/[no]warn</u>	<u>/[no]watch</u>
<u>/what</u>	<u>/winapp</u>

For More Information:

- On DF command examples, see [Examples of the DF Command Format](#)
- On using the FL32 command, see [Microsoft Fortran PowerStation Command-Line Compatibility](#)
- About Fortran Powerstation options (such as /MD) and their DF command equivalents, see [Equivalent Visual Fortran Compiler Options](#)

Examples of the DF Command Format

The following examples demonstrate valid and invalid DF commands:

Valid DF commands

In the following example, the file to be compiled is test.f90 and the file proj.obj is passed to the linker:

```
DF test.f90 proj.obj
```

In this example, the .f90 file extension indicates test.f90 is a Fortran free-form source file to be compiled. The file extension of obj indicates proj.obj is an object file to be passed to the linker. You can optionally add the [/link](#) option before the file proj.obj to indicate it should be passed directly to the linker.

In the following example, the [/check:bounds](#) option requests that the Fortran compiler generate additional code to perform run-time checking for out-of-bounds array and substring references for the files myfile.for and test.for (fixed-form source):

```
DF /check:bounds myfile.for test.for
```

In the following example, the [/link](#) option indicates that files and options after the [/link](#) option are passed directly to the linker:

```
DF myfile.for /link myobject.obj /out:myprog.exe
```

Invalid DF commands

The following DF command is invalid because the [/link](#) option indicates that items after the [/link](#) option are passed directly to the linker, but the file test.for should be passed to the compiler:

```
DF myfile.for /link test.for /out:myprog.exe
```

The following DF command is invalid because the [/link](#) option is missing and the [/out](#) linker option is not recognized as a compiler option:

```
DF myfile.for test.for /out:myprog.exe
```

A correct form of this command is:

```
DF myfile.for test.for /link /out:myprog.exe
```

In this case, you can alternatively use one of the DF options (/exe) that specifies information to the linker:

```
DF myfile.for test.for /exe:myprog.exe
```

For More Information:

- [Environment Variables Used with the DF Command](#)
- [Specifying Project Types with DF Command Options](#)
- [Using the DF Command to Compile and Link](#)
- [DF Indirect Command File Use](#)
- [Compiler Limits and Messages](#)

Input and Output Files

You can use the DF command to process multiple files. These files can be source files, object files, or object libraries.

When a file is not in your path or working directory, specify the directory path before the file name.

The file extension determines whether a file gets passed to the compiler or to the linker. The following types of files are used with the DF command:

- Files passed to the compiler: .f90, .for, .f, .fpp, .i, .i90, .inc, .h, .c, .cpp, .fi, .fd, .f77
Typical Fortran (DF command) source files have a file extension of .f90, .for, and .f.
- Files passed to the linker: .lib, .obj, .o, .exe, .res, .rbj, .def, .dll
For example, object files usually have a file extension of .obj. Files with extensions of .lib are usually library files.

The output produced by the DF command includes:

- An object file (.OBJ) if you specify the /compile_only, /keep, or /object option on the command line.
- An executable file (.EXE) if you do not specify the **/compile_only** option
- A dynamic-link library file (.DLL) if you specify the /dll option and do not specify the **/compile_only** option
- A module file (.MOD) if a source file being compiled defines a Fortran 90 module (MODULE statement)
- A program database file (.PDB) if you specify the /pdbfile or /debug:full (or equivalent) options
- A listing file (.LST) if you specify the /list option
- A browser file (.SBR) if you specify the /browser option

You control the production of these files by specifying the appropriate options on the DF command line. Unless you specify the **/compile_only** option or **/keep** option, the compiler generates a single

temporary object file from one or more source files. The linker is then invoked to link the object file into one executable image file.

If fatal errors are encountered during compilation, or if you specify certain options such as **/compile_only**, linking does not occur.

For more information about naming input and output files, see:

- [Naming Output Files](#)
- [Temporary Files](#)

Naming Output Files

To specify a file name for the executable image file, you can use one of several DF options:

- The **/exe:file** or the **/out:file** linker option to name an executable program file.
- The **/dll:file** alone or the **/dll** option with the **/out:file** linker option to name an executable dynamic-link library.

You can also use the **/object:file** option to specify the object file name. If you specify the **/compile_only** option and omit the **/object:file** option, each source file is compiled into a separate object file. For more information about the output file(s) created by compiling and linking multiple files, see [Compiling and Linking Multiple Fortran Source Files](#).

Many compiler options allow you to specify the name of the *file* being created. If you specify only a filename without an extension, a default extension is added for the file being created, as summarized below:

Option	Default File Extension
/asmfile:file	.ASM
/browser:file	.SBR
/dll:file	.DLL
/exe:file	.EXE
/list:file	.LST
/map:file	.MAP
/pdbfile:file	.PDB (default filename is df50.pdb)

Temporary Files

Temporary files created by the compiler or linker reside in the directory used by the operating system to store temporary files. To store temporary files, the operating system first checks for the TMP environment variable.

If the TMP environment variable is defined, the directory that it points to is used for temporary files. If the TMP environment variable is not defined, the operating system checks for the TEMP environment variable. If the TEMP environment variable is not defined, the current working directory is used for temporary files. Temporary files are usually deleted, unless the **/keep** option was specified. For performance reasons, use a local drive (rather than using a network drive) to contain the temporary files.

To view the file name and directory where each temporary file is created, use the `/watch:cmd` option. To create object files in your current working directory, use the `/compile_only` or `/keep` option. Any object files (.obj files) that you specify on the DF command line are retained.

Environment Variables Used with the DF Command

The following environment variables affect the DF command:

Environment Variable	Description
PATH	The PATH environment variable sets the search path.
LIB	The linker uses the LIB environment variable to determine the location of .LIB files. If the LIB environment variable is not set, the linker looks for .LIB files in the current directory.
IMSL_F90	The IMSL_F90 environment variable contains a list of libraries used for linking IMSL libraries (Professional Edition), as listed in Library Naming Conventions .
INCLUDE	The make facility (NMAKE) uses the INCLUDE environment variable to locate INCLUDE files and module files. The DIGITAL Fortran compiler uses the INCLUDE environment variable to locate files included by an INCLUDE statement or module files referenced by a USE statement. Similarly, the resource compiler uses the INCLUDE environment variable to locate #include and RCINCLUDE files.
DF	The DF environment variable can be used to specify frequently used DF options and files. The options and files specified by the DF environment variable are added to the DF command; they are processed before any options specified on the command line. You can override an option specified in the DF environment variable by specifying an option on the command line. For information about using the DF environment variable to specify frequently-used options, see Using the DF Environment Variable to Specify Options .

You can set these environment variables by using the DFVARS.BAT file or the F90 command-line window (see [Using the Command-Line Interface](#)).

For a list of environment variables recognized at run-time, see [Run-Time Environment Variables](#).

Specifying Project Types with DF Command Options

This section provides the DF command options that correspond to Developer Studio project types.

When creating an application, you should choose a *project type*. The first four projects are main project types, requiring main programs:

- To create a [Win32 Console Application](#) with the DF command, you do not need to specify a specific option. This is the default project type created.
- To create a [Standard Graphics Application](#) with the DF command, specify the `/libs=qwins` option.
- To create a [QuickWin Application](#) with the DF command, specify the `/libs:qwin` option.
- To create a [Win32 \(Windows\) Application](#) with the DF command, specify the `/winapp` option.

The following types are library projects, without main programs:

- To create a Win32 Static library with the DF command, specify the /libs:static and /compile_only options to create the object files. Use the LIB command to create the library (see Managing Libraries with LIB).
- To create a Win32 Dynamic-Link Library with the DF command, specify the /dll option (sets the /libs:dll option).

For an introduction to Visual Fortran project types and corresponding sample programs, see Visual Fortran Projects.

Using the DF Command to Compile and Link

By default, when you use DF, your source files are compiled and then linked. To suppress linking, use the /compile_only option. The following topics show how to use the DF command:

- Compiling and Linking a Single Source File
- Using the DF Environment Variable to Specify Options
- Compiling, but not Linking, a Fortran Source File
- Compiling and Linking Multiple Fortran Source Files
- Generating a Listing File
- Linking Against Additional Libraries
- Linking Object Files
- Compiling and Linking for Debugging
- Compiling and Linking for Optimization
- Compiling and Linking Mixed-Language Programs

Compiling and Linking a Single Source File

The following command compiles x.for, links, and creates an executable file named x.exe. This command generates a temporary object file, which is deleted after linking:

```
DF x.for
```

To name the executable file, specify the /exe option:

```
DF x.for /exe:myprog.exe
```

Alternatively, you can name the executable file by using linker /out option:

```
DF x.for /link /out:myprog.exe
```

Using the DF Environment Variable to Specify Options

The following command-line sequences show the use of the DF environment variable. In the first command sequence, the SET command sets the DF environment variable. When the DF command is invoked, it uses the options specified by the DF environment variable, in this case, **/debug:minimal** and **/list**:

```
set DF=/debug:minimal /list
DF myprog.for
```

You can also specify additional options on the DF command line. In the following command sequence, the SET command sets the DF environment variable. The DF options specified are **/debug:minimal** and **/list**.

```
set DF=/debug:minimal /list
DF myprog.for /show:map
```

If the options specified on the command line conflict with the options specified by the DF environment variable, the option specified on the command line takes precedence. In the following command sequence, the **/debug:minimal** option specified by the DF environment variable is overridden by the **/debug:none** option specified on the command line:

```
set DF=/debug:minimal /list
DF myprog.for /debug:none
```

Compiling, but not Linking, a Fortran Source File

The following command compiles x.for and generates the object file x.obj. The /compile_only option prevents linking (it does not link the object file into an executable file):

```
DF x.for /compile_only
```

Compiling and Linking Multiple Fortran Source Files

The following command compiles a.for, b.for, and c.for. It creates a single temporary object file, then links the object file into an executable file named a.exe:

```
DF a.for b.for c.for
```

If the files a.for, b.for, and c.for were the only .for files in the current directory, you could use a wildcard character to similarly compile the three source files:

```
DF *.for
```

If you use the /compile_only option to prevent linking, also use the /object:file option so that multiple sources files are compiled into a single object file, allowing more optimizations to occur:

```
DF /compile_only /object:a.obj a.for b.for c.for
```

When you use modules and compile multiple files, compile the source files that define modules *before* the files that reference (USE) the modules.

If you use multiple DF commands, compile the files that define module definitions first and then compile the files that reference the modules.

When you use a single DF command, the order in which files are placed on the command line is significant. For example, if the free-form source file `moddef.f90` defines the modules referenced by the file `projmain.f90`, use the following DF command line:

```
DF moddef.f90 projmain.f90
```

Generating a Listing File

To request a listing file, specify the `/list` option with the DF command. When you request a listing file, a separate listing file is generated for each object file created.

The content of the listing file is affected by the `/show` option. For more information about this option, see [Compiler and Linker Options](#).

The following command compiles and links `a.for`, `b.for`, and `c.for`. It generates one listing file for the three source files:

```
DF a.for b.for c.for /list
```

The following command compiles `a.for`, `b.for`, and `c.for`. It generates three listing files (and three object files) for the three source files:

```
DF a.for b.for c.for /list /compile_only
```

The following command sequence compiles and links `a.for`, `b.for`, and `c.for`. It generates one named object file (`a.obj`) and one listing file (`a.lst`). The second command links the object files into an executable file (`a.exe`):

```
DF a.for b.for c.for /list /compile_only /object:a.obj  
DF a.obj
```

The following command sequence compiles and links `a.for`, `b.for`, and `c.for`. It generates three object files (`a.obj`, `b.obj`, and `c.obj`) and three listing files (`a.lst`, `b.lst`, and `c.lst`). The second command links the object files into an executable file (`a.exe`).

```
DF a.for b.for c.for /list /compile_only  
DF a.obj b.obj c.obj
```

Linking Against Additional Libraries

By default, the DF command automatically adds the libraries needed to build a console application to the link command that it generates. The `/libs:dll` option indicates that you want to link against single-threaded DLLs; other `/libs` options allow you to link against other types of libraries. The `/libs:static` option (the default) indicates that you want to link against single-threaded static libraries. You can link against additional libraries by listing those libraries on the command line.

For example, the following command links against static libraries. In addition to linking against the default libraries, it links against `mylib.lib`:

```
DF x.f90 mylib.lib
```

The following command links against single-threaded DLLs:

```
DF x.f90 /libs:dll
```

The following command links against single-threaded DLLs. It links against the default libraries and mylib.lib:

```
DF x.f90 /libs:dll mylib.lib
```

For more information on the types of libraries available to link against, see the [/libs](#) and [/winapp](#) options.

Linking Object Files

The following command links x.obj into an executable file. This command automatically links with the default DIGITAL Fortran libraries:

```
DF x.obj
```

Compiling and Linking for Debugging

If you use a single DF command to compile and link, specify the [/debug](#) option ([/debug](#) sets the default optimization level to [/optimize:0](#)), as follows:

```
DF x.for /debug
```

By default, the debugger symbol table information is created in a PDB file, which is needed for the debugger integrated with Developer Studio.

If you use separate DF commands to compile and link, you will want to specify the same debugging information level for the compiler and the linker. For example, if you specify [/debug:minimal](#) to the compiler, you will also specify [/link /debug:minimal](#). The following command sequence compiles and then links x.for for debugging with the integrated Developer Studio debugger:

```
DF x.for /debug:full /optimize:0 /compile_only  
DF x.obj /debug:full
```

Compiling and Linking for Optimization

If you omit both the [/compile_only](#) and the [/keep](#) options, the specified Fortran source files are compiled together into a single object module and then linked. (The object file is deleted after linking.) Because all the Fortran source files are compiled together into a single object module, full interprocedural optimizations can occur. With the DF command, the default optimization level is [/optimize:4](#) (unless you specify [/debug](#) with no keyword).

If you specify the **/compile_only** or **/keep** option and you want to allow full interprocedural optimizations to occur, you should also specify the **/object** option. The combination of the **/compile_only** and **/object:file** options creates a single object file from multiple Fortran source files, allowing full interprocedural optimizations. The object file can be linked later.

The following command uses both the **/compile_only** and **/object** options to allow interprocedural optimization (explicitly requested by the **/optimize:4** option):

```
DF /compile_only /object:out.obj /optimize:4 ax.for bx.for cx.for
```

If you specify the **/compile_only** or **/keep** option without specifying the **/object** option, each source file is compiled into an object file. This is acceptable if you specified no optimization (**/optimize:0**) or local optimization (**/optimize:1**). An information message appears when you specify multiple input files and specify an option that creates multiple object files (such as **/compile_only** without **/object**) and specify or imply global optimization (**/optimize:2** or higher optimization level).

If you specify the **/compile_only** option, you must link the object file (or files) later by using a separate DF command. You might do this using a makefile processed by the NMAKE command for incremental compilation of a large application.

However, keep in mind that either omitting the **/compile_only** or **/keep** option or using the **/compile_only** option with the **/object:file** option provides the benefit of full interprocedural optimizations for compiling multiple Fortran source files.

Other optimization options are summarized in [Software Environment and Efficient Compilation](#).

Compiling and Linking Mixed-Language Programs

Your application can contain both C and Fortran source files. If your main program is a Fortran source file (myprog.for) that calls a routine written in C (cfunc.c), you could use the following sequence of commands to build your application:

```
cl -c cfunc.c
DF myprog.for cfunc.obj /link /out:myprog.exe
```

The cl command (invokes the C compiler) compiles but does not link cfunc.c. The -c option specifies that the linker is not called. This command creates cfunc.obj. The DF command compiles myprog.for and links cfunc.obj with the object file created from myprog.for to create myprog.exe.

For more information about compiling and linking Visual Fortran and Visual C++® programs, and the libraries used, see [Visual Fortran/Visual C++ Mixed-Language Programs](#).

DF Indirect Command File Use

The DF command allows the use of indirect command files. For example, assume the file text.txt contains the following:

```
/pdbfile:testout.pdb /exe:testout.exe /debug:full /optimize:0 test.f90 rest.f90
```

The following DF command executes the contents of file text.txt as an indirect command file to

create a debugging version of the executable program and its associated PDB file:

```
DF @test.txt
```

Indirect command files do not use continuation characters; all lines are appended together as one command.

Compiler Limits and Messages

The following table lists the limits to the size and complexity of a single DIGITAL Visual Fortran program unit and to individual statements contained in it.

The amount of data storage, the size of arrays, and the total size of executable programs are limited only by the amount of process virtual address space available, as determined by system parameters:

Language Element	Limit
Actual number of arguments per CALL or function reference	255
Array dimensions	7
Array elements per dimension	2,147,483,647 or process limit
Constants; character and Hollerith	2000 characters
Constants; characters read in list-directed I/O	2048 characters
Continuation lines	99
DO and block IF statement nesting (combined)	128
DO loop index variable	2,147,483,647 or process limit
Format group nesting	8
Fortran source line length	132 characters
INCLUDE file nesting	10
Labels in computed or assigned GOTO list	500
Lexical tokens per statement	3000
Named common blocks	250
Parentheses nesting in expressions	40
Structure nesting	20
Symbolic name length	31 characters

For more information about compiler limits and messages, see:

- [Compiler Diagnostic Messages and Error Conditions](#)
- [Linker Diagnostic Messages and Error Conditions](#)

Compiler Diagnostic Messages and Error Conditions

The Visual Fortran compiler identifies syntax errors and violations of language rules in the source program. If the compiler finds any errors, it writes messages to the standard error output file and any listing file. If you enter the DF command interactively, the messages are displayed.

Compiler messages have the following format:

```
filename(n) : severity: message-text
```

```
      [text-in-error]  
-----^
```

The pointer (---^) indicates the exact place on the source program line where the error was found. The following error message shows the format and message text in a listing file when an **END DO** statement was omitted:

```
echar.for(7): Severe: Unclosed DO loop or IF block  
      DO I=1,5  
-----^
```

Diagnostic messages usually provide enough information for you to determine the cause of an error and correct it.

When using the command line, make sure that the appropriate environment variables have been set by executing the `DFVARS.BAT` file (see [Using the Command-Line Interface](#) in *Getting Started*). For example, this BAT file sets the environment variables for the include directory paths.

For errors related to `INCLUDE` and module (`USE` statement) file use, see [/\[no\]include](#).

For a list of environment variables used by the `DF` command during compilation, see [Environment Variables Used with the DF Command](#).

To view the passes as they execute on the `DF` command line, specify [/watch:cmd](#) or [/watch:all](#).

Linker Diagnostic Messages and Error Conditions

If the linker detects any errors while linking object modules, it displays messages about their cause and severity. If any errors occur, the linker does not produce an executable file.

Linker messages are descriptive, and you do not normally need additional information to determine the specific error.

To view the libraries being passed to the linker on the `DF` command line, specify [/watch:cmd](#) or [/watch:all](#).

On the command line, make sure the `DFVARS.BAT` file was executed to set the appropriate environment variables (see [Using the Command-Line Interface](#) in *Getting Started*). For example, this BAT file sets the environment variables for the library directory paths. For a list of environment variables used by the `DF` command during compilation, see [Environment Variables Used with the DF Command](#).

For information on handling build errors in Developer Studio, see [Errors During the Build Process](#).

Compiler and Linker Options

Most of the compiler and linker options can be specified on the command line or within the Microsoft Developer Studio environment. This section contains a description of the options available to you in building programs.

You can set compiler options from:

- The DF command line. Compiler options must precede the /LINK option.
- Within Microsoft Developer Studio, by using the Fortran tab in the Project menu, Settings dialog box.

Unless you specify certain options, the DF command line will both compile and link the files you specify. To compile without linking, specify the /compile_only (or equivalent) option.

After the /LINK option on the DF command line, you can specify [linker options](#). Linker options and any libraries specified get passed directly to the linker, such as /NODEFAULTLIB. If you choose to use separate compile and link commands, you can also specify linker options on a separate LINK command.

For compatibility information, see [Microsoft Fortran Powerstation Command-Line Compatibility](#).

Compiler Options

This section describes the compiler options and how they are used. It includes the following topics:

- [Categories of compiler options](#), according to functional grouping.
- Descriptions of each compiler option, [listed alphabetically](#).

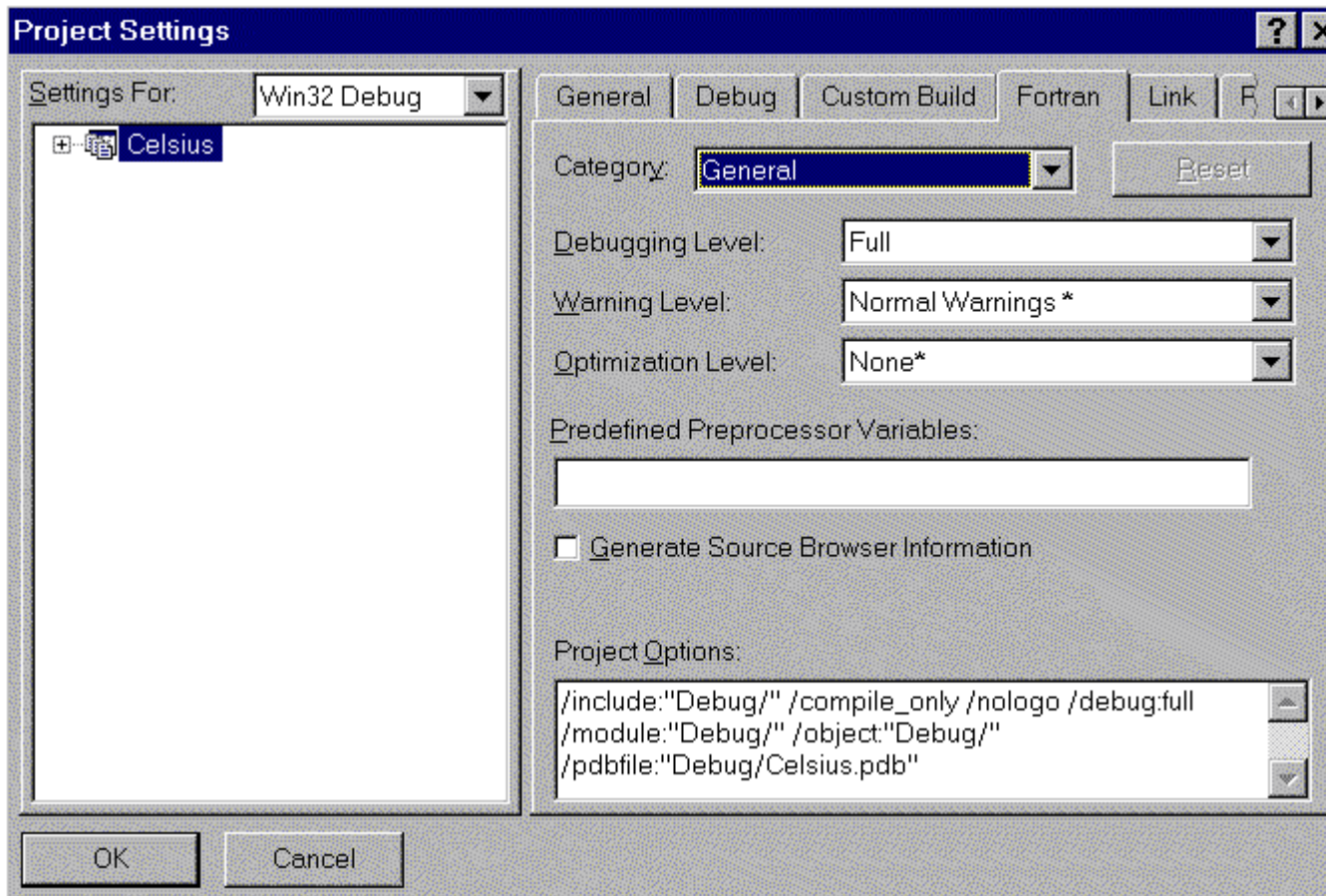
Categories of Compiler Options

If you will be using the compiler and linker from the command line, specify the options needed on the DF command line (as described in [Using the Compiler and Linker from the Command Line](#)).

You can use the functional categories of options below to locate the options needed for your application.

If you will be using the compiler and linker from the Developer Studio Environment, select the options needed by using the various tabs in the Project menu Settings item (see [Building Programs and Libraries](#)). The following graphic shows a sample Fortran tab:

Figure: Project Settings, Fortran Tab



The options are grouped under functional categories to help you locate the options needed for your application. From the Fortran tab, you can select one of the following categories from the Category drop-down list:

- General Compatibility
- Debug External Procedures
- Floating Point Fortran Data
- Fortran Language Library
- Listing Files Miscellaneous Compilation
- Optimizations Preprocessor
- Run-Time Checking

If a compiler option is not available in the dialog boxes, you can enter the option in the lower part of the window just as you would at the command line.

The following tables list the Visual Fortran compiler options by category in the Fortran tab:

General-Purpose Options	
Debugging Level	<u>/[no]debug and /[no]pdbfile[:file]</u>
Warning Level	<u>/[no]warn[:keyword]</u>
Optimization Level	<u>/[no]optimize</u>
Generate Source Browser File	<u>/[no]browser[:file]</u>
Predefined Preprocessor Variables	<u>/define</u>

Compatibility Options (See also Fortran Data)

Unformatted Files (Nonnative Data)	<u>/convert</u> (also see <u>/assume:[no]byterecl</u>)
VMS	<u>/[no]vms</u>
Select F77 Run-Time Library	<u>/[no]f77rtl</u>
Integer Constant Kind Used	<u>/[no]intconstant</u>
Microsoft Fortran Powerstation V4	<u>/[no]fpscomp</u> (various keywords)

Debug Options	
Debugging Level	<u>/[no]debug</u>
Program Database for Debug Information and File Name	<u>/[no]pdbfile[:file]</u>
Compile Lines With D in Column 1	<u>/[no]d_lines</u>

External Procedures (and Argument Passing) Options	
Default Calling Conventions	<u>/[no]iface:keyword</u>
String Length Argument Passing	<u>/[no]iface:mixed_str_len_arg</u>
Names Case Interpretation	<u>/names:keyword</u>
Append Underscore to External Names	<u>/assume:[no]underscore</u>

Floating-Point Options (See also Optimizations)	
Floating-Point Exception Handling	<u>/fpe</u>
Rounding Mode	<u>/rounding_mode</u> (Alpha only)
Enable Synchronous Floating-Point Exceptions	<u>/[no]synchronous_exceptions</u> (Alpha only)
Enable Floating-Point Consistency	<u>/[no]fltconsistency</u> (x86 only)
Extended Precision of Single-Precision Constants	<u>/[no]fpconstant</u>

Fortran Data Options (See also Compatibility)	
Common Element Alignment	<u>/[no]alignment:[no]common</u>
Structure Element Alignment (Derived Type and Record Data)	<u>/alignment:[no]records</u>
Thread Access Granularity	<u>/granularity</u> (Alpha only)
Default INTEGER and LOGICAL Kind	<u>/integer_size</u>
Default REAL and COMPLEX Kind	<u>/real_size</u>
Variables Default to Automatic (or Static Storage)	<u>/[no]automatic</u> or <u>/[no]static</u>
Use Bytes as Unit for Unformatted Files	<u>/assume:[no]byterecl</u>
Enable Dummy Argument Aliasing	<u>/assume:[no]dummy_aliases</u>

Fortran Language Options	
Standards Checking (None, Fortran 90, or Fortran 95)	<u>/stand:keyword</u> (also see <u>/warn:stderrs</u>)
Enable FORTRAN 66 Semantics	<u>/[no]f66</u>
Enable Alternate PARAMETER Syntax	<u>/[no]altparam</u>
Name Case Interpretation	<u>/names</u>
Source Form (File Extension, Fixed Form, or Free Form)	<u>/[no]free</u> or <u>/[no]fixed</u>
Fixed-Form Line Length	<u>/[no]extend_source</u>
Pad Fixed-Form Source Records	<u>/[no]pad_source</u>

Library Options (See also External Procedures)	
Use Fortran Run-Time Libraries	<u>/libs:keyword</u>

Use Multi-Threaded Library	<u>/[no]threads</u>
Enable (or Disable) Reentrancy Support	<u>/[no]reentrancy</u>
Use Common Windows Libraries	<u>/winapp</u>
Create Dynamic Link Library (DLL)	<u>/dll</u>
Disable OBJCOMMENT Library Names in Object	<u>/libdir:nouser</u>
Disable Default Library Search Rules	<u>/libdir:noauto</u>
Use Debug C Libraries	<u>/[no]dbglibs</u>

Listing and Assembly File Options	
Source Listing	<u>/[no]list</u>
Contents of Source Listing File	<u>/show:keyword...</u> or <u>/[no]machine_code</u>
Assembly Listing	<u>/[no]asmfile [:file]</u> and <u>/[no]asmattributes</u>

Miscellaneous Compilation Options	
Compilation Error Limit	<u>/[no]error_limit</u>
Directory to Place Modules	<u>/module[:file]</u>
Insert <i>string</i> into object file	<u>/bintext:string</u>
Control Warning Messages	<u>/[no]warn</u>
Compile, Do Not Link	<u>/compile_only</u> or <u>/c</u> (command line only)
Name of Executable Program or DLL File	<u>/[no]exe[:file]</u> (command line only)
Name of Object File	<u>/[no]object[:file]</u> (command line only)
Perfrom Syntax Check Only (No Object File)	<u>/[no]syntax_only</u> (command line only)
Display Help Text File	<u>/help</u> or <u>/?</u> (command line only)
Specify Custom File Extension for Compiler	<u>/extfor</u> (command line only)
Specify Custom File Extension for Linker	<u>/extlnk</u> (command line only)
Display Copyright and Compiler Version	<u>/nologo</u> and <u>/what</u> (command line only)
Display Compilation Details	<u>/[no]watch</u> (command line only)
Specify Linker Options (after /link)	<u>/link</u> (Use Linker tab)
Generate Link Map	<u>/[no]map</u> (use Linker tab)

Optimization Options	
Optimization Level	<u>/[no]optimize</u>
Inlining Procedures	<u>/[no]inline</u>
Loop Unrolling	<u>/unroll</u>
Reorder Floating-Point Calculations	<u>/assume:[no]accuracy_sensitive</u>
Loop Transformations	<u>/[no]transform_loops</u> (Alpha only)
Math Library: Checking or Fast Performance	<u>/math_library</u>
Software Instruction Scheduling	<u>/[no]pipeline</u> (Alpha only)
Code Generation for Alpha Chip	<u>/architecture</u> (Alpha only)
Code Tuning for Alpha Chip	<u>/tune</u> (Alpha only)

Preprocessor Options	
Define Preprocessor Symbols	<u>/define</u>
Default INCLUDE and USE Path	<u>/assume:[no]source_include</u>
Custom INCLUDE and USE Path	<u>/[no]include</u>

Write C-Style Comments	<code>/comments</code> (command line only)
Pass Options to FPP As Is	<code>/fpp</code> (command line only)
Specify Custom File Extension for Preprocessor	<code>/extfpp</code> (command line only)
Only Preprocess FPP Files	<code>/preprocess_only</code> (command line only)
Undefine Preprocessor Symbols	<code>/undefine</code> (command line only)

Run-Time Checking and Recursion	
Enable Recursive Routines	<code>/[no]recursive</code>
Check Array and String Bounds	<code>/check:[no]bounds</code>
Check Integer Overflow	<code>/check:[no]overflow</code>
Check Edit Descriptor Data Type	<code>/check:[no]format</code>
Check for Flawed Pentium Chip	<code>/check:[no]flawed_pentium</code> (x86 only)
Check Power Expressions	<code>/check:[no]power</code> (Alpha only)
Check Floating-Point Underflow	<code>/check:[no]underflow</code>
Check Edit Descriptor Data Size	<code>/check:[no]output_conversion</code>

For a table of DF command options listed alphabetically, see [Options List, Alphabetic Order](#).

`/[no]alignment`

Syntax:

`/alignment[:keyword...]`, `/noalignment`, or `/Zpn`

The `/alignment` option specifies the alignment of data items in common blocks, [record structures](#), and derived-type structures. The `/Zpn` option specifies the alignment of data items in derived-type or [record structures](#).

The `/alignment` options are:

- `/align:[no]commons`
The `/align:commons` option aligns the data items of all **COMMON** data blocks on natural boundaries up to four bytes. The default is `/align:nocommons` (unless `/fast` is specified), which does not align data blocks on natural boundaries. In Developer Studio, specify the Common Element Alignment as 4 in the Fortran Data [Compiler Option Category](#).
- `/align:dcommons`
The `/align:dcommons` option aligns the data items of all **COMMON** data blocks on natural boundaries up to eight bytes. The default is `/align:nocommons` (unless `/fast` is specified), which does not align data blocks on natural boundaries. In Developer Studio, specify the Common Element Alignment as 8 in the Fortran Data [Compiler Option Category](#).
- `/align:[no]records`
The `/align:records` option (the default, unless you specify the `/vms` option) requests that components of derived types and fields of [records](#) be aligned on natural boundaries up to 8 bytes. The `/align:norecords` option (the default if the `/vms` option is specified) requests that components and fields be aligned on arbitrary byte boundaries, instead of on natural boundaries up to 8 bytes. In Developer Studio, specify the Structure Element Alignment in the Fortran Data [Compiler Option Category](#).
- `/align:recNbyte` or `/Zpn`

The `/align:recNbyte` or `/Zpn` options requests that fields of **records** and components of derived types be aligned on the smaller of:

- The size byte boundary (*N*) specified.
- The boundary that will naturally align them.

Specifying `/align:recNbyte`, `/Zpn`, or `/align:[no]records` does not affect whether common block fields are naturally aligned or packed. In Developer Studio, specify the Structure Element Alignment in the Fortran Data Compiler Option Category.

Specifying	Is the Same as Specifying
<code>/Zp</code>	<code>/alignment:records</code> or <code>/align:rec8byte</code>
<code>/Zp1</code>	<code>/alignment:norecords</code> or <code>/align:rec1byte</code>
<code>/Zp2</code>	<code>/align:rec2byte</code>
<code>/Zp4</code>	<code>/align:rec4byte</code>
<code>/alignment</code>	<code>/Zp8</code> with <code>/align:dcommons</code> , <code>/alignment:all</code> , or <code>/alignment:(dcommons,records)</code>
<code>/noalignment</code>	<code>/Zp1</code> , <code>/alignment:none</code> , or <code>/alignment:(noccommons,nodcommons,norecords)</code>
<code>/align:rec1byte</code>	<code>/align:norecords</code>
<code>/align:rec8byte</code>	<code>/align:records</code>

When you omit the `/alignment` option, **records** and components of derived types are naturally aligned, but fields in common blocks are packed. This default is equivalent to:

```
/alignment=(noccommons,nodcommons,records)
```

`/[no]altparam`

Syntax:

`/altparam`, `/noaltparam`, `/4Yaltparam`, or `/4Naltparam`

The `/altparam` option determines how the compiler will treat the alternate syntax for PARAMETER statements, which is:

```
PARAMETER par1=exp1 [, par2=exp2...]
```

This form does not have parentheses around the assignment of the constant to the parameter name. With this form, the type of the parameter is determined by the type of the expression being assigned to it and not by any implicit typing.

In Developer Studio, specify the Enable Alternate PARAMETER Syntax in the Fortran Language Compiler Option Category.

When the `/[no]altparam` or equivalent options are not specified, the compiler default will be to allow the alternate syntax for **PARAMETER** statements (`/altparam`).

To disallow use of this form, specify `/noaltparam` or `/4Naltparam`. To allow use of this form, allow the default or specify `/altparam` or `/4Yaltparam`.

`/architecture (Alpha only)`

Syntax:

/architecture:keyword

The `/architecture (/arch)` option determines the type of Alpha chip code that will be generated for this program. The `/arch:keyword` option uses the same keywords as the `/tune:keyword` option. This option is ignored on x86 processor systems.

Whereas the `/tune:keyword` option is primarily used by certain higher-level optimizations for instruction scheduling purposes, the `/arch:keyword` option determines the type of code instructions generated for the program unit being compiled.

All Alpha processors implement a core set of instructions. Certain Alpha processor versions include additional instruction extensions.

In Developer Studio, specify the Generate Code For in the Optimizations Compiler Option Category. Supported `/arch` keywords are as follows:

- `/arch:generic`
Generates code that is appropriate for all Alpha processor generations. This is the default. Programs compiled with the generic keyword will run on all implementations of the Alpha architecture.
- `/arch:host`
Generates code for the processor generation in use on the system being used for compilation. Depending on the host system used, the program may or may not run on other Alpha processor generations:
 - Programs compiled on an ev4 or ev5 chip Alpha system with the host keyword will run on all Alpha processor generations.
 - Programs compiled on an ev56 chip system with the host keyword should *not* be run on ev4 and ev5 processors.
 - Programs compiled on a pca56 chip system with the host keyword should *not* be run on ev4, ev5, or ev56 processors.
- `/arch:ev4`
Generates code for the 21064, 21064A, 21066, and 21068 implementations of the Alpha architecture. Programs compiled with the ev4 keyword will run on all Alpha processor generations.
- `/arch:ev5`
Generates code for the some 21164 chip implementations of the Alpha architecture that use only the base set of Alpha instructions (no extensions). Programs compiled with the ev5 keyword will run on all Alpha processor generations.
- `/arch:ev56`
Generates code for some 21164 chip implementations that use the byte and word manipulation instruction extensions of the Alpha architecture. Programs compiled with the ev56 keyword will run correctly on ev56 and pca56 processors, but should *not* be run on ev4 and ev5 processors.

- `/arch:pca56`
Generates code for the 21164PC chip implementation that uses the byte and word manipulation instruction extensions and multimedia instruction extensions of the Alpha architecture. Programs compiled with the `pca56` keyword will run correctly on `pca56` processors, but should *not* be run on `ev4`, `ev5`, or `ev56` processors.

`/[no]asmattributes`

Syntax:

`/asmattributes:keyword`, `/noasmattributes`, `/FA`, `/FAs`, `/FAc`, or `/FAcs`

The `/asmattributes` option indicates what information, in addition to the assembly code, should be generated in the assembly listing file.

In Developer Studio, specify Assembly Options in the Listing File Compiler Option Category. The `/asmattributes` options are:

- `/asmattributes:source` or `/FAs`
Intersperses the source code as comments in the assembly listing file.
- `/asmattributes:machine` or `/FAc`
Lists the hex machine instructions at the beginning of each line of assembly code.
- `/asmattributes:all` or `/FAcs`
Intersperses both the source code as comments and lists the hex machine instructions at the beginning of each line of assembly code. This is equivalent to `/asmattributes`.
- `/asmattributes:none` or `/FA`
Provides neither interspersed source code comments nor a listing of hex machine instructions. This is equivalent to `/noasmattributes`.

If you omit the `/asmattributes` option, `/asmattributes:none` is used (default).

The `/asmattributes` option is ignored if the `/[no]asmfile[:file]` option is *not* specified. The `/FA`, `/FAs`, `/FAc`, or `/FAcs` options can be used without the `/[no]asmfile[:file]` option.

`/[no]asmfile`

Syntax:

`/asmfile[:file]`, `/noasmfile`, `/Fa[file]`, `/Fc[file]`, `/FI[file]`, or `/Fs[file]`

The `/asmfile` option or equivalent `/Fx` option indicates that an assembly listing file should be generated. If the `file` is not specified, the default filename used will be the name of the source file with an extension of `.asm`.

In Developer Studio, specify Assembly Listing in the Listing File Compiler Option Category.

When the `/asmfile` option or equivalent `/Fx[file]` option is specified and there are multiple source files being compiled, each source file will be compiled separately. Compiling source files separately turns off interprocedural optimization from being performed.

When you specify `/noasmfile` or the `/asmfile` option is not specified, the compiler does not generate

any assembly files.

To specify the content of the assembly listing file, also specify `/[no]asmattributes:keyword` or specify the `/Fx[file]` options:

- `/FA[file]` provides neither interspersed source code comments nor a listing of hex machine instructions.
- `/FAs[file]` provides interspersed source code as comments in the assembly listing file.
- `/FAc[file]` provides a list of hex machine instructions at the beginning of each line of assembly code.
- `/FAcs[file]` provides interspersed source code as comments and lists hex machine instructions at the beginning of each line of assembly code.

/assume

Syntax:

`/assume:keyword`

The `/assume` option specifies assumptions made by the Fortran syntax analyzer, optimizer, and code generator. These option keywords are:

- `/assume:[no]accuracy_sensitive`
Specifying `/assume:noaccuracy_sensitive` allows the compiler to reorder code based on algebraic identities (inverses, associativity, and distribution) to improve performance. In Developer Studio, specify Allow Reordering of Floating-Point Operations in the Optimizations [Compiler Option Category](#).

The numeric results can be slightly different from the default (`/assume:accuracy_sensitive`) because of the way intermediate results are rounded.

Numeric results with `/assume:noaccuracy_sensitive` are not categorically less accurate. They can produce more accurate results for certain floating-point calculations, such as dot product summations. For example, the following expressions are mathematically equivalent but may not compute the same value using finite precision arithmetic.

$$\begin{aligned} X &= (A + B) - C \\ X &= A + (B - C) \end{aligned}$$

If you omit `/assume:noaccuracy_sensitive` and omit `/fast`, the compiler uses a limited number of rules for calculations, which might prevent some optimizations.

If you specify `/assume:noaccuracy_sensitive`, or if you specify `/fast` and omit `/assume:accuracy_sensitive`, the compiler can reorder code based on algebraic identities to improve performance.

For more information on `/assume:noaccuracy_sensitive`, see [Arithmetic Reordering Optimizations](#).

- `/assume:[no]byterecl`
The `/assume:byterecl` option applies only to unformatted files. In Developer Studio, specify the Use Bytes as Unit for Unformatted Files in the Fortran Data [Compiler Option Category](#).

Specifying the `/assume:byterecl` option:

- Indicates that the units for an explicit OPEN statement RECL specifier value are in bytes.
- Forces the record length value returned by an INQUIRE by output list to be in byte units.

Specifying `/assume:nobyterecl` indicates that the units for RECL values with unformatted files are in four-byte (longword) units. This is the default for the DF command.

- `/assume:[no]dummy_aliases`

Specifying the `/assume:dummy_aliases` option *requires* that the compiler assume that dummy (formal) arguments to procedures share memory locations with other dummy arguments or with variables shared through use association, host association, or common block use. The default is `/assume:nodummy_aliases`.

In Developer Studio, specify Enable Dummy Argument Aliasing in the Fortran Data Compiler Option Category.

These program semantics do not strictly obey the Fortran 90 Standard and they slow performance. If you omit `/assume:dummy_aliases`, the compiler does not need to make these assumptions, which results in better run-time performance. However, omitting `/assume:dummy_aliases` can cause some programs that depend on such aliases to fail or produce wrong answers.

You only need to compile the called subprogram with `/assume:dummy_aliases`.

If you compile a program that uses dummy aliasing with `/assume:nodummy_aliases` in effect, the run-time behavior of the program will be unpredictable. In such programs, the results will depend on the exact optimizations that are performed. In some cases, normal results will occur; however, in other cases, results will differ because the values used in computations involving the offending aliases will differ.

For more information, see Dummy Alias Assumption.

- `/assume:[no]source_include`

This option controls the directory searched for module files specified by a USE statement or source files specified by an INCLUDE statement:

- Specifying `/assume:source_include` requests a search for module or include files in the directory where the source file being compiled resides. This is the default.
- Specifying `/assume:nosource_include` requests a search for module or include files in the current (default) directory.

In Developer Studio, specify the Default INCLUDE and USE Paths in the Preprocessor Compiler Option Category.

- `/assume:[no]underscore`

Specifying `/assume:underscore` option controls the appending of an underscore character to external user-defined names: the main program name, named COMMON, BLOCK DATA, and names implicitly or explicitly declared EXTERNAL. The name of blank COMMON remains `_BLNK_`, and Fortran intrinsic names are not affected.

In Developer Studio, specify Append Underscore to External Names in the External Procedures Compiler Option Category.

Specifying /assume:nounderscore option does not append an underscore character to external user-defined names. This is the default.

For example, the following command requests the noaccuracy_sensitive and nosource_include keywords and accepts the defaults for the other /assume keywords:

```
df /assume:(noaccuracy_sensitive,nosource_include) testfile.f90
```

/[no]automatic

Syntax:

/automatic, **/noautomatic**, **/4Ya**, or **/4Na**

The /automatic or /4Ya option requests that local variables be put on the run-time stack. In Developer Studio, specify Variables Default to Automatic in the Fortran Data Compiler Option Category.

The /noautomatic or /4Na option is the same as the /static option. The default is /noautomatic or /4Na, which causes all local variables to be statically allocated.

If you specify /recursive, the /automatic (/4Ya) option is set.

/bintext

Syntax:

/bintext:string or **/Vstring**

Specifying /bintext (or /V) places the text *string* specified into the object file (.OBJ) being generated by the compiler. This *string* also gets propagated into the executable file. For example, the string might contain version number or copyright information.

In Developer Studio, specify Object Text in the Miscellaneous Compiler Option Category.

If the string contains a space or tab, the string must be enclosed by double quotation marks ("). A backslash (\) must precede any double quotation marks contained within the string.

If the command line contains multiple /bintext or /V options, the last (right-most) one is used. You can specify /nobintext to override previous /bintext or /V options on the same command line.

/[no]browser

Syntax:

/browser[:filename], **/nobrowser**, or **/FR**

The /browser or /FR option controls the generation of source browser information. When the

`/browser` option is not specified, the compiler will not generate browser files (same as `/nobrowser`).

In Developer Studio, specify Generate Source Browse Information in the General Compiler Option Category. Also, in the BrowseInfo tab, set Build Browse info check box instead of using BCSMAKE.

Browser information includes:

- Information about all the symbols in the source file.
- The source code line in which a symbol is defined.
- Each source code line where there is a reference to a symbol.
- The relationships between calling functions and called functions.

The default extension for source browser files is `.SBR`.

The browser output is intended to be used as input to the Browse Information File Maintenance Utility (BSCMAKE), which generates a browse information file (`.BSC`) that can be examined in browse windows in Microsoft Developer Studio.

Instead of using BCSMAKE, you can use the the Project Settings dialog box in Developer Studio:

- Click the BrowseInfo tab
- Set the Build Browse info check box.

When the `/browser` or `/FR` option is specified and there are multiple source files being compiled, each source file will be compiled separately. Compiling source files separately turns off interprocedural optimizations.

`/[no]check`

Syntax:

`/check:keyword, /nocheck, /4Yb, /4Nb`

The `/check`, `/4Yb`, or `/4Nb` option controls whether extra code is generated for certain run-time checking. Run-time checks can result in issuing run-time messages for certain conditions.

In Developer Studio, specify the Extended Error Checking items in the Run time Compiler Option Category. The `/check` keywords and `/4Yb`, and `/4Nb` options are as follows:

- `/check:bounds`
Requests a run-time error message if a reference to an array subscript or character substring is outside of the declared bounds. The default is `/check:nobounds`, which does not issue a run-time message for this condition.
- `/check:flawed_pentium`
On x86 systems, requests a run-time error message if a flawed Pentium® processor is detected. The default is `/check:flawed_pentium`, which *does* issue a run-time error message for this condition and stops program execution. To allow program execution to continue when this condition occurs, set the environment variable `FOR_RUN_FLAWED_PENTIUM` to true and rerun the program (see Run-Time Environment Variables). For more information on the Pentium flaw, see Intel Pentium Floating-Point Flaw. You can also use the

FOR_CHECK_FLAWED_PENTIUM routine.

- /check:format
Requests a run-time error message when the data type for an item being formatted for output does not match the FORMAT descriptor. Specifying /check:noformat suppresses the run-time error message for this condition.
- /check:output_conversion
Requests a run-time continuable error message when a data item is too large to fit in a designated FORMAT descriptor field. Specifying /check:nooutput_conversion results in a noncontinuable error message when a data item is too large to fit in a designated FORMAT descriptor field.
- /check:overflow
Requests a continuable run-time message when integer overflow occurs. Specifying /check:nooverflow suppresses the run-time message.
- /check:nopower
Suppresses the run-time error message for $0.0 ** 0.0$ and *negative-value ** integer-value-of-type-real*, so $0.0 ** 0.0$ is 1.0 and $(-3.0) ** 3.0$ is -27.0.
On Alpha systems, either use the default of /check:power to allow a run-time error message to be issued for this type of expression, or specify /check:nopower to suppress the run-time error message. On x86 systems, /check:nopower is always used.
- /check:underflow
Requests an informational run-time message when floating-point underflow occurs. Specifying /check:nounderflow suppresses a run-time message when floating-point underflow occurs.
- /4Yb
Sets /check:(overflow,bounds,underflow).
- /4Nb or /check:none or /nocheck
Equivalent to:
/check:(nobounds,noformat,nopower,nooutput_conversion,nooverflow,nounderflow).
- /check or /check:all
Equivalent to:
/check:(bounds,flawed_pentium,format,power,output_conversion,overflow,underflow).

On x86 systems, if you omit these options, the default is:

/check:(nobounds,flawed_pentium,noformat,nopower,nooutput_conversion,nooverflow,nounderflow).

On Alpha systems, if you omit these options, the default is:

/check:(nobounds,noformat,power,nooutput_conversion,nooverflow,nounderflow).

/[no]comments

Syntax:

/comments or **/nocomments**

The **/comments** option writes C-style comments to the output file. The **/nocomments** option does not write C-style comments to the output file. For more information, type FPP /? to view FPP options.

/[no]compile_only

Syntax:

/compile_only, **/nocompile_only**, or **/c**

The **/compile_only** or **/c** option suppresses linking. The default is **/nocompile_only** (perform linking).

If you specify the **/compile_only** option at higher levels of optimization and also specify **/object:filename**, the **/object:filename** option causes multiple Fortran input files (if specified) to be compiled into a single object *file*. This allows interprocedural optimizations to occur.

However, if you use multiple source files and the **/compile_only** option without the **/object:file** option, multiple object files are created and interprocedural optimizations do not occur.

/convert

Syntax:

/convert:keyword

The **/convert** option specifies the format of unformatted files containing numeric data. On x86 and Alpha systems, the format used in memory is always IEEE little endian format. If you want to read and write unformatted data in IEEE little endian format, you do not need to convert your unformatted data and can omit this option (or specify **/convert:native**).

In Developer Studio, specify the Unformatted File Conversion in the Compatibility Compiler Option Category. The **/convert** options are:

- **/convert:big_endian**
Specifies that unformatted files containing numeric data are in IEEE big endian (nonnative) format. The resulting program will read and write unformatted files containing numeric data assuming the following:
 - Big endian integer format (INTEGER declarations of the appropriate size).
 - Big endian IEEE floating-point formats (REAL and COMPLEX declarations of the appropriate size).
- **/convert:cray**
Specifies that unformatted files containing numeric data are in CRAY (nonnative) big endian format. The resulting program will read and write unformatted files containing numeric data assuming the following:
 - Big endian integer format (INTEGER declarations of the appropriate size).
 - Big endian CRAY proprietary floating-point formats (REAL and COMPLEX declarations of the appropriate size).
- **/convert:ibm**

Specifies that unformatted files containing numeric data are in IBM (nonnative) big endian format. The resulting program will read and write unformatted files containing numeric data assuming the following:

- Big endian integer format (INTEGER declarations of the appropriate size).
 - Big endian IBM proprietary floating-point formats (REAL and COMPLEX declarations of the appropriate size).
- **/convert:little_endian**
Specifies that numeric data in unformatted files is in native little endian integer format and IEEE little endian floating-point format (same as used in memory), as follows:
 - Integer data is in native little endian format.
 - REAL(KIND=4) and COMPLEX(KIND=4) (SINGLE PRECISION) data is in IEEE little endian S_floating format.
 - REAL(KIND=8) and COMPLEX (KIND=8) (DOUBLE PRECISION) data is in IEEE little endian T_floating format.
 - **/convert:native**
Specifies that numeric data in unformatted files is not converted. This is the default.
 - **/convert:vaxd**
Specifies that numeric data in unformatted files is in VAXD little endian format, as follows:
 - Integer data is in native little endian format.
 - REAL(KIND=4) and COMPLEX(KIND=4) (SINGLE PRECISION) data is in VAX F_floating format.
 - REAL(KIND=8) and COMPLEX (KIND=8) (DOUBLE PRECISION) data is in VAX D_floating format.
 - **/convert:vaxg**
Specifies that numeric data in unformatted files is in VAXG little endian format, as follows:
 - Integer data is in native little endian format.
 - REAL(KIND=4) and COMPLEX(KIND=4) (SINGLE PRECISION) data is in VAX F_floating format.
 - REAL(KIND=8) and COMPLEX (KIND=8) (DOUBLE PRECISION) data is in VAX G_floating format.

/[no]d_lines

Syntax:

/d_lines, /nod_lines, /4ccD, or /4ccd

The **/d_lines**, **/4ccD**, or **/4ccd** option indicates that lines in fixed-format files that contain a D in column 1 should be treated as source code. Specifying **/nod_lines** (the default) indicates that these lines are to be treated as comment lines.

In Developer Studio, specify **Compile DEBUG (D) Lines** in the Debug Compiler Option Category.

Visual Fortran does not support the use of characters other than a D or d with the */4ccstring*.

/[no]dbglibs

Syntax:

`/dbglibs` or `/nodbglibs`

The `/dbglibs` option controls whether the debug version or the non-debug version of the C run-time library is linked against. The default is `/nodbglibs`, which will link against the non-debug version of the C library, even when `/debug:full` is specified.

If you specify `/debug:full` for an application that calls C library routines and you need to debug calls into the C library, you should also specify `/dbglibs` to request that the debug version of the library be linked against.

In Developer Studio, specify the Use Debug C Libraries in the Libraries Compiler Option Category.

When you specify `/dbglibs`, the C debug library linked against depends on the specified `/libs:keyword` and `/[no]threads` options, and is one of: `libcd.lib`, `libcmdtd.lib`, or `msvcrttd.lib` (see Visual Fortran/Visual C++ Mixed-Language Programs).

`/[no]debug`

Syntax:

`/debug:keyword`, `/nodebug`, `/Z7`, `/Zd`, or `/Zi`

The `/debug`, `/Z7`, `/Zd`, or `/Zi` option controls the level of debugging information associated with the program being compiled.

In Developer Studio, specify the Debugging Level in the General or Debug Compiler Option Category. The options are:

- `/debug:none` or `/nodebug`
If you specify `/debug:none` or `/nodebug`, the compiler produces no traceback or symbol table information needed for debugging or profiling. Only symbol information needed for linking (global symbols) is produced. The size of the resulting object module is the minimum size. If this option is specified, `/debug:none` is passed to the linker.
- `/debug:minimal` or `/Zd`
If you specify `/debug:minimal` or `/Zd`, the compiler produces minimal traceback information, which allows global symbol table information needed for linking, but not local symbol table information needed for debugging. If `/debug:minimal` is specified, `/debug:minimal` and `/debugtype:cv` is passed to the linker.

If you omit the `/[no]debug:keyword`, `/Z7`, `/Zd`, and `/Zi` options, this is the default.

The `/Zd` option implies `/nopdbfile` and passes `/debug:minimal /pdb:none /debugtype:cv` to the linker.

The object module size is somewhat larger than if `/debug:none` was specified, but is smaller than if `/debug:full` was specified.

- `/debug:partial`
If you specify `/debug:partial`, the compiler produces traceback information, which allows program counter to source file line correlation, global symbol table information needed for

linking, but not local symbol table information needed for debugging. If `/debug:partial` is specified, `/debug:partial /debugtype:cv /pdb:none` is passed to the linker.

The object module size is somewhat larger than if `/debug:none` was specified, but is smaller than if `/debug:full` was specified.

- `/debug:full`, `/debug`, `/Zi`, or `/Z7`
 If you specify `/debug:full`, `/debug`, `/Zi`, or `/Z7`, the compiler produces traceback information, symbol table information needed for full symbolic debugging of unoptimized code, and global symbol information needed for linking.

If you specify `/debug:full` for an application that make calls to C library routines and you need to debug calls into the C library, you should also specify `/dbglibs` to request that the appropriate C debug library is linked against.

The `/Z7` option implies `/nopdbfile` and passes `/debug:full /debugtype:cv /pdb:none` to the linker.

The `/debug:full`, `/debug`, and `/Zi` options imply `/pdbfile` and pass `/debug:full` and `/debugtype:cv` to the linker.

If you specify `/debug` (with no keyword), the default optimization level changes to `/optimize:0` (instead of `/optimize:4`) for the `DF` command.

/define

Syntax:

`/define:symbol[=integer]`

The `/define` option defines the *symbol* specified for use with conditional compilation directives. If a value is specified, it must be an *integer* value. If a value is not specified, 1 is assigned to *symbol*.

In Developer Studio, specify the Predefined Preprocessor Variables in the General or Preprocessor Compiler Option Category.

You can use the directives to detect symbol definitions, such as the IF Directive Construct. Like certain other compiler options, an equivalent directive exists (DEFINE directive).

The following preprocessor symbols are defined by the compiler:

<code>_DF_VERSION_= 500</code> (500 for Version 5.0)	<code>_WIN32=1</code> (always defined)
<code>_X86_=1</code> (on x86 systems only)	<code>_ALPHA_=1</code> (on Alpha systems only)
<code>_WIN95=1</code> (on Windows 95 systems only)	<code>_MT=1</code> (only if <code>/threads</code> or <code>/MT</code> is specified)
<code>_DLL=1</code> (only if <code>/dll</code> or <code>/LD</code> is specified)	<code>_MSFORTRAN_=401</code> (only if <code>/fpscomp:symbols</code> is specified or you use the <code>FL32</code> command)

/dll

Syntax:

/dll[:file]*, */nodll*, or */LD

The */dll* or */LD* option indicates that the program should be linked as a DLL file. The */dll* or */LD* option overrides any specification of the run-time routines to be used and activates the */libs:dll* option. A warning is generated when the */libs=qwin* or */libs=qwins* option and */dll* option are used together.

In Developer Studio, specify the project type as Dynamic Link Library (DLL).

If you omit *file*, the */dll* or */LD* option interacts with the */exe* and the */Fe* options, as follows:

- If neither */exe* nor */Fe* is specified, the first file name used on the command line is used with an extension of .DLL.
- If either */exe:file* or */Fe:file* is specified with a *file* name, that name is used for the DLL file. If the specified file name does not end with a "." or have an extension, an extension of .DLL is added to it.

To request linking with multithreaded libraries, specify the */threads* option.

For information about building DLL files from Developer Studio, see Dynamic-Link Library Projects and Building Dynamic-Link Library Projects.

For a list of Fortran Powerstation style options (such as */LD* and */MDs*) and their DF command equivalents, see Equivalent Visual Fortran Compiler Options.

/[no]error_limit

Syntax:

/error_limit[:count]* or */noerror_limit

The */error_limit* option specifies the maximum number of error-level or fatal-level compiler errors allowed for a given file before compilation aborts. If you specify */noerror_limit*, there is no limit on the number of errors that are allowed.

In Developer Studio, specify the Compilation Error Limit in the Miscellaneous Compiler Option Category.

The default is */error_limit:30* or a maximum of 30 error-level and fatal-level messages. If the maximum number of errors is reached, a warning message is issued and the next file (if any) on the command line is compiled.

/[no]exe

Syntax:

/exe[:file]*, */noexe*, or */Fe:file

The */exe* or */Fe* option specifies the name of the executable program (EXE) or dynamic-link library (DLL) *file* being created. To request that a DLL be created instead of an executable program, specify

the `/dll` option.

`/[no]extend_source`

Syntax:

`/extend_source[:size]`, **`/noextend_source`**, or **`/4Lsize`**

The `/extend_source` or `/4Lsize` option controls the column used to end the statement field in fixed-format source files. When a size is specified, that will be the last column parsed as part of the statement field. Any columns after that will be treated as comments.

In Developer Studio, specify the Fixed-Form Line Length in the Fortran Language Compiler Option Category. The following options are equivalent:

- `/noextend_source`, `/extend_source:72`, or `/4L72` specify the last column as 72.
- `/extend_source:80` or `/4L80` specify the last column as 80.
- `/extend_source`, `/extend_source:132`, or `/4L132` specify the last column as 132.

`/extfor`

Syntax:

`/extfor:ext`

The `/extfor:` option specifies file extensions to be processed (`/extfor`) by the DIGITAL Fortran compiler. One or more file extensions can be specified. A leading period before each extension is optional (for and `.for` are equivalent).

`/extfpp`

Syntax:

`/extfpp:ext`

The `/extfpp` option specifies file extensions to be processed (`/extfpp`) by the FPP preprocessor. One or more file extensions can be specified. A leading period before each extension is optional (fpp and `.fpp` are equivalent).

`/extlnk`

Syntax:

`/extlnk:ext`

The `/extlnk` option specifies file extensions to be processed (`/extlnk`) by the linker. One or more file extensions can be specified. A leading period before each extension is optional (obj and `.obj` are equivalent).

`/[no]f66`

Syntax:

/f66 or /nof66

The /f66 option requests that the compiler select FORTRAN-66 interpretations in cases of incompatibility. One of these differences is that DO loops are executed at least once.

In Developer Studio, specify Enable FORTRAN-66 Semantics in the Fortran Language Compiler Option Category.

/[no]f77rtl

Syntax:

/f77rtl or /nof77rtl

The /f77rtl option controls the run-time support that is used when a program is executed. Specifying /f77rtl uses the DIGITAL Fortran 77 run-time behavior. In Developer Studio, specify Enable F77 Run-Time Compatibility in the Compatibility Compiler Option Category.

Specifying /nof77rtl uses the Visual Fortran (DIGITAL Fortran 90) run-time behavior. Unless you specify /f77rtl, /nof77rtl is used.

/fast

Syntax:

/fast

The /fast option sets several options that generate optimized code for fast run-time performance. Specifying this option is equivalent to specifying:

- /assume:noaccuracy_sensitive
- /math_library:fast
- /alignment:(dcommons, records)

If you omit /fast, these performance-related options will not be set.

/[no]fixed

Syntax:

/fixed, /nofixed, /4Nf, or /4Yf

The /fixed or /4Nf option specifies that the source file should be interpreted as being in fixed-source format. Equivalent options are as follows:

- The /fixed, /nofree, and /4Nf options request fixed-source form.
- The /nofixed, /free, and /4Yf options request free-source form.

In Developer Studio, specify the Source Form in the Fortran Language Compiler Option Category.

If you omit /[no]fixed, /4Nf, and /4Yf:

- Files with an extension of .f90 or .F90 are assumed to be free-format source files.

- Files with an extension of .f, .for, .FOR, or .i are assumed to be fixed-format files.

/[no]fltconsistency (x86 only)

Syntax:

/fltconsistency, /nofltconsistency, or /Op

The /fltconsistency or /Op option enables improved floating-point consistency. Floating-point operations are not reordered and the result of each floating-point operation is stored into the target variable rather than being kept in the floating-point processor for use in a subsequent calculation. This option is ignored on Alpha systems.

In Developer Studio, specify Enable Floating-Point Exception Consistency in the Floating Point Compiler Option Category.

The default is /nofltconsistency, which provides better run-time performance at the expense of less consistent floating-point results.

/[no]fpconstant

Syntax:

/fpconstant or /nofpconstant

The /fpconstant option requests that a single-precision constant assigned to a double-precision variable be evaluated in double precision. If you omit /fpconstant (or specify the default /nofpconstant), a single-precision constant assigned to a double-precision variable is evaluated in single precision. The Fortran 90 standard requires that the constant be evaluated in single precision.

In Developer Studio, specify Extended Precision of Single-Precision Constants in the Floating Point Compiler Option Category.

Certain programs created for FORTRAN-77 compilers (including DIGITAL Fortran 77) may show different floating-point results, because they rely on single-precision constants assigned to a double-precision variable to be evaluated in double precision.

In the following example, if you specify /fpconstant, identical values are assigned to D1 and D2. If you omit the /fpconstant option, the compiler will obey the standard and assign a less precise value to D1:

```
REAL (KIND=8) D1, D2
DATA D1 /2.71828182846182/    ! REAL (KIND=4) value expanded to double
DATA D2 /2.71828182846182D0/ ! Double value assigned to double
```

/fpe

Syntax:

/fpe:level

The `/fpe:level` option controls floating-point exception handling at run time for the main program. This includes whether exceptional floating-point values are allowed and how precisely run-time exceptions are reported. The `/fpe:level` option specifies how the compiler should handle the following floating-point exceptions:

- When floating-point calculations result in a divide by zero, overflow, or invalid data.
- When floating-point calculations result in an underflow operation.
- When a denormalized number or other exceptional number (positive infinity, negative infinity, or a NaN) is present in an arithmetic expression

For performance reasons:

- On x86 systems, the default is `/fpe:3`. Using `/fpe:0` will slow run-time performance on x86 systems.
- On Alpha systems, the default is `/fpe:0` (many programs do not need to handle denormalized numbers or other exceptional values). Using `/fpe:3` will slow run-time performance on Alpha systems.

On Alpha systems, to associate an exception with the instruction that causes the exception, specify `/fpe:3` or specify `/synchronous_exceptions`.

In Developer Studio, specify the Floating-Point Exception Handling in the Floating Point Compiler Option Category. The `/fpe:level` (*level* is 0, 1, or 3) options are as follows:

Option	Handling of Underflow	Handling of Divide by Zero, Overflow, and Invalid Data Operation
<code>/fpe:0</code> (default on Alpha systems)	Sets any calculated denormalized value (result) to zero and lets the program continue. A message is displayed only if <code>/check:underflow</code> is also specified. Any use of a denormalized number (non-finite data) in an arithmetic expression results in an invalid operation error and the program terminates.	Exceptional values are <i>not</i> allowed. The program terminates after displaying a message. The exception location is one or more instructions <i>after</i> the instruction that caused the exception, unless (on Alpha systems) <u><code>/synchronous_exceptions</code></u> was specified.
<code>/fpe:1</code> (Alpha systems only)	Sets any calculated denormalized value (result) to zero and lets the program continue. A message is displayed only if <code>/check:underflow</code> is also specified. Use of a denormalized (or exceptional) number in an arithmetic expression results in program continuation, but with slower performance.	The program continues. No message is displayed. A NaN or Infinity (+ or -) will be generated.
<code>/fpe:3</code> (default on x86 systems)	Leaves any calculated denormalized value as is. The program continues, allowing gradual underflow. Use of a denormalized (or exceptional) number in an arithmetic expression results in program continuation, but with slower performance. A message is displayed only if	The program continues. No message is displayed. A NaN or Infinity (+ or -) will be generated.

/check:underflow is also specified.	
-------------------------------------	--

The exception message reporting specified by the `/fpe:level` option applies only to the main program and cannot be changed during program execution.

When compiling different routines in a program separately, you should use the same `/fpe:level` value.

On x86 systems, for programs that flush denormalized values to zero (such as those that allow gradual underflow with `/fpe:0`), the impact on run-time performance can be significant. On Alpha systems, for programs that use a number of denormalized values (such as those that allow gradual underflow with `/fpe:3`), the impact on run-time performance can be significant.

On Alpha systems, if you use the `/math_library:fast` along with an `/fpe:level` option, the `/fpe:level` option is ignored when arithmetic values are evaluated by math library routines.

To help you debug a routine, you can associate an exception with the instruction that caused it by specifying `/fpe:3`, or, on Alpha systems, by specifying `/fpe:0` with `/synchronous_exceptions`.

On x86 systems, the `/fpe` option, `/check:underflow` option, and `MATHERRQQ` routine interact as follows:

Specified <code>/fpe:n</code> Option	Was <code>/check:underflow</code> Specified?	Is a User-Written <code>MATHERRQQ</code> Routine Present?	Underflow Handling by the Visual Fortran Run-Time System on x86 Systems
<code>/fpe:0</code>	No	No	The underflowed result is set to zero (0). The program continues.
<code>/fpe:0</code>	No	Yes	The underflowed result is set to zero (0). The program continues.
<code>/fpe:0</code>	Yes	No	The underflowed result is set to zero (0). The program continues. The number of underflowed results are counted and messages are displayed for the first two occurrences.
<code>/fpe:0</code>	Yes	Yes	The underflowed result is set to zero (0). The program continues. The number of underflowed results are counted and messages are displayed for the first two occurrences.
<code>/fpe:3</code>	No	No	Denormalized results are allowed and the program continues. Traps are masked and no handlers are invoked.
<code>/fpe:3</code>	No	Yes	Denormalized results are allowed and the program continues. Traps are masked and no handlers are invoked.
<code>/fpe:3</code>	Yes	No	For Version 5.0, a fatal error results and the program terminates.
<code>/fpe:3</code>	Yes	Yes	Depends on the source causing the underflow: <ul style="list-style-type: none"> • If the underflow occurs in an

			<p>intrinsic procedure, the undefined result is left as is. The program continues with the assumption that the user-specified MATHERRQQ handler will perform any result fix up needed.</p> <ul style="list-style-type: none"> • If the underflow does not occur in an intrinsic procedure, for Version 5.0, a fatal error results and the program terminates.
--	--	--	--

For more information about the floating-point environment and the MATHERRQQ routine (x86 systems), see: [The Floating-Point Environment](#).

For information about routines that can obtain or set the floating-point exception settings used by Visual Fortran at run-time, see [FOR_SET_FPE](#) and [FOR_GET_FPE](#).

For more information on IEEE floating-point exception handling, see *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Standard 754-1985).

/fpp

Syntax:

/fpp:options

The /fpp option passes *options* to FPP as is. You can type FPP /HELP for information on FPP options.

/[no]fpscomp

Syntax:

/fpscomp[:keyword...] or **/nofpscomp**

The /fpscomp option controls whether certain aspects of the run-time system and semantic language attributes within the compiler are compatible with Visual Fortran or Microsoft Fortran Powerstation.

If you experience problems when porting applications from Fortran Powerstation, specify /fpscomp:keyword (or /fpscomp:all). When porting applications from DIGITAL Fortran, use /fpscomp:none (the default).

In Developer Studio, specify the Powerstation 4.0 Compatibility Options in the [Compatibility Compiler Option Category](#). The /fpscomp options are:

- /fpscomp:[no]filesfromcmd
Specifying /fpscomp:filesfromcmd for a file where the **OPEN** statement **FILE** specifier is blank (FILE=' '), requests that the following actions be taken at run-time:
 - The program reads a filename from the list of arguments (if any) in the command line that invoked the program. If any of the command-line arguments contain a null string

that invoked the program. If any of the command-line arguments contain a null string (''), the program asks the user for the corresponding filename. Each additional **OPEN** statement with a nameless **FILE** specifier reads the next command-line argument.

- If there are more nameless **OPEN** statements than command-line arguments, the program prompts for additional file names.
- In a QuickWin application, a File Select dialog box appears to request file names.

Specifying `/fpscomp:nofilesfromcmd` disables the run-time system from using the filename specified on the command line when the **OPEN** statement **FILE** specifier is omitted, allowing the application of default directory, file name, and extensions like DIGITAL Fortran, such as the `FORTn` environment variable and the `FORT.n` file name (where *n* is the unit number).

Specifying `/fpscomp:filesfromcmd` affects the following Fortran constructs:

- The **OPEN** statement **FILE** specifier.

For example, assume a program `OPENTEST` contains the following statements:

```
OPEN(UNIT = 2, FILE = ' ')
OPEN(UNIT = 3, FILE = ' ')
OPEN(UNIT = 4, FILE = ' ')
```

The command line, `opentest test.dat " "` assigns the file `TEST.DAT` to Unit 2, prompts the user for a filename to associate with Unit 3, then prompts again for a filename to associate with Unit 4.

- Implicit file open statements such as the **WRITE**, **READ**, and **ENDFILE** statements.

Unopened files referred to in **READ** or **WRITE** statements are opened implicitly as if there had been an **OPEN** statement with a name specified as all blanks. The name is read from the command line.

```
WRITE(UNIT = 8, FMT='(2I5)') int1, int2
! Where "8" has not been explicitly associated with a file.
```

For more information about running Visual Fortran programs with the `/fpscomp:filesfromcmd` option set, see [Running Fortran Applications](#).

- `/fpscomp:[no]general`

Controls which run-time behavior is used when a difference exists between Visual Fortran and Microsoft Fortran Powerstation and either semantic must remain available for compatibility reasons. Specify `/fpscomp:general` to request Fortran Powerstation semantics. Specify `/fpscomp:nogeneral` to request Visual Fortran semantics. This affects the following Fortran constructs:

- The **BACKSPACE** statement:
 - Allows files opened with `ACCESS="APPEND"` to be used with the **BACKSPACE** statement.
 - Allows files opened with `ACCESS="DIRECT"` to be used with the **BACKSPACE** statement.

Note: Allowing files that are not opened with sequential access (such as `ACCESS="DIRECT"`) to be used with the **BACKSPACE** statement violates the Fortran 90 standard and may be removed in the future. Section 9.5 states the following:

- BACKSPACE**, an **ENDFILE**, or a **REWIND** statement..."
- The **REWIND** statement.
Allows files opened with **ACCESS="DIRECT"** to be used with the **REWIND** statement.
Note: Allowing files that are not opened with sequential access (such as **ACCESS="DIRECT"**) to be used with the **REWIND** statement violates the Fortran 90 standard and may be removed in the future. Section 9.5 states the following: "A file that is not connected for sequential access must not be referred to by a **BACKSPACE**, an **ENDFILE**, or a **REWIND** statement..."
 - The **READ** statement.
 - Formatted: `READ(eunit, format [,advance][,iostat]...)`
Reading from a formatted file opened for direct access will read records that have the same record type format as Fortran Powerstation when `/fpscomp:general` is set. This consists of accounting for the trailing Carriage Return/Line Feed pair (`<CR><LF>`) which is part of the record.
Allows sequential reads from a formatted file opened for direct access.
Note: Allowing files that are not opened with sequential access (such as **ACCESS="DIRECT"**) to be used with the sequential **READ** statement violates the Fortran 90 standard and may be removed in the future. Section 9.2.1.2.2 states the following: "Reading and writing records is accomplished only by direct access input/output statements."
 - Allows the last record in a file opened with **FORM="FORMATTED"** and a record type of **STREAM_LF** or **STREAM_CR** that does not end with a proper record terminator (`<line feed>` or `<carriage return>`) to be read without producing an error.
 - Unformatted: `READ(eunit, [,iostat]...)`
Allows sequential reads from an unformatted file opened for direct access.
Note: Allowing files that are not opened with sequential access (such as **ACCESS="DIRECT"**) to be read with the sequential **READ** statement violates the Fortran 90 standard and may be removed in the future. Section 9.2.1.2.2 states the following: "Reading and writing records is accomplished only by direct access input/output statements."
 - The **INQUIRE** statement
 - The **CARRIAGECONTROL** specifier returns the value "UNDEFINED" instead of "UNKNOWN" when the carriage control is not known and when `/fpscomp:general` is set.
 - The **NAME** specifier returns the file name "UNKNOWN" instead of space filling the file name when the file name is not known and when `/fpscomp:general` is set.
 - The **SEQUENTIAL** specifier returns the value "YES" instead of "NO" for a direct access formatted file when `/fpscomp:general` is set.
 - The **UNFORMATTED** specifier returns the value "NO" instead of "UNKNOWN" when it is not known whether unformatted I/O can be performed to the file and when `/fpscomp:general` is set.
Note: Returning the value "NO" instead of "UNKNOWN" for this specifier violates the Fortran 90 standard and may be removed in the future. See Section 9.6.1.12.
 - The **OPEN** statement
 - If a file is opened with an unspecified "STATUS" keyword value, and is not

named (no **FILE** specifier), the file is opened as a scratch file when /fpscomp:general is set. For example:

```
OPEN (UNIT = 4)
```

In contrast, when /fpscomp:nogeneral is in effect with an unspecified **STATUS** keyword value with no **FILE** specifier, the FORT n environment variable and the FORT. n file name are used (where n is the unit number).

- If the **STATUS** keyword value was not specified and if the name of the file is "USER", the file is marked for deletion when it is closed when /fpscomp:general is set.
- Allows a file to be opened with the **APPEND** and **READONLY** attributes when /fpscomp:general is set.
- If the **CARRIAGECONTROL** specifier is defaulted, gives the "LIST" carriage control attribute to direct access formatted files instead of "NONE" when /fpscomp:general is set.
- Gives an opened file the additional default of write sharing when /fpscomp:general is set.
- Gives the a file a default block size of 1024 as opposed to 8192 when /fpscomp:general is set.
- If the **MODE** and **ACTION** specifier is defaulted and there was an error opening the file, then try opening the file read only, then write only.
- If the **CARRIAGECONTROL** keyword is defaulted and if the device type is a terminal file the file is given the default carriage control value of "FORTRAN" as opposed to "LIST" when /fpscomp:general is set.
- If a file that is being re-opened has a different file type than the current existing file, an error is returned when /fpscomp:general is set.
- Gives direct access formatted files the same record type as Fortran Powerstation when /fpscomp:general is set. This means accounting for the trailing Carriage Return/Line Feed pair (<CR><LF>) which is part of the record.
- The **STOP** statement
 - Writes the Fortran Powerstation output string and/or returns the same exit condition values when /fpscomp:general is set.
- The **WRITE** statement
 - Formatted: WRITE(eunit, format [,advance][,iostat]...)
 - 1. Writing to formatted direct files
 - When writing to a formatted file opened for direct access, records are written in the same record type format as Fortran Powerstation when /fpscomp:general is set. This consists of adding the trailing Carriage Return/Line Feed pair (<CR><LF>) which is part of the record.
 - Ignores the **CARRIAGECONTROL** specifier setting when writing to a formatted direct access file.
 - 2. Interpreting Fortran carriage control characters
 - When interpreting Fortran carriage control characters during formatted I/O, carriage control sequences are written which are the same as Fortran Powerstation when /fpscomp:general is set. This is true for the "Space, 0, 1 and + " characters.
 - 3. Performing non-advancing I/O to the terminal.

When performing non-advancing I/O to the terminal, output is written in the same format as Fortran Powerstation when /fpscomp:general is set.

4. Interpreting the backslash (\) and dollar (\$) edit descriptors

When interpreting backslash and dollar edit descriptors during formatted I/O, sequences are written the same as Fortran Powerstation when /fpscomp:general is set.

- Unformatted: WRITE(eunit, [,iostat]...)
Allows sequential writes from an unformatted file opened for direct access.
Note: Allowing files that are not opened with sequential access (such as ACCESS="DIRECT") to be read with the sequential WRITE statement violates the Fortran 90 standard and may be removed in the future. Section 9.2.1.2.2 states the following: "Reading and writing records is accomplished only by direct access input/output statements."

- /fpscomp:[no]ioformat

Controls which run-time behavior is used for the semantic format for list-directed formatted I/O and unformatted I/O. Specify /fpscomp:ioformat to request Microsoft Fortran Powerstation semantic conventions. Specify /fpscomp:noioformat to request DIGITAL Fortran semantic conventions. This affects the following Fortran constructs:

- The **WRITE** statement

- Formatted List-Directed: WRITE(eunit, * [,iostat]...)
- Formatted Internal List-Directed: WRITE(iunit,* [,iostat]...)
- Formatted Namelist: WRITE(eunit, nml-group [,iostat]...)

If /fpscomp:ioformat is set, the output line, field width values, and the list-directed data type semantics are dictated according to the following sample for real constants:

- For $1 \leq N < 10^{*7}$, use F15.6 for single precision or F24.15 for double.
- For $10^{*7} \leq N < 1$, use E15.6E2 for single precision or E24.15E3 for double.

See the Fortran Powerstation documentation for more detailed information about the other data types affected.

- Unformatted: WRITE(eunit, [,iostat]...)
If /fpscomp:ioformat is set, the unformatted file semantics are dictated according to the Fortran Powerstation documentation. Be aware that the file format differs from that used by DIGITAL Fortran. See the Fortran Powerstation documentation for more detailed information.

The following table summarizes the default output formats for list-directed output with the intrinsic data types:

Default Formats for List-Directed Output		
Data Type	Output Format with /fpscomp:noioformat	Output For /fpscomp:
BYTE	I5	I12
LOGICAL (all)	L2	L2
INTEGER(1)	I5	I12
INTEGER(2)	I7	I12
INTEGER(4)	I12	I12

INTEGER(8) (Alpha only)	I22	I22
REAL(4)	1PG15.7E2	1PG16.6E2
REAL(8) T_floating	1PG24.15E3	1PG25.15E3
COMPLEX(4)	'(',1PG14.7E2, ', ',1PG14.7E2, ') '	'(',1PG16.6E2, ', ',1PG16.6E2, ') '
COMPLEX(8)	'(',1PG23.15E3, ', ',1PG23.15E3, ') '	'(',1PG25.15E3, ', ',1PG25.15E3, ') '
CHARACTER	Aw ³	Aw ³

- The **READ** statement

- Formatted List-Directed: `READ(eunit,* [,iostat]...)`
- Formatted Internal List-Directed: `READ(iunit,* [,iostat]...)`
- Formatted Namelist: `READ(eunit, nml-group [,iostat]...)`

If `/fpscomp:ioformat` is set, the field width values and the list-directed semantics are dictated according to the following sample for real constants:

- For $1 \leq N < 10^{**7}$, use F15.6 for single precision or F24.15 for double.
- For $10^{**7} \leq N < 1$, use E15.6E2 for single precision or E24.15E3 for double.

See the Fortran Powerstation documentation for more detailed information about the other data types affected.

- Unformatted: `READ(eunit, [,iostat]...)`

If `/fpscomp:ioformat` is set, the unformatted file semantics are dictated according to the Fortran Powerstation documentation. Be aware that the file format to read differs from that used by DIGITAL Fortran. See the Fortran Powerstation documentation for more detailed information.

- `/fpscomp:[no]libs`

Controls whether the library `dfport.lib` is passed to the compiler and linker. Specifying `/fpscomp:libs` passes this library. Specifying `/fpscomp:nolib`s does not pass this library.

- `/fpscomp:[no]logicals`

Controls the value used for logical true. Microsoft Fortran Powerstation and DIGITAL Fortran with the `/fpscomp:logical` option set uses any non-zero value (default is 1) for true. DIGITAL Fortran with the `/fpscomp:nological` option set only looks at the low bit of the value, using a -1 for true. Differences can occur when a logical is stored into an integer. Both use 0 (zero) for false.

This affects the results of all logical expressions and affects the return value for following Fortran constructs:

- The **INQUIRE** statement specifiers **OPENED**, **IOFOCUS**, **EXISTS**, and **NAMED**.
- The **EOF** intrinsic function.
- The **BTEST** intrinsic function.
- The lexical intrinsic functions **LLT**, **LLE**, **LGT**, and **LGE**.

- `/fpscomp:[no]symbols`

Adds one or more symbols related to Microsoft Fortran Powerstation to preprocessor and compiler invocations. The symbol currently set by specifying `/fpscomp:symbols` is `_MSFORTRAN_=401`.

- `/fpscomp:all` and `/fpscomp`
Enable full Microsoft Fortran Powerstation compatibility or
`/fpscomp:(filesfromcmd,general,ioformat,libs,logicals,symbols)`.
- `/nofpscomp` or `/fpscomp:none`
Enables full DIGITAL Fortran compatibility or
`/fpscomp:(nofilesfromcmd,nogeneral,noioformat,nolib,nologicals,nosymbols)`.

If you omit `/fpscomp`, the defaults are `/nofpscomp (/fpscomp:none)`.

The `/fpscomp` and `/vms` options are not allowed in the same command.

`/[no]free`

Syntax:

`/free`, `/nofree`, `/4Yf`, or `/4Nf`

The `/free` or `/4Yf` option specifies that the source file should be interpreted as being in free source format. Equivalent options are as follows:

- `/free`, `/nofixed`, or `/4Yf` request free-source form.
- `/nofree`, `/fixed`, or `/4Nf` request fixed-source form.

In Developer Studio, specify the Source Form in the Fortran Language Compiler Option Category.

If you omit `/[no]free`, `/[no]fixed`, `/4Nf`, and `/4Yf`, the compiler assumes:

- Files with an extension of `.f90` or `.F90` are free-format source files.
- Files with an extension of `.f`, `.for`, `.FOR`, or `.i` are fixed-format files.

`/granularity (Alpha only)`

Syntax:

`/granularity:keyword`

On Alpha systems, the `/granularity` option ensures that data of the specified or larger size can be accessed from different threads sharing data in memory. Such data must be aligned on the natural boundary and declared as VOLATILE (so it is not held in registers). This option is ignored on x86 processor systems.

In Developer Studio, specify the Thread Access Granularity in the Fortran Data Compiler Option Category.

You do not need to specify this option for local data access by a single process, unless you have requested multithread library use or asynchronous write access from outside the user process might occur. The `/granularity:keyword` options are as follows:

- `/granularity:byte`
Specifies that all data (one byte or greater) can be accessed from different threads sharing data in memory. This option will slow run-time performance.

- `/granularity:longword`
Specifies that naturally aligned data of four bytes or greater can be accessed safely from different threads sharing access to that data in memory. Accessing data items of three bytes or less and misaligned data may result in data items written from multiple threads being inconsistently updated.
- `/granularity:quadword`
Specifies that naturally aligned data of eight bytes can be accessed safely from different threads sharing data in memory. This is the default. Accessing data items of seven bytes or less and misaligned data may result in data items written from multiple threads being inconsistently updated.

/help

Syntax:

/help or **/?**

The `/help` and `/?` option display information about the DF command. The option can be placed anywhere on the command line.

For a table of DF command options listed alphabetically, see [Options List, Alphabetic Order](#).

/iface

Syntax:

/iface[:keyword...]

The `/iface` option determines the type of argument-passing conventions used by your program for general arguments and for hidden-length character arguments.

In Developer Studio, specify the Default Calling Conventions and the String Length Argument Passing in the External Procedures [Compiler Option Category](#). The `/iface` keywords are as follows:

- The general argument-passing convention keywords are one of: `cref`, `stdref`, and `default` (`stdref` and `default` are equivalent). The functions performed by each are described in the following table:

	/iface:cref	iface:default	/iface:stdref
Arguments are passed	By reference	By reference	By reference
Append @n to names on x86 systems?	No	Yes	Yes
Who cleans up stack	Caller	Callee	Callee
Var args support?	Yes	No	No

- To specify the convention for passing the hidden-length character arguments, specify `/iface:[no]mixed_str_len_arg`:
 - `/iface:mixed_str_len_arg`
Requests that the hidden lengths be placed *immediately after* their corresponding character argument in the argument list, which is the method used by Microsoft Fortran

Powerstation.

- `/iface:nomixed_str_len_arg`
Requests that the hidden lengths be placed in sequential order at the *end* of the argument list, which is the method used by DIGITAL Fortran on Windows NTtm Alpha (and DIGITAL UNIX®) systems by default. When porting mixed-language programs that pass character arguments, either this option must be specified correctly or the order of hidden length arguments changed in the source code.

If you omit the `/iface` option, the following is used:

```
/iface=(default,mixed_str_len_arg)
```

For more information on argument passing, see [Programming with Mixed Languages](#).

`/[no]include`

Syntax:

`/include[:path...]`, **`/noinclude`**, or **`/Ipath`**

The `/include` or `/I` option specifies one or more additional directories (*path*) to be searched for module files ([USE](#) statement) and include files ([INCLUDE](#) statement).

In Developer Studio, specify Custom INCLUDE and USE Path in the Preprocessor [Compiler Option Category](#).

When module or include file names do not begin with a device or directory name, the directories searched are as follows:

1. The directory containing the first source file or the current directory (depends on whether `/assume:source_include` was specified).
2. The current default directory where the compilation is taking place
3. If specified, the directory or directories listed in the `/include:path` or `/Ipath` option. The order of searching multiple directories occurs within the specified list from left to right
4. The directories indicated in the environment variable INCLUDE

To request that the compiler search first in the directory where the source file resides instead of the current directory, specify `/assume:source_include`.

Specifying `/noinclude` (or `/include` or `/I` without a *path*) prevents searching in the standard directory specified by the INCLUDE environment variable.

`/[no]inline`

Syntax:

`/inline[:keyword]`, **`/noinline`**, or **`/Ob2`**

The `/inline` or `/Ob2` option allows users to have some control over inlining. Inlining procedures can greatly improve the run-time performance for certain applications.

When requesting procedure inlining (or interprocedural optimizations), compile all source files together into a single object file whenever possible. With very large applications, compile as many related source files together as possible.

If you compile sources without linking (see the [/compile_only](#) or [/c](#) option), be sure to also specify the [/object\[:filename\]](#) or [/Fofilename](#) option to create a single object file.

In Developer Studio, specify the Inlining type in the Optimizations [Compiler Option Category](#). The [/inline](#) options are:

- [/inline:none](#) or [/noinline](#)
Prevents the inlining of procedures, except for statement functions. This type of inlining occurs when you specify [/optimize:0](#) or [/Od](#).
- [/inline:manual](#)
Prevents the inlining of procedures, except for statement functions. This type of inlining occurs when you specify [/optimize:0](#) or [/Od](#).
- [/inline:size](#)
Inlines procedures that will improve run-time performance without significantly increasing program size. It includes the types of procedures inlined when you specify [/inline:manual](#). This type of inlining is available with [/optimize:1](#) or higher.
- [/inline:speed](#) or [/Ob2](#)
Inlines procedures that will improve run-time performance with a significant increase in program size. This type of inlining is available with [/optimize:1](#) or higher. If you omit [/\[no\]inline](#) or [/Ob2](#), [/inline:speed](#) occurs automatically if you specify [/optimize:4](#), [/optimize:5](#), [/Ox](#), or [/Oxp](#).
- [inline:all](#)
Inlines absolutely every call that it is possible to inline while still getting correct code. However, recursive routines will not cause an infinite loop at compile time. This type of inlining is available with [/optimize:1](#) or higher. It includes the types of procedures inlined when you specify other [/inline](#) options.

Using [/inline:all](#) can significantly increase program size and slow compilation speed.

For more detailed information on this option, see [Controlling the Inlining of Procedures](#).

[/\[no\]intconstant](#)

Syntax:

[/intconstant](#) or **[/nointconstant](#)**

The [/intconstant](#) option requests that Fortran 77 semantics (type determined by the value) be used to determine the kind of integer constants instead of Fortran 90 default INTEGER type. If you do not specify [/intconstant](#), the type is determined by the default INTEGER type.

In Developer Studio, specify Use F77 Integer Constants in the Compatibility [Compiler Option Category](#).

[/integer_size](#)

Syntax:

`/integer_size:size` or `/I2`

The `/integer_size` or `/I2` option specifies the size (in bits) of integer and logical declarations, constants, functions, and intrinsics. In Developer Studio, specify the Default Integer Kind in the Fortran Data [Compiler Option Category](#). These options are:

- `/integer_size:16` or `/I2` makes the default integer and logical variables 2 bytes long. INTEGER and LOGICAL declarations are treated as (KIND=2).
- `/integer_size:32` makes the default integer and logical variables 4 bytes long (default). INTEGER and LOGICAL declarations are treated as (KIND=4).
- `/integer_size:64` (Alpha only) makes the default integer and logical variables 8 bytes long. INTEGER and LOGICAL declarations are treated as (KIND=8).

`/[no]keep`

Syntax:

`/keep` or `/nokeep`

The `/keep` option creates one object file for each input source file specified, which may not be desirable when compiling multiple source files. The `/keep` option does not remove temporary files, which might be created by the FPP preprocessor or the DIGITAL Fortran compiler.

If the `/keep` option is specified, the FPP output files and object files are created in the current directory and retained. The `/keep` option also affects the number of files that are created and the file names used for these files.

`/[no]libdir`

Syntax:

`/libdir[:keyword]`, `/nolibdir`, or `/Zl` or `/Zla`

The `/libdir`, `/Zl`, or `/Zla` option controls whether library search paths are placed into object files generated by the compiler. Specify one or more of the following options:

- Specify `/libdir:all` or `/libdir` to request the insertion of linker search path directives for libraries automatically determined by the DF command driver and for those specified by the `cDEC$` `OBJCOMMENT LIB` source directives. Specifying `/libdir:all` is equivalent to `/libdir:(automatic, user)`. This is the default.
- Specify `/libdir:none`, `/nolibdir`, or `/Zla` to prevent *all* linker search path directives from being inserted into the object file (neither automatic nor user specified).
- Specify `/libdir:automatic` to request the insertion of linker search path directives for libraries automatically determined by the DF command driver (default libraries). To prevent the insertion of linker directives for default libraries, specify `/libdir:noautomatic` or `/Zl`. In Developer Studio, specify Disable Default Library Search Rules (for `/libdir:noautomatic`) in the Libraries [Compiler Option Category](#).
- Specify `/libdir:user` to allow insertion of linker search path directives for any libraries specified

by the `cDEC$ OBJCOMMENT LIB` source directives. To prevent the insertion of linker directives for any libraries specified by the `OBJCOMMENT` directives, specify `/libdir:nouser`. In Developer Studio, specify `Disable OBJCOMMENT Directives (for /libdir:nouser)` in the Libraries Compiler Option Category.

/libs

Syntax:

`/libs[:keyword], /MD, /MDd, /MDs, /ML, /MLd, /MT, /MTd, /MTs, /MW, or /MWs`

The `/libs` option controls the type of libraries your application is linked with. The default is `/libs:static` (same as `/libs`). In Developer Studio, specify the `Use Fortran Run-Time Libraries` in the Libraries Compiler Option Category. These options are:

- `/libs:dll` or `/MDs`
The `/libs:dll` or `/MDs` option causes the linker to search for unresolved references in single threaded, dynamic link reference libraries (DLLs). If the unresolved reference is found in the DLL, it gets resolved when the program is executed (during program loading), reducing executable program size.
Specifying `/libs:dll` with `/threads` is equivalent to `/MD`.
Specifying `/libs:dll` with `/threads` and `/dbglibs` is equivalent to `/MDd`.
- `/libs:static` or `/ML`
The `/libs:static` or `/ML` option requests that the linker searches only in single threaded, static libraries for unresolved references; dynamic link libraries (DLLs) are not searched. This is the default. Specifying `/libs` (with no keyword) is the same as specifying `/libs:static`.
Specifying `/libs:static` with `/nothreads` is equivalent to `/ML`.
Specifying `/libs:static` with `/nothreads` and `/dbglibs` is equivalent to `/MLd`.
Specifying `/libs:static` with `/threads` is equivalent to `/MT`.
Specifying `/libs:static` with `/threads` and `/dbglibs` is equivalent to `/MTd`.
- `/libs:qwin` or `/MW`
Specifying `/libs:qwin` or `/MW` requests the creation of a QuickWin multi-doc (QuickWin) application.
- `/libs:qwins` or `/MWs`
Specifying `/libs:qwins` or `/MWs` requests the creation of a Standard Graphics (QuickWin single-doc) application.

The following related options request additional libraries to link against:

- `/dbglibs`
- `/threads`
- `/winapp`
- `/fpscomp:libs`

To request the creation of a dynamic-link library, see `/dll`.

For information about compiling and linking Visual Fortran and Visual C++ programs (and the libraries used), see Visual Fortran/Visual C++ Mixed-Language Programs.

For command-line examples of using the `/libs` option, see [Linking Against Additional Libraries](#).

`/[no]link`

Syntax:

`/link:options` or **`/nolink`**

The `/link` option (without specifying *options*) precedes options to be passed to the linker as is (see [Linker Options](#)). You can also specify the *options* to be passed to the linker as is using the form: `/link:options`. In Developer Studio, you can specify linker options using the Linker tab in the Project menu Settings dialog box.

The `/nolink` option suppresses linking and forces an object file to be produced even if only one program is compiled. Any options specified after the `/nolink` option are ignored.

`/[no]list`

Syntax:

`/list[:file]`, **`/nolist`**, or **`/Fsfile`**

The `/list` or `/Fs` option creates a listing of the source file with compile-time information appended. To name the source listing file, specify *file*. If you omit the `/list` or `/Fs` options (or specify `/nolist`), no listing file is created.

In Developer Studio, specify Source Listing in the Listing File [Compiler Option Category](#).

When a diagnostic message is displayed, the listing file contains a column pointer (such as1) that points to the specific part of the source line that caused the error. To request a listing with Assembly instructions, see [asmfile](#).

The name of the listing file is the same as the source file (unless specified by *file*), with the extension `.LST` (unless the extension is specified by *file*).

If multiple object files are created, multiple listing files are usually created. For example, if you specify multiple source files with the `/compile_only` and `/list` options without a named object file (`/object:file`), multiple files are created. If you specify multiple source files with the `/list`, `/compile_only`, and `/object:file`, a single listing file is created. For command-line examples, see [Generating a Listing File](#).

`/[no]logo`

Syntax:

`/nologo` or **`/logo`**

The `/nologo` option suppresses the copyright notice displayed by the compiler and linker. This option can be placed anywhere on the command line.

`/[no]machine_code`

Syntax:

`/machine_code` or **`/nomachine_code`**

The `/machine_code` option requests that a machine language representation be included in the listing file. The `/machine_code` option is a synonym for `/show:code`. In Developer Studio, specify Source Listing Options, Machine Code in the Listing File [Compiler Option Category](#).

This option is ignored unless you specify `/list[:file]` or `/Fsfile`.

`/[no]map`

Syntax:

`/map[:file]`, **`/nomap`**, or **`/Fmfile`**

The `/map` or `/Fm` option controls whether or not a link map is created. To name the map file, specify *file*. In Developer Studio, you can specify the Generate mapfile option in the Linker tab of the Project menu Settings dialog box.

If you omit `/map` or `/Fm`, a map file is not created.

`/math_library`

Syntax:

`/math_library:keyword`

The `/math_library` option specifies whether argument checking of math routines is done on *x86* systems and the type of math library routines used on Alpha systems.

In Developer Studio, specify the Math Library in the Optimizations [Compiler Option Category](#). The `/math_library` options are:

- `/math_library:accurate` (Alpha only)
On Alpha systems, specifying `/math_library:accurate` uses the standard math library routines for Fortran intrinsics (for example, SIN), that provide highly accurate answers with good performance and error checking. This is the default on Alpha systems (unless the `/fast` option is specified).

The standard math library routines are designed to obtain very accurate "near correctly rounded" results and provide the robustness needed to check for IEEE exceptional argument values, rather than achieve the fastest possible run-time execution speed. Using `/math_library:accurate` allows user control of arithmetic exception handling with the [/fpe:level](#) option (in addition to the default).

- `/math_library:fast`
On *x86* systems, `/math_library:fast` improves performance by not checking the arguments to the math routines. Using `/math_library:fast` makes tracing the cause of unexpected exceptional values results difficult. On *x86* systems, `/math_library:fast` does not change the accuracy of calculated floating-point numbers.

On Alpha systems, `/math_library:fast` improves performance by using tuned routines in the math library. These routines trade off a small amount of accuracy and less reliable arithmetic exception handling for improved performance. There are tuned routines for such intrinsic procedures as `SQRT` and `EXP`, allowing certain math library functions to get significant performance improvements when the applicable intrinsic procedure is used.

The fast math library routines on Alpha systems do not necessarily check for IEEE exceptional values and should not be used with the `/fpe:level` option other than `fpe:0`.

When you use `/math_library:fast` on Alpha systems, you should carefully check the calculated output from your program to verify that it is not relying on the full fractional accuracy of the floating-point data type to produce correct results and not producing unexpected exceptional values (exception handling is indeterminate).

Programs that do not produce acceptable results on Alpha systems with `/math_library:fast` and single-precision data might produce acceptable results with `/math_library:fast` if they are modified (or compiled) to use double-precision data.

- `/math_library:check`

On x86 systems, `/math_library:check` validates the arguments to and results from calls to the Fortran math routines. This provides slower run-time performance than `/math_library:fast` on x86 systems, but with earlier detection of exceptional values. This is the default on x86 systems.

On Alpha systems, `/math_library:check` is equivalent to `/math_library:accurate` (see previous description of `/math_library:accurate`).

`/[no]module`

Syntax:

`/module[:path]` or `/nomodule`

The `/module` option controls where the module files (extension `MOD`) are placed. If you omit this option (or specify `/nomodule`), the `.MOD` files are placed in the directory where the source file being compiled resides.

When `/module:path` is specified, the `path` specifies the directory location where the module files will be placed.

In Developer Studio, specify the Module Path in the Miscellaneous Compiler Option Category.

When `/module` is entered without specifying a path, it is interpreted as a request to place the `MOD` files in the same location that the object is being created. Should a `path` be specified on the `/object` option, that location would also be used for the `MOD` files.

You need to ensure that the module files are created before they are referenced when using the `DF` command (see Compile With Appropriate Options and Multiple Source Files).

`/names`

Syntax:

/names:keyword, /GNa, /GNI, or /GNu

The `/names` option specifies how source code identifiers and external names are interpreted and the case convention used for external names. This naming convention applies whether names are being defined or referenced. The default is `/names:uppercase` (same as `/GNu`).

In Developer Studio, specify the Name Interpretation in the External Procedures or the Fortran Language Compiler Option Category. The `/names` options are:

- `/names:as_is` or `/GNa` causes the compiler to:
 - Distinguish between uppercase and lowercase letters in source code identifiers (treat uppercase and lowercase letters as different).
 - Distinguish between uppercase and lowercase letters in external names.
- `/names:lowercase` or `/GNI` causes the compiler to:
 - Not distinguish between uppercase and lowercase letters in source code identifiers (treat lowercase and uppercase letters as equivalent).
 - Force all letters to be lowercase in external names.
- `/names:uppercase` or `/GNu` (default) causes the compiler to:
 - Not distinguish between uppercase and lowercase letters in source code identifiers (treat lowercase and uppercase letters as equivalent).
 - Force all letters to be uppercase in external names.

`/[no]object`

Syntax:

/object[:filename], /noobject, or /Fofilename

The `/object` or `/Fo` option names the object file *filename*. Specify `/noobject` to prevent creation of an object file. The default is `/object`, where the file name is the same as the first source file with a file extension of `.OBJ`.

If you omit `/compile_only` (or `/c`) and specify `/object:filename` or `/Fofilename`, the `/object` option names the object file *filename*.

If you specify `/object:filename` or `/Fofilename` and specify the `/compile_only` option, the `/object` or `/Fo` option causes multiple Fortran input files (if specified) to be compiled into a single object file. This allows interprocedural optimizations to occur at higher optimization levels, which usually improves run-time performance.

For information on where module files are placed, see `/module[:path]`.

`/[no]optimize`

Syntax:

/optimize[:level], /nooptimize, /Od, /Ox, or /Oxp

The `/optimize` option controls the level of optimization performed by the compiler. To provide efficient run-time performance, DIGITAL Fortran increases compile time in favor of decreasing run time. If an operation can be performed, eliminated, or simplified at compile time, the compiler does so rather than have it done at run time. Also, the size of object file usually increases when certain optimizations occur (such as with more loop unrolling and more inlined procedures).

In Developer Studio, specify the Optimization Level in the General or Optimizations Compiler Option Category. The `/optimize` options are:

- `/optimize:0` or `/Od`
Disables nearly all optimizations. This is the default if you specify `/debug` (with no keyword). Specifying this option causes certain `/warn` options to be ignored. Specifying `/Od` sets the `/optimize:0` and `/math_library:check` options.
- `/optimize:1`
Enables local optimizations within the source program unit, recognition of common subexpressions, and expansion of integer multiplication and division (using shifts).
- `/optimize:2`
Enables global optimization. This includes data-flow analysis, code motion, strength reduction and test replacement, split-lifetime analysis, and instruction scheduling. Specifying `/optimize:2` includes the optimizations performed by `/optimize:1`.
- `/optimize:3`
Enables additional global optimizations that improve speed (at the cost of extra code size). These optimizations include:
 - Loop unrolling, including instruction scheduling
 - Code replication to eliminate branches
 - Padding the size of certain power-of-two arrays to allow more efficient cache use (see Use Arrays Efficiently)

Specifying `/optimize:3` includes the optimizations performed by `/optimize:1` and `/optimize:2`.

- `/optimize:4`, `/Ox`, and `/Oxp`
Enables interprocedure analysis and automatic inlining of small procedures (with heuristics limiting the amount of extra code). Specifying `/optimize:4` includes the optimizations performed by `/optimize:1`, `/optimize:2`, and `/optimize:3`. For the DF command, `/optimize:4` is the default unless you specify `/debug` (with no keyword). Specifying `/Ox` sets: `/optimize:4`, `/math_library:check`, and `/assume:nodummy_aliases`. Specifying `/Oxp` sets: `/optimize:4`, `/math_library:check`, `/assume:nodummy_aliases`, and `/fpconsistency` (x86 systems).
- `/optimize:5` (Alpha only)
Activates the loop transformation optimizations (also set by transform loops) and the software pipelining optimization (also set by pipeline):
 - The loop transformation optimizations are a group of optimizations that apply to array references within loops. These optimizations can improve the performance of the memory system and can apply to multiple nested loops. Loop transformation optimizations include loop blocking, loop distribution, loop fusion, loop interchange, loop scalar replacement, and outer loop unrolling. You can

specify loop transformation optimizations without software pipelining (see [/\[no\]transform_loops](#)).

- The software pipelining optimization applies instruction scheduling to certain innermost loops, allowing instructions within a loop to "wrap around" and execute in a different iteration of the loop. This can reduce the impact of long-latency operations, resulting in faster loop execution. Software pipelining also enables the prefetching of data to reduce the impact of cache misses.

You can specify software pipelining without loop transformation optimizations (see [/\[no\]pipeline](#)).

In addition to loop transformation and software pipelining, specifying `/optimize:5` activates certain optimizations that are not activated by `/transform_loops` and `/pipeline`, including byte-vectorization, and insertion of additional NOP (No Operation) instructions for alignment of multi-issue sequences.

To determine whether using `/optimize:5` benefits your particular program, you should compare program execution timings for the same program (or subprogram) compiled at levels `/optimize:4` and `/optimize:5`.

Specifying `/optimize:5` (Alpha systems only) includes the optimizations performed by `/optimize:1` `/optimize:2`, `/optimize:3`, and `/optimize:4`.

For detailed information on these optimizations, see [Optimization Levels: the /optimize Option](#)

To compile your application for efficient run-time performance, see [Compile With Appropriate Options and Multiple Source Files](#).

`/[no]pad_source`

Syntax:

`/pad_source` or `/nopad_source`

The `/pad_source` option requests that source records shorter than the statement field width are to be padded with spaces on the right out to the end of the statement field. This affects the interpretation of character and Hollerith literals that are continued across source records.

In Developer Studio, specify the Pad Fixed-Form Source Records in the Fortran Language [Compiler Option Category](#).

The default is `/nopad_source`, which causes a warning message to be displayed if a character or Hollerith literal that ends before the statement field ends is continued onto the next source record. To suppress this warning message, specify the `/warn:nousage` option.

Specifying `/pad_source` can prevent warning messages associated with `/warn:usage`.

`/[no]pdbfile`

Syntax:

`/pdbfile[:filename]`, `/nopdbfile`, or `/Fdfilename`

The `/pdbfile` or `/Fd` option indicates that any debug information generated by the compiler should be to a program database file, *filename.PDB*. If you omit *filename*, the default file name used is `df50.pdb`.

In Developer Studio, specify Use Program Database for Debug Information (and optionally specify the Program Database .PDB Path) in the Debug Compiler Option Category.

When full debug information is requested (`/debug:full`, `/debug`, or equivalent), the debug information is placed in the PDB file (unless `/nopdbfile` is specified).

The compiler places debug information in the object file if you specify `/nopdbfile` or omit both `/pdbfile` and `/debug:full` (or equivalent).

`/[no]pipeline (Alpha only)`

Syntax:

`/pipeline` or `/nopipeline`

On Alpha systems, the `/pipeline` (or `/optimize:5`) option activates the software pipelining optimization. This optimization applies instruction scheduling to certain innermost loops, allowing instructions within a loop to "wrap around" and execute in a different iteration of the loop. This can reduce the impact of long-latency operations, resulting in faster loop execution.

In Developer Studio, specify the Apply Software Pipelining Optimizations in the Optimizations Compiler Option Category.

For this version of Visual Fortran, loops chosen for software pipelining are always innermost loops and do not contain branches, procedure calls, or COMPLEX floating-point data.

Software pipelining can be more effective when you combine `/pipeline` with the appropriate `/tune:keyword` keyword option for the target Alpha processor generation.

Software pipelining also enables the prefetching of data to reduce the impact of cache misses.

Software pipelining is a subset of the optimizations activated by `/optimize:5`. Instead of specifying both `/pipeline` and `/transform_loops`, you can specify `/optimize:5`.

To specify software pipelining without loop transformation optimizations, do one of the following:

- Specify `/optimize:5` with `/notransform_loops` (preferred method)
- Specify `/pipeline` with `/optimize:4`, `/optimize:3`, or `/optimize:2`. This optimization is not performed at optimization levels below `/optimize:2`.

To determine whether using `/pipeline` benefits your particular program, you should time program execution for the same program (or subprogram) compiled with and without software pipelining (such as with `/pipeline` and `/nonopipeline`).

For programs that contain loops that exhaust available registers, longer execution times may result with `/optimize:5`, requiring use of `/unroll:count` to limit loop unrolling. The `/optimize:5` option applies only to Alpha systems.

For more information, see [Software Pipelining](#).

/preprocess_only

Syntax:

/preprocess_only

The `/preprocess_only` option runs only the FPP preprocessor and puts the result for each source file in a corresponding `.i` or `.i90` file. The `.i` or `.i90` file does not have line numbers (`#`) in it.

/real_size

Syntax:

/real_size:size or **/4R8**

The `/real_size` or `/4R8` option controls the *size* (in bits) of REAL and COMPLEX declarations, constants, functions, and intrinsics. In Developer Studio, specify the Default Real Kind in the Fortran Data [Compiler Option Category](#). The `/real_size` options are:

- `/real_size:32`
Defines REAL declarations, constants, functions, and intrinsics as REAL(KIND=4) (SINGLE PRECISION). It also defines COMPLEX declarations, constants, functions, and intrinsics as COMPLEX(KIND=4) (COMPLEX). This is the default.
- `/real_size:64` or `/4R8`
Defines REAL declarations, constants, functions, and intrinsics as REAL(KIND=8) (DOUBLE PRECISION). It also defines COMPLEX declarations, constants, functions, and intrinsics as COMPLEX(KIND=8).

Specifying `/real_size:64` causes intrinsic functions to produce a REAL(KIND=8) or COMPLEX(KIND=8) result instead of a REAL(KIND=4) or COMPLEX(KIND=4) result, except if the argument is explicitly typed as REAL(KIND=4) or COMPLEX(KIND=4), including CMPLX, FLOAT, REAL, SNGL, and AIMAG. For instance, references to the CMPLX intrinsic produce DCMPLX results (COMPLEX(KIND=8)), except if the argument to CMPLX is explicitly typed as REAL(KIND=4), REAL*4, COMPLEX(KIND=4), or COMPLEX*8. In this case the resulting data type is COMPLEX(KIND=4).

/[no]recursive

Syntax:

/recursive or **/norecursive**

The `/recursive` option compiles all procedures (functions and subroutines) for possible recursive execution. Specifying the `/recursive` option sets the [/automatic](#) option. The default is `/norecursive`.

In Developer Studio, specify Enable Recursive Routines in the Run time [Compiler Option Category](#).

/[no]reentrancy

Syntax:

/reentrancy[:keyword] or /noreentrancy

The `/reentrancy` or `/reentrancy:threads` option requests that the compiler generate reentrant code that supports a multithreaded application. In Developer Studio, specify the Enable Reentrancy Support or Disable Reentrancy Support in the Libraries [Compiler Option Category](#).

If you omit `/reentrancy`, `/reentrancy:threads`, or `/threads`, `/reentrancy:none` (same as `/noreentrancy`) is used.

Specifying [/threads](#) sets `/reentrancy:threads`, since multithreaded code must be reentrant.

/rounding_mode (Alpha only)

Syntax:

/rounding_mode:keyword

On Alpha systems, the `/rounding_mode` option allows you to control how rounding occurs during floating-point calculations. The rounding mode applies to each program unit being compiled.

In Developer Studio, specify the Rounding Mode in the Floating Point [Compiler Option Category](#). The `/rounding_mode` options are:

- `/rounding_mode:nearest`
The normal rounding mode, where results are rounded to the nearest representable value. If you omit other `/rounding_mode` options, `/rounding_mode:nearest` is used.
- `/rounding_mode:chopped`
Rounds results toward zero.
- `/rounding_mode:minus_infinity`
Rounds results toward the next smallest representative value.
- `/rounding_mode:dynamic`
Lets you set the rounding mode at run-time. You can modify your program to call the appropriate Windows NT Alpha routine to obtain or set the current rounding mode.

When you call the appropriate Windows NT Alpha routine (such as `_controlfp` or `_control87`), you can set the rounding mode to one of the following settings:

- Round toward zero or truncate (same as `/rounding_mode:chopped`)
- Round toward nearest (same as `/rounding_mode:nearest`)
- Round toward plus infinity
- Round toward minus infinity (same as `/rounding_mode:minus_infinity`)

If you compile with `/rounding_mode:dynamic` and do not call the appropriate Windows NT routine, the initial rounding mode is round toward nearest `/rounding_mode:nearest`.

For the fastest run-time performance, avoid using `/rounding_mode:dynamic`.

For information on setting the rounding mode on x86 systems, see [Floating-Point Control Word \(x86 only\)](#).

/[no]show

Syntax:

/show:keyword... or **/noshow**

The /show option specifies what information is included in a listing. In Developer Studio, specify the Source Listing Options in the Listing File Compiler Option Category. The /show keywords are:

- /show:code
Includes a machine-language representation of the compiled code in the listing file. The default is /show:nocode. The /show:code and /machine_code options are equivalent.
- /show:include
Lists any text file included in the source file (unless that source is included using the INCLUDE 'filespec /NOLIST' syntax; see the vms option). The default is /show:noinclude.
- /show:map (default)
Includes a symbol map in the listing file.
- /show:nomap
Do not include a symbol map in the listing file.

Specifying /show or /show:all is equivalent to /show:(code,include,map). Specifying /noshow or /show:none is equivalent to /show:(nocode,noinclude,nomap).

This option is ignored unless you specify /list[:file] or /Fsfile.

/source

Syntax:

/source:file or **/Tffile**

The /source or /Tf option indicates that the *file* is a Fortran source file with a non-standard file extension (not one of .F, .FOR, or .F90) that needs to be compiled.

The default for any file that does *not* have an extension of .F90 or .f90 is to be a fixed-format Fortran file.

/[no]stand

Syntax:

/stand[:keyword], **/nostand**, or **/4Ns**

The /stand or /4Ns option issues compile-time messages for language elements that are not standard in the Fortran 90 or Fortran 95 language that can be identified at compile-time. In Developer Studio, specify the Fortran Standards Checking in the Fortran Language Compiler Option Category. These options are:

- Specify /stand or /stand:f90 to request that diagnostic messages be generated with a warning-level severity (allows an object file to be created) for extensions to the Fortran 90 standard.

- Specify `/stand:f95` to request that diagnostic messages be generated with a warning-level severity (allows an object file to be created) for extensions to the proposed Fortran 95 standard.
- If you omit the `/stand`, or `/stand:keyword`, `/warn:stderrors`, or `/4Ys` options, messages are not issued for language elements that are not standard in the Fortran 90 or Fortran 95 language. This is equivalent to `/stand:none`, `/nostand`, or `/4Ns`.

Specify `/warn:stderrors` to request that diagnostic messages be generated with an error-level severity (instead of warning) to prevent an object file from being created.

Specifying `/stand` issues warning messages for:

- Obsolescent and deleted features specified by the Fortran standard.
- Syntax extensions to the Fortran 90 standard. Syntax extensions include nonstandard statements and language constructs.
- Fortran 90 standard-conforming statements that become nonstandard due to the way in which they are used. Data type information and statement locations are considered when determining semantic extensions.
- For fixed-format source files, lines that use tab formatting.

Source statements that do not conform to Fortran 90 language standards are detected by the compiler under the following circumstances:

- The statements contain ordinary syntax and semantic errors.
- A source program containing nonconforming statements is compiled with the `/stand` or `/check` options.

Given these circumstances, the compiler is able to detect *most* instances of nonconforming usage. It does not detect all instances because the `/stand` option does not produce checks for all nonconforming usage at compile time. In general, the unchecked cases of nonconforming usage arise from the following situations:

- The standard violation results from conditions that cannot be checked at compile time.
- The compile-time checking is prone to false alarms.

Most of the unchecked cases occur in the interface between calling and called subprograms. However, other cases are not checked, even within a single subprogram.

The following items are known to be unchecked:

- Use of a data item prior to defining it
- Use of the `SAVE` statement to ensure that data items or common blocks retain their values when reinvoked
- Association of character data items on the right and left sides of character assignment statements
- Mismatch in order, number, or type in passing actual arguments to subprograms with implicit interfaces
- Association of one or more actual arguments with a data item in a common block when calling a subprogram that assigns a new value to one or more of the arguments

`/[no]static`

Syntax:

`/static` or `/nostatic`

The `/static` option is the same as the `/noautomatic` option. The default is `/static`, which causes all local variables to be statically allocated. The `/nostatic` option is the same as `/automatic`. In Developer Studio, specify `/nostatic` as Variables Default to Automatic in the Fortran Data Compiler Option Category.

If you specify `/recursive`, the `/automatic` option is set.

`/[no]synchronous_exceptions (Alpha only)`

Syntax:

`/synchronous_exceptions` or `/nosynchronous_exceptions`

On Alpha systems, the `/synchronous_exceptions` option associates an exception with the instruction that causes it. This slows program execution, so only specify it when debugging a specific problem, such as locating the source of an exception.

In Developer Studio, specify Enable Synchronous Floating-Point Exceptions in the Floating Point Compiler Option Category.

If you omit `/synchronous_exceptions`, exceptions can be reported one or more instructions *after* the instruction that caused the exception, depending on the `/fpe:level` option used.

The default is `/nosynchronous_exceptions` (if you specify or imply `/fpe:0`). If you specify a higher `/fpe` level, the default is `/synchronous_exceptions`.

`/[no]syntax_only`

Syntax:

`/syntax_only` or `/nosyntax_only`

The `/syntax_only` option requests that only the syntax of the source file be checked. If the `/syntax_only` option is specified, code generation is suppressed. The default is `/nosyntax_only`.

`/[no]threads`

Syntax:

`/threads` or `/nothreads`

The `/threads` option requests linking with multithreaded libraries, which creates a multithreaded program or DLL. If you specify `/threads`, this sets the `/reentrancy` option.

In Developer Studio, specify Use Multithreaded Library in the Libraries Compiler Option Category.

The default is `/nothreads`, which links with single-threaded libraries to create a single-threaded program or DLL.

Related options that control the libraries used during linking include:

- [/libs](#)
- [/winapp](#)
- [/fpscomp:libs](#)

`/[no]transform_loops` (Alpha only)

Syntax:

`/transform_loops` or `/notransform_loops`

On Alpha systems, the `/transform_loops` (or `/optimize:5`) option activates a group of loop transformation optimizations that apply to array references within loops. These optimizations can improve the performance of the memory system and usually apply to multiple nested loops. The loops chosen for loop transformation optimizations are always *counted loops* (which include DO or IF loops, but not uncounted DO WHILE loops).

In Developer Studio, specify the Apply Loop Transformation Optimizations in the Optimizations [Compiler Option Category](#).

Conditions that typically prevent the loop transformation optimizations from occurring include subprogram references that are not inlined (such as an external function call), complicated exit conditions, and uncounted loops.

The types of optimizations associated with `/transform_loops` include the following:

- Loop blocking
- Loop distribution
- Loop fusion
- Loop interchange
- Loop scalar replacement
- Outer loop unrolling

The loop transformation optimizations are a subset of optimizations activated by `/optimize:5`. Instead of specifying both `/pipeline` and `/transform_loops`, you can specify `/optimize:5`.

To specify loop transformation optimizations without software pipelining, do one of the following:

- Specify `/optimize:5` with `/nopipeline` (preferred method)
- Specify `/transform_loops` with `/optimize:4`, `/optimize:3`, or `/optimize:2`. This optimization is not performed at optimization levels below `/optimize:2`.

To determine whether using `/transform_loops` benefits your particular program, you should time program execution for the same program (or subprogram) compiled with and without loop transformation optimizations (such as with `/transform_loops` and `/notransform_loops`). This option applies only to Alpha systems.

For more information, see [Loop Transformation](#).

`/tune` (Alpha only)

Syntax:

/tune:keyword

On Alpha systems, the `/tune` option specifies the types of processor-specific instruction tuning for implementations of the Alpha architecture. This option is ignored on x86 processor systems.

Regardless of the setting of `/tune:keyword` option you use, the generated code runs correctly on all implementations of the Alpha architecture. Tuning for a specific implementation can improve run-time performance; it is also possible that code tuned for a specific Alpha processor may run slower on another Alpha processor.

If you omit `/tune:keyword`, `/tune:generic` is used. In Developer Studio, specify the Optimize For in the Optimizations [Compiler Option Category](#). The `/tune` options are:

- `/tune:generic`
Generates and schedules code that will execute well for all generations of Alpha processor chips. This provides generally efficient code for those cases where all processor generations are likely to be used. This is the default.
- `/tune:host`
Generates and schedules code optimized for the processor generation in use on the system being used for compilation.
- `/tune:ev4`
Generates and schedules code optimized for the 21064, 21064A, 21066, and 21068 implementations of the Alpha chip.
- `/tune:ev5`
Generates and schedules code optimized for some 21164 implementations of the Alpha architecture that use only the base set of Alpha instructions (no extensions).
- `/tune:ev56`
Generates and schedules code for some 21164 chip implementations that use the byte and word manipulation instruction extensions of the Alpha architecture.
- `/tune:pca56`
Generates and schedules code for the 21164PC chip implementation that uses the byte and word manipulation instruction extensions and multimedia instruction extensions of the Alpha architecture.

To request a specific set of instructions for an Alpha architecture generation, see [/architecture:keyword](#).

`/undefine`

Syntax:

/undefine:symbol

The `/undefine` option removes any initial definition of *symbol* for the FPP preprocessor.

/unroll

Syntax:

/unroll:count

For higher optimization levels, the `/unroll` option allows you to specify how many times loops are unrolled. If the `/unroll` option is not specified, the compiler determines how many times loops are unrolled (4 times for most loops or 2 times for certain loops with large code size or branches outside the loop).

In Developer Studio, specify the Loop Unroll Count in the Optimizations Compiler Option Category.

If the `/optimize:3`, `/optimize:4` (or equivalent), or `/optimize:5` (Alpha systems only) options are specified, loop unrolling occurs. The *count* should be an integer in the range 0 to 16. A count value of 0 is used to indicate that the compiler should determine how many times a loop is unrolled (default).

The compiler attempts to unroll certain innermost loops, minimizing the number of branches and grouping more instructions together to allow efficient overlapped instruction execution (instruction pipelining). The best candidates for loop unrolling are innermost loops with limited control flow.

For more information, see Loop Unrolling.

/[no]vms

Syntax:

/vms or **/novms**

The `/vms` option causes the run-time system to provide functions like DIGITAL Fortran for OpenVMS[™] VAX[™] Systems (previously called VAX FORTRAN[™]).

In Developer Studio, specify Enable VMS Compatibility in the Compatibility Compiler Option Category. The `/vms` option:

- In the absence of other options, sets the following command-line defaults: `/align:norecords`, `/align:nocommons`, `/check:format`, `/check:output_conversion`, `/static`, `/norecursive`, and `/names:lowercase`.
- Allows use of the DELETE statement for relative files. When a record in a relative file is deleted, the first byte of that record is set to a known character (currently '@'). Attempts to read that record later result in ATTACCNON errors. The rest of the record (the whole record when `/vms` is not set) is set to nulls for unformatted files and spaces for formatted files.
- When an ENDFILE is performed on a sequential unit, an actual 1-byte record containing a Ctrl+D (04 hex) is written to the file. When you omit `/vms`, an internal ENDFILE flag is set and the file is truncated. The `/vms` option does not affect ENDFILE on relative files; the file is truncated.
- Enables recognition of certain environment variables at run time for ACCEPT, PRINT, and TYPE statements and for READ and WRITE statements that do not specify a unit number (such as READ (*,1000)).
- Changes certain OPEN statement BLANK keyword defaults. Changes the default

interpretation from BLANK='NULL' to BLANK='ZERO' for an implicit OPEN or internal file OPEN. For an explicit OPEN, the default is always BLANK='NULL'.

- Changes certain OPEN statement effects. If the CARRIAGE CONTROL is defaulted, the file is formatted, and the unit is connected to a terminal, then the carriage control defaults to FORTRAN. Otherwise, it defaults to LIST. The /vms option affects the record length for relative organization files. The buffer size is increased by 1 to accommodate the deleted record character.
- LIST and /NOLIST are recognized at the end of the file specification to the INCLUDE statement at compile time. If you specified /vms and if the file specification does not include the directory path, the current working directory is used as the default directory path. If you omitted /vms, the directory path is where the file that contains the INCLUDE statement resides.
- Changes internal file writes using list-directed I/O. A list-directed write to an internal file results in removal of the first character from the first element; the field length is decremented accordingly.
- The run-time direct access READ routine checks the first byte of the retrieved record. If this byte is '@' or NULL ('\0'), then ATTACCNON is returned. The run-time sequential access READ routine checks to see if the record it just read is 1 byte long and contains a Ctrl+D (04 hex) or a Ctrl+Z (1A hex). If this is true, it returns EOF.

The default is /novms.

/[no]warn

Syntax:

/warn[:keyword...]), /nowarn, /4Yd, /4Nd, /4Ys, /W0, /W1, or /WX

The /warn option instructs the compiler to generate diagnostic messages for defined classes of additional checking that can be performed at compile-time. It also can change the severity level of issued compilation messages.

In Developer Studio, specify the Warning Level in the General Compiler Option Category or specify individual Warning Options in the Miscellaneous Compiler Option Category. The /warn options are:

- /warn:noalignments suppresses warning messages for data that is not naturally aligned. The default is /warn:alignments.
- /warn:argument_checking enables warnings about argument mismatches between callers and callees, when compiled together. The default is /warn:noargument_checking.
- /warn:declarations or /4Yd issues an error message for any undeclared symbols. This option makes the default type of a variable undefined (IMPLICIT NONE) rather than using the default Fortran rules. The default is /warn:nodeclarations or /4Nd.
- /warn:errors or /WX changes the severity of all warning diagnostics into error diagnostics. The default is /warn:noerrors. Specifying /warn:errors (or /WX) sets /warn:stderrs.
- /warn:nofileopt suppresses the display of an informational-level diagnostic message when compiling multiple files separately. The default is /warn:fileopt (displays the message: Some interprocedural optimizations may be disabled when compiling in this mode).
- /warn:nogeneral suppresses all informational-level and warning-level diagnostic messages from the compiler. The default is /warn:general or /W1.
- /warn:nogranularity (Alpha systems only) suppresses the display of a warning message that the

compiler cannot generate code for the requested granularity (see /granularity). The default is /warn:granularity.

- /warn:stderrs or /4Ys requests Fortran 90 standards checking (see /stand) with error-level compilation messages instead of warning-level messages. Specifying /warn:stderrs sets /stand:f90 and is equivalent to /4Ys. Specifying /warn:stderrs with /stand:f95 requests error-level messages for extensions to the proposed Fortran 95 standard. Specifying /warn:errors sets /warn:stderrs. The default is /warn:nostderrors.
- /warn:nouncalled suppresses warning messages for when a statement function is never called. The default is /warn:uncalled.
- /warn:nouninitialized suppresses warning messages for a variable that is used before a value was assigned to it. The default is /warn:uninitialized.
- /warn:nousage suppresses warning messages about questionable programming practices and the use of intrinsic functions that use a two-digit year (year 2000). The questionable programming practices, although allowed, often are the result of programming errors. For example, /warn:usage detects a continued character or Hollerith literal whose first part ends before the statement field ends and appears to end with trailing spaces. The default is /warn:usage. The /pad_source option can prevent warning messages from /warn:usage.
- /warn:all or /warn requests all possible warning messages, but does not set /warn:errors or /warn:stderrs. To enable all the additional checking to be performed and force the severity of the diagnostics to be severe enough to not generate an object file, specify /warn:(all,errors) or /warn:(all,stderrs).
- /warn:none, /nowarn, or /W0 suppresses all warning messages.

If you omit /warn, the defaults are:

- DF command:
/warn:(alignments,noargument_checking,nodeclarations,noerrors,fileopts,general,granularity,nos
- FL32 command:
/warn:(alignments,argument_checking,nodeclarations,noerrors,nofileopts,general,granularity,nos

/[no]watch

Syntax:

/watch[:keyword] or **/nowatch**

The /watch option requests the display of processing information to the console terminal. You can request the display of the passes (compiler, linker) with their respective command arguments and/or the input and output files by specifying any of the following:

- Specify /watch:cmd to display the passes (compiler, linker) with the respective command arguments.
- Specify /watch:source to display the names of sources file(s) being processed. Source file names are listed one per line. This is the default.
- Specify /watch:all or /watch to request /watch:(cmd, source). This displays both pass information and source file names.
- Specify /nowatch or /watch:none to request /watch:(nocmd, nosource).

/what

Syntax:

/what

The /what option displays Visual Fortran version number information.

/winapp

Syntax:

/winapp or **/MG**

The /winapp or /MG option requests the creation of a graphics or windows application and links against the most commonly used libraries. In Developer Studio, specify the Use Common Windows Libraries in the Libraries Compiler Option Category.

The following related options request libraries:

- /libs
- /threads
- /fpscomp:libs

For information on Windows Applications, including requesting additional link libraries with the FULLAPI.F90 file, see Creating Windows Applications.

Linker Options and Related Information

You can set Linker options from:

- The DF command line.

When using the DF command line, specify linker options *after* the /LINK option. For example:

```
DF file.f90 file.lib /LINK /NODEFAULTLIB
```

- The LINK command line.

You can specify linker options and libraries with the LINK command. For example:

```
LINK file.obj file.lib /NODEFAULTLIB
```

- Within Microsoft Developer Studio, in the Project menu, Settings dialog box

You can specify linker options and libraries by using the Linker tab in the Project menu, Settings dialog box.

This table describes the Linker options and how they are used.

LINK option	Function
<u>/ALIGN</u>	Specifies the alignment of each section within the linear address space of the program.

<u>/COMMENT</u>	Inserts a comment string into the header of an executable file or DLL, after the array of section headers.
<u>/DEF</u>	Passes a module-definition (.DEF) file to the linker.
<u>/DEFAULTLIB</u>	Adds one or more <i>libraries</i> to the list of libraries that LINK searches when resolving references.
<u>/DLL</u>	Builds a DLL as the main output file.
<u>/EXPORT</u>	Exports a function from your program.
<u>/FIXED</u>	Tells the operating system to load the program only at its preferred base address.
<u>/HEAP</u>	Sets the size of the heap in bytes.
<u>/IMPLIB</u>	Sets the name for the import library that LINK creates when it builds a program that contains exports.
<u>/NOENTRY</u>	Prevents LINK from linking a reference to <code>_main</code> into the DLL.
<u>/OUT</u>	Overrides the default name and location of the image file that LINK creates.
<u>/RELEASE</u>	Sets the checksum in the header of an executable file.
<u>/SUBSYSTEM</u>	Tells the operating system how to run the executable file.
<u>/WARN</u>	Determines the output of LINK warnings.

This table lists the Linker options, along with the equivalent Microsoft Developer Studio option if one is available. Options listed as command-line only can be entered in the "Common Options" text box of the Project ... Settings dialog box. For instructions on how to work with the Microsoft Developer Studio environment, see the [Developer Studio Environment User's Guide](#).

Command-Line Option	Microsoft Developer Studio Option
<u>/ALIGN</u>	Command-line only
<u>/BASE</u>	Output Category
<u>/COMMENT</u>	Command-line only
<u>/DEBUG</u>	Debug Category
<u>/DEBUGTYPE</u>	Debug Category
<u>/DEF</u>	Command-line only
<u>/DEFAULTLIB</u>	Command-line only
<u>/DLL</u>	Command-line only
<u>/ENTRY</u>	Output Category
<u>/EXPORT</u>	Command-line only
<u>/FIXED</u>	Command-line only
<u>/FORCE</u>	Customize Category
<u>/HEAP</u>	Command-line only
<u>/IMPLIB</u>	Command-line only
<u>/INCLUDE</u>	Input Category
<u>/INCREMENTAL</u>	Customize Category
<u>/MAP</u>	Debug Category
<u>/NODEFAULTLIB</u>	Input Category
<u>/NOENTRY</u>	Command-line only
<u>/NOLOGO</u>	Customize Category
<u>/OPT</u>	Command-line only
<u>/ORDER</u>	Command-line only

<u>/OUT</u>	Customize Category
<u>/PDB</u>	Customize Category
<u>/PROFILE</u>	General Category
<u>/RELEASE</u>	Command-line only
<u>/STACK</u>	Output Category
<u>/STUB</u>	Input Category
<u>/SUBSYSTEM</u>	Command-line only
<u>/VERBOSE</u>	Customize Category
<u>/VERSION</u>	Output Category
<u>/WARN</u>	Command-line only

Besides discussing linker options individually, this section also discusses Module-Definition Files and Linker Reserved Words.

Setting LINK Options in Microsoft Developer Studio

You can set linker options in Microsoft Developer Studio by using the Link tab in the Build Settings dialog box. The following tables list the linker options by category in Microsoft Developer Studio, along with the equivalent command-line options:

General category	Command-line equivalent
Output File Name	<i>/OUT:filename</i>
Object/Library Modules	<i>filename</i> on command line
Generate Debug Info	<i>/DEBUG</i>
Ignore All Default Libraries	<i>/NODEFAULTLIB</i>
Link Incrementally	<i>/INCREMENTAL:{YES NO}</i>
Generate Mapfile	<i>/MAP</i>
Enable Profiling	<i>/PROFILE</i>

Output category	Command-line equivalent
Base Address	<i>/BASE:address</i>
Entry-Point Symbol	<i>/ENTRY:function</i>
Stack Allocations	<i>/STACK:reserve,commit</i>
Version Information	<i>/VERSION:major.minor</i>

Input category	Command-line equivalent
Object/Library Modules	<i>filename</i> on command line
Ignore Libraries	<i>/NODEFAULTLIB:library</i>
Ignore All Default Libraries	<i>/NODEFAULTLIB</i>
Force Symbol References	<i>/INCLUDE:symbol</i>
MS-DOS Stub File Name	<i>/STUB:filename</i>

Customize category	Command-line equivalent
Use Program Database	<i>/PDB:filename</i>
Link Incrementally	<i>/INCREMENTAL:{YES NO}</i>

Program Database Name	/PDB: <i>filename</i>
Output File Name	/OUT: <i>filename</i>
Force File Output	/FORCE
Print Progress Messages	/VERBOSE
Suppress Startup Banner	/NOLOGO

Debug category	Command-line equivalent
Mapfile Name	/MAP: <i>filename</i>
Generate Mapfile	/MAP
Generate Debug Info	/DEBUG
Microsoft Format	/DEBUGTYPE:CV
COFF Format	/DEBUGTYPE:COFF
Both Formats	/DEBUGTYPE:BOTH

Rules for LINK Options

An option consists of an option specifier, either a dash (-) or a forward slash (/), followed by the name of the option. Option names cannot be abbreviated. Some options take an argument, specified after a colon (:). No spaces or tabs are allowed within an option specification, except within a quoted string in the /COMMENT option.

Specify numeric arguments in decimal or C-language notation. (The digits 1-9 specify decimal values, an integer constant preceded by a zero (0) specifies an octal value, and an integer constant preceded by zero and x (0x or 0X) specifies a hexadecimal value.) Option names and their keyword or filename arguments are not case sensitive, but identifiers as arguments are case sensitive.

LINK first processes options specified in the LINK environment variable. Next, LINK processes options in the order specified on the command line and in command files. If an option is repeated with different arguments, the last one processed takes precedence.

Options apply to the entire build. No options can be applied to specific input files.

/ALIGN

Syntax:

/ALIGN:*number*

Specifies the alignment of each section within the linear address space of the program. The *number* argument is in bytes and must be a power of 2. The default is 4K. The linker generates a warning if the alignment produces an invalid image.

/BASE

Syntax:

/BASE:{*address* | @*filename,key*}

This option sets a base address for the program, overriding the default location for an executable file (at 0x400000) or a DLL (at 0x10000000). The operating system first attempts to load a program at its specified or default base address. If sufficient space is not available there, the system relocates the program. To prevent relocation, use the /FIXED option.

Specify the preferred base address in the text box (or in the *address* argument on the command line). The linker rounds the specified number up to the nearest multiple of 64K.

Another way to specify the base address is by using a *filename*, preceded by an at sign (@), and a *key* into the file. The *filename* is a text file that contains the locations and sizes of all DLLs your program uses. The linker looks for *filename* in either the specified path or, if no path is specified, in directories named in the LIB environment variable. Each line in *filename* represents one DLL and has the following syntax:

key address size ;comment

The *key* is a string of alphanumeric characters and is not case sensitive. It is usually the name of a DLL but it need not be. The *key* is followed by a base *address* in C-notation hexadecimal or decimal and a maximum *size*. All three arguments are separated by spaces or tabs. The linker issues a warning if the specified *size* is less than the virtual address space required by the program. Indicate a *comment* by a semicolon (;). Comments can be on the same or a separate line. The linker ignores all text from the semicolon to the end of the line. The following example shows part of such a file:

```
main    0x00010000    0x08000000    ; for PROJECT.EXE
one     0x28000000    0x00100000    ; for DLLONE.DLL
two     0x28100000    0x00300000    ; for DLLTWO.DLL
```

If the file that contains these lines is called DLLS.TXT, the following example command applies this information.

```
link dlltwo.obj /dll /base:dlls.txt,two
```

You can reduce paging and improve performance of your program by assigning base addresses so that DLLs do not overlap in the address space.

An alternate way to set the base address is with the **BASE** argument in a **NAME** or **LIBRARY** module-definition statement. The /BASE and /DLL options together are equivalent to the **LIBRARY** statement. For information on module-definition statements, see Module-Definition Files.

/COMMENT

Syntax:

/COMMENT:[*"*] *comment* [*"*]

Inserts a comment string into the header of an executable file or DLL, after the array of section headers. The type of operating system determines whether the string is loaded into memory. This comment string, unlike the comment specified with DESCRIPTION in a .DEF file, is not inserted into the data section. Comments are useful for embedding copyright and version information.

To specify a *comment* that contains spaces or tabs, enclose it in double quotation marks ("). LINK

removes the quotation marks before inserting the string. If more than one /COMMENT option is specified, LINK concatenates the strings and places a null byte at the end of each string.

/DEBUG

Syntax:

/DEBUG

This option creates debugging information for the executable file or DLL.

The linker puts the debugging information into a program database (PDB). It updates the program database during subsequent builds of the program. For details about PDBs, see [/PDB](#).

An executable file or DLL created for debugging contains the name and path of the corresponding PDB. Visual Fortran reads the embedded name and uses the PDB when you debug the program. The linker uses the base name of the program and the extension .PDB to name the PDB, and embeds the path where it was created. To override this default, use */PDB:filename*.

The object files must contain debugging information. Use the compiler's /Zi (Program Database), /Zd (Line Numbers Only), or /Z7 (C7 Compatible) option. If an object (whether specified explicitly or supplied from a library) was compiled with Program Database, its debugging information is stored in a PDB for the object file, and the name and location of the .PDB file is embedded in the object. The linker looks for the object's PDB first in the absolute path written in the object file and then in the directory that contains the object file. You cannot specify a PDB's filename or location to the linker.

If you have turned off Use Program Database (or specified /PDB:NONE on the command line), or if you have chosen either /DEBUGTYPE:COFF or /DEBUGTYPE:BOTH, the linker does not create a PDB but instead puts the debugging information into the executable file or DLL.

The /DEBUG option changes the default for the [/OPT](#) option from REF to NOREF.

/DEBUGTYPE

Syntax:

/DEBUGTYPE:{CV|COFF|BOTH}

This option generates debugging information in one of three ways: Microsoft format (CV), COFF format, or both:

- **/DEBUGTYPE:CV**

Visual Fortran requires new-style Microsoft Symbolic Debugging Information in order to read a program for debugging. To select this option in Microsoft Developer Studio, choose the Link tab of the Project Settings dialog box. In the Debug category, select the Microsoft Format button, which is only available if you have checked the Generate Debug Info box.

For information on using Microsoft Developer Studio, see the *Microsoft Developer Studio Environment User Guide* in InfoViewer.

- **/DEBUGTYPE:COFF**

This option generates COFF-style debugging information. Some debuggers require Common Object File Format (COFF) debugging information.

When you set this option, the linker does not create a PDB; in addition, incremental linking is disabled.

To select this option in Microsoft Developer Studio, choose the Link tab of the Project Settings dialog box. In the Debug category, select the COFF Format button, which is only available if you have checked the Generate Debug Info box. For information on using Microsoft Developer Studio, see the *Microsoft Developer Studio Environment User Guide* in InfoViewer.

- **/DEBUGTYPE:BOTH**

This option generates both COFF debugging information and old-style Microsoft debugging information.

When you set this option, the linker does not create a PDB; in addition, incremental linking is disabled. The linker must call the CVPACK.EXE tool to process the old-style Microsoft debugging information. CVPACK must be in the same directory as LINK or in a directory in the PATH environment variable.

In Microsoft Developer Studio, specify this option with the Both Formats button, which is only available if you have selected Generate Debug Info. For information on using Microsoft Developer Studio, see the *Microsoft Developer Studio Environment User Guide* in InfoViewer.

If you do not specify /DEBUG, /DEBUGTYPE is ignored. If you specify /DEBUG but not /DEBUGTYPE, the default type is /DEBUGTYPE:CV.

/DEF

Syntax:

/DEF:filename

Passes a module-definition (.DEF) file to the linker. Only one .DEF file can be specified to LINK. For details about .DEF files, see [Module-Definition Files](#).

When a .DEF file is used in a build, whether the main output file is an executable file or a DLL, LINK creates an import library (.LIB) and an exports file (.EXP). These files are created regardless of whether the main output file contains exports.

Do not specify this option in the Microsoft Developer Studio environment; this option is for use only on the command line. To specify a .DEF file, add it to the project along with other files.

/DEFAULTLIB

Syntax:

/DEFAULTLIB:*libraries...*

This option adds one or more *libraries* to the list of libraries that LINK searches when resolving references. A library specified with /DEFAULTLIB is searched after libraries specified on the command line and before default libraries named in object files. To specify multiple libraries, type a comma (,) between library names.

Ignore All Default Libraries (**/NODEFAULTLIB**) overrides **/DEFAULTLIB:library**. Ignore Libraries (**/NODEFAULTLIB:library**) overrides **/DEFAULTLIB:library** when the same *library* name is specified in both.

/DLL

Syntax:

/DLL

This option builds a DLL as the main output file. A DLL usually contains exports that can be used by another program. There are three methods for specifying exports, listed in recommended order of use:

- cDEC\$ ATTRIBUTES DLLEXPORT in the source code
- An /EXPORT specification in a LINK command
- An EXPORTS statement in a module definition (.DEF) file

A program can use more than one method.

An alternate way to build a DLL is with the LIBRARY module-definition statement. The /BASE and /DLL options together are equivalent to the **LIBRARY** statement.

In Microsoft Developer Studio, you can set this option by choosing Dynamic-Link Library under Project Type in the New Project dialog box.

/ENTRY

Syntax:

/ENTRY:*function*

This option sets the starting address for an executable file or DLL. Specify a function name that is defined with cDEC\$ ATTRIBUTES STDCALL. The parameters and return value must be defined as documented in the Win32 API for **WinMain** (for an .EXE) or **DllEntryPoint** (for a DLL). It is recommended that you let the linker set the entry point.

By default, the starting address is a function name from the run-time library. The linker selects it according to the attributes of the program, as shown in the following table.

Function name	Default for
mainCRTStartup	An application using /SUBSYSTEM:CONSOLE; calls main

WinMainCRTStartup	An application using /SUBSYSTEM:WINDOWS; calls WinMain , which must be defined with the STDCALL attribute
_DllMainCRTStartup	A DLL; calls DllMain (which must be defined with the STDCALL attribute) if it exists

If the /DLL or /SUBSYSTEM option is not specified, the linker selects a subsystem and entry point depending on whether **main** or **WinMain** is defined.

The functions **main**, **WinMain**, and **DllMain** are the three forms of the user-defined entry point.

/EXPORT

Syntax:

/EXPORT:entryname[=*internalname*][, @*ordinal* [, NONAME]] [, DATA]

This option lets you export a function from your program to allow other programs to call the function. You can also export data. Exports are usually defined in a DLL.

The *entryname* is the name of the function or data item as it is to be used by the calling program. You can optionally specify the *internalname* as the function known in the defining program; by default, *internalname* is the same as *entryname*. The *ordinal* specifies an index into the exports table in the range 1 - 65535; if you do not specify *ordinal*, LINK assigns one. The NONAME keyword exports the function only as an ordinal, without an *entryname*.

The DATA keyword specifies that the exported item is a data item. The data item in the client program must be declared using DLLIMPORT. (The CONSTANT keyword is supported for compatibility but is not recommended.)

There are three methods for exporting a definition, listed in recommended order of use:

- cDEC\$ ATTRIBUTES DLLEXPORT in the source code
- An /EXPORT specification in a LINK command
- An EXPORTS statement in a module definition (.DEF) file

All three methods can be used in the same program. When LINK builds a program that contains exports, it also creates an import library, unless an .EXP file is used in the build.

LINK uses decorated forms of identifiers. A "decorated name" is 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.

The compiler decorates an identifier when it creates the object file. If *entryname* or *internalname* is specified to the linker in its undecorated form as it appears in the source code, LINK attempts to match the name. If it cannot find a unique match, LINK issues an error.

Use the DUMPBIN tool described in Examining Files with DUMPBIN to get the decorated form of an identifier when you need to specify it to the linker. Do not specify the decorated form of identifiers declared with the C or STDCALL attributes. For more information on when and how to

use decorated names, see [Adjusting Naming Conventions in Mixed-Language Programming](#).

/FIXED

Syntax:

/FIXED

This option tells the operating system to load the program only at its preferred base address. If the preferred base address is unavailable, the operating system does not load the file. For more information on base address, see [/BASE](#).

When you specify **/FIXED**, LINK does not generate a relocation section in the program. At run-time, if the operating system cannot load the program at that address, it issues an error and does not load the program.

Some Win32 operating systems, especially those that coexist with MS-DOS, frequently must relocate a program. A program created with **/FIXED** will not run on Win32s operating systems.

Note: Do not use **/FIXED** when building device drivers.

/FORCE

Syntax:

/FORCE:[{MULTIPLE|UNRESOLVED}]

This option tells the linker to create a valid executable file or DLL even if a symbol is referenced but not defined or is multiply defined.

The **/FORCE** option can take an optional argument:

- Use **/FORCE:MULTIPLE** to create an output file whether or not LINK finds more than one definition for a symbol.
- Use **/FORCE:UNRESOLVED** to create an output file whether or not LINK finds an undefined symbol.

A file created with this option may not run as expected. The linker will not link incrementally with the **/FORCE** option.

You can select this option in Microsoft Developer Studio by checking the Force File Output box in the Customize category of the Link tab in the Project Settings dialog box.

/HEAP

Syntax:

/HEAP:*reserve*,[*commit*]

Sets the size of the heap in bytes.

The *reserve* argument specifies the total heap allocation in virtual memory. The default heap size is 1MB. The linker rounds up the specified value to the nearest 4 bytes.

The optional *commit* argument is subject to interpretation by the operating system. In Windows NT, it specifies the amount of physical memory to allocate at a time. Committed virtual memory causes space to be reserved in the paging file. A higher *commit* value saves time when the application needs more heap space but increases the memory requirements and possibly startup time.

Specify the *reserve* and *commit* values in decimal or C-language notation. (Use the digits 1-9 for decimal values, precede octal values with zero (0), and precede hexadecimal values with zero and x (0x or 0X).

/IMPLIB

Syntax:

/IMPLIB:*filename*

This option overrides the default name for the import library that LINK creates when it builds a program that contains exports. The default name is formed from the base name of the main output file and the extension .LIB. A program contains exports if one or more of the following is true:

- cDEC\$ ATTRIBUTES DLLEXPORT in the source code
- An /EXPORT specification in a LINK command
- An EXPORTS statement in a module definition (.DEF) file

LINK ignores the /IMPLIB option when an import library is not being created. If no exports are specified, LINK does not create an import library. If an export (.EXP) file is used in the build, LINK assumes an import library already exists and does not create one. For information on import libraries and export files, see [Import Libraries and Exports Files](#) in Using Visual Fortran Tools.

/INCLUDE

Syntax:

/INCLUDE:*symbol*

This option tells the linker to add a specified symbol to the symbol table.

Specify a *symbol* name in the text box. To specify multiple symbols, specify **/INCLUDE:***symbol* once for each symbol.

The linker resolves *symbol* by adding the object that contains the symbol definition to the program. This is useful for including a library object that otherwise would not be linked to the program.

Specifying a symbol in the /INCLUDE option overrides the removal of that symbol by /OPT:REF.

To select this option in Microsoft Developer Studio, choose the Force Symbol References text box in the Input category of the Link tab of the Project Settings dialog box.

/INCREMENTAL

Syntax:

/INCREMENTAL:{YES|NO}

This option controls how the linker handles incremental linking.

By default, the linker runs in nonincremental mode. However, the default mode is incremental if you specify /DEBUG. To override a default incremental link, turn off Link Incrementally (or specify /INCREMENTAL:NO on the command line).

To link incrementally regardless of the default, turn on Link Incrementally (or specify /INCREMENTAL:YES on the command line). When you specify this option, the linker issues a warning if it cannot link incrementally and then links the program nonincrementally. Certain options and situations override /INCREMENTAL:YES.

Most programs can be linked incrementally. However, some changes are too great, and some options are incompatible with incremental linking. LINK performs a full link if any of the following options are specified:

- Link Incrementally is turned off (/INCREMENTAL:NO)
- COFF Format (/DEBUGTYPE:COFF)
- Both Formats (/DEBUGTYPE:BOTH)
- /OPT:REF
- /ORDER
- Use program Database is turned off (/PDB:NONE) when Generate Debug Info (/DEBUG) is specified

Additionally, LINK performs a full link if any of the following occur:

- Missing .ILK file. (LINK creates a new .ILK file in preparation for subsequent incremental linking.)
- No write permission for the .ILK file. (LINK ignores the .ILK file and links nonincrementally.)
- Missing .EXE or .DLL output file.
- Changing the timestamp of the .ILK, .EXE, or .DLL.
- Changing a LINK option. Most LINK options, when changed between builds, cause a full link.
- Adding or omitting an object file.

To select this option in Microsoft Developer Studio, select the Link Incrementally check box in the Customize category of the Link tab in the Project Settings dialog box.

/MAP

Syntax:

/MAP[:filename]

This option tells the linker to generate a mapfile. You can optionally specify a map file name to

override the default.

The linker names the mapfile with the base name of the program and the extension .MAP. To override the default name, use the *filename* argument.

A map file is a text file that contains the following information about the program being linked:

- The module name, which is the base name of the file
- The timestamp from the program file header (not from the file system)
- A list of groups in the program, with each group's start address (as *section:offset*), length, group name, and class
- A list of public symbols, with each address (as *section:offset*), symbol name, flat address, and object file where the symbol is defined
- The entry point (as *section:offset*)
- A list of fixups

To select this in Microsoft Developer Studio, select the Generate Mapfile check box in the Debug category of the Link tab in the Project Settings dialog box.

/NODEFAULTLIB

Syntax:

/NODEFAULTLIB[:*library*]

This option tells the linker to remove all default libraries from the list of libraries it searches when resolving external references. If you specify *library*, the linker only ignores the libraries you have named. To specify multiple *libraries*, type a comma (,) between the library names.

The linker resolves references to external definitions by searching first in libraries specified on the command line, then in default libraries specified with the /DEFAULTLIB option, then in default libraries named in object files.

Ignore All Default Libraries (/NODEFAULTLIB) overrides /DEFAULTLIB*library*. Ignore Libraries (/NODEFAULTLIB:*library*) overrides /DEFAULTLIB:*library* when the same *library* name is specified in both.

To select this in Microsoft Developer Studio, select the Ignore Libraries or Ignore All Default Libraries check box in the Input category of the Link tab in the Project Settings dialog box.

/NOENTRY

Syntax:

/NOENTRY

This option is required for creating a resource-only DLL.

Use this option to prevent LINK from linking a reference to `_main` into the DLL.

/NOLOGO

Syntax:

/NOLOGO

This option prevents display of the copyright message and version number. This option also suppresses echoing of command files.

By default, this information is sent by the linker to the Output window. On the command line, it is sent to standard output and can be redirected to a file.

To select this option in Microsoft Developer Studio, select the Suppress Startup Banner check box in the Customize category of the Link tab in the Project Settings dialog box.

/OPT

Syntax:

/OPT:{REF|NOREF}

This option controls the optimizations LINK performs during a build. Optimizations generally decrease the image size and increase the program speed, at a cost of increased link time.

By default, LINK removes unreferenced packaged functions (COMDATs). This optimization is called transitive COMDAT elimination. To override this default and keep unused packaged functions in the program, specify **/OPT:NOREF**. You can use the **/INCLUDE** option to override the removal of a specific symbol. It is not possible to create packaged functions with the Visual Fortran 5.0 compiler. This description is included for mixed-language applications with languages such as Visual C++ that support packaged functions (with the **/Gy** compiler option).

If you specify the **/DEBUG** option, the default for **/OPT** changes from REF to NOREF and all functions are preserved in the image. To override this default and optimize a debugging build, specify **/OPT:REF**. The **/OPT:REF** option disables incremental linking.

/ORDER

Syntax:

/ORDER:@filename

This option lets you perform optimization by telling LINK to place certain packaged functions into the image in a predetermined order. It is not possible to make packaged functions with the Visual Fortran 5.0 compiler. This description is included for mixed-language applications with languages such as Visual C++ that support packaged functions (with the **/Gy** compiler option).

LINK places packaged functions in the specified order within each section in the image.

Specify the order in *filename*, which is a text file that lists the packaged functions in the order you want to link them. Each line in *filename* contains the name of one packaged function. Function

names are case sensitive. A comment is specified by a semicolon (;) and can be on the same or a separate line. LINK ignores all text from the semicolon to the end of the line.

LINK uses decorated forms of identifiers. A *decorated name* is 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.

The compiler decorates an identifier when it creates the object file. If the name of the packaged function is specified to the linker in its undecorated form as it appears in the source code, LINK attempts to match the name. If it cannot find a unique match, LINK issues an error.

Use the DUMPBIN tool to get the decorated form of an identifier when you need to specify it to the linker. Do not specify the decorated form of identifiers declared with `_DEC$ ATTRIBUTES C` or `STDCALL`. For more information on when and how to use decorated names, see [Adjusting Naming Conventions](#) in Mixed-Language Programming.

If more than one `/ORDER` specification is used, the last one specified takes effect.

Ordering allows you to optimize your program's paging behavior through swap tuning. Group a function with the functions it calls. You can also group frequently called functions together. These techniques increase the probability that a called function is in memory when it is needed and will not have to be paged from disk.

This option disables incremental linking.

/OUT

Syntax:

/OUT:*filename*

This option overrides the default name and location of the image file that LINK creates. By default, LINK forms the filename using the base name of the first file specified and the appropriate extension (.EXE or .DLL).

The `/OUT` option controls the default base name for a mapfile or import library. For details, see the descriptions of [/MAP](#) and [/IMPLIB](#).

/PDB

Syntax:

/PDB**[:***filename***]**

This option controls how the linker produces debugging information. The optional *filename* argument overrides the default filename for the program database. The default filename for the PDB has the base name of the program and the extension .PDB.

By default when you specify `/DEBUG`, the linker creates a program database (PDB), which holds

debugging information. If you have not specified /DEBUG, the linker ignores /PDB.

If you specify /PDB:NONE, the linker does not create a PDB, but instead puts old-style debugging information into the executable file or DLL. The linker then calls the CVPACK.EXE tool, which must be in the same directory as LINK.EXE or in a directory in the PATH environment variable.

Debugging information in a program database must be in Microsoft Format (/DEBUGTYPE:CV). If you choose either COFF Format (/DEBUGTYPE:COFF) or Both Formats (/DEBUGTYPE:BOTH), no PDB is created.

Incremental linking is suppressed if you specify /PDB:NONE.

You can select this option in Microsoft Developer Studio by selecting the Use Program Database check box in the Customize category of the Link tab in the Project Settings dialog box.

/PROFILE

Syntax:

/PROFILE

This option creates an output file that can be used with the profiler. This option is found only in the General category on the Link tab.

A profiler-ready program has a map file. If it contains debugging information, the information must be stored in the output file instead of a program database file (.PDB file) and must be in Microsoft old-style format.

In Microsoft Developer Studio, setting Enable Profiling enables the Generate Mapfile option in the General and Debug categories. If you set the Generate Debug option, be sure to choose Microsoft Format in the Debug category.

On the command line, /PROFILE has the same effect as setting the /MAP option; if the /DEBUG option is specified, then /PROFILE also implies the options /DEBUGTYPE:CV and /PDB:NONE. In either case, /PROFILE implies /INCREMENTAL:NO.

You can select this option in Microsoft Developer Studio by selecting the Enable Profiling check box in the General category of the Link tab in the Project Settings dialog box.

/RELEASE

Syntax:

/RELEASE

This option sets the checksum in the header of an executable file.

The operating system requires the checksum for certain files such as device drivers. To ensure compatibility with future operating systems, set the checksum for release versions of your programs.

This option is set by default when you specify the /SUBSYSTEM:NATIVE option.

/STACK

Syntax:

/STACK:*reserve*[,*commit*]

This option sets the size of the stack in bytes.

The *reserve* argument specifies the total stack allocation in virtual memory. The default stack size is 1MB. The linker rounds up the specified value to the nearest 4 bytes.

The optional *commit* argument is subject to interpretation by the operating system. In Windows NT, it specifies the amount of physical memory to allocate at a time. Committed virtual memory causes space to be reserved in the paging file. A higher *commit* value saves time when the application needs more stack space but increases the memory requirements and possibly startup time.

Specify the *reserve* and *commit* values in decimal or C-language notation.

An alternate way to set the stack is with the STACKSIZE statement in a .DEF file. **STACKSIZE** overrides Stack Allocations (/STACK) if you specify both. You can change the stack after the executable file is built by using the EDITBIN.EXE tool. For more information, see Editing Files with EDITBIN.

To set these options in Microsoft Developer Studio, type values in the Reserve and Commit boxes in the Output category of the Link tab in the Project Settings dialog box.

/STUB

Syntax:

/STUB:*filename*

This option attaches an MS-DOS stub program to a Win32 program.

A stub program is invoked if the file is executed in MS-DOS. Usually, it displays an appropriate message; however, any valid MS-DOS application can be a stub program.

Specify a *filename* for the stub program after a colon (:). The linker checks *filename* to be sure that it is a valid MS-DOS executable file and issues an error if the file is not valid. The program must be an .EXE file; a .COM file is invalid for a stub program.

If you do not specify /STUB, the linker attaches a default stub program that generates the following message:

```
This program cannot be run in MS-DOS mode.
```

You can select this option in Microsoft Developer Studio by typing the stub file name in the MS-DOS Stub File Name box in the Input category of the Link tab of the Project Settings dialog box.

/SUBSYSTEM

Syntax:

/SUBSYSTEM:{CONSOLE|WINDOWS|NATIVE}[*major* [*.minor*]]

Tells the operating system how to run the executable file. The subsystem is specified as follows:

- The CONSOLE subsystem is used for Win32 character-mode applications. Console applications are given a console by the operating system. If main or wmain is defined, CONSOLE is the default.
- The WINDOWS subsystem is appropriate for an application that does not require a console. It creates its own windows for interaction with the user. If WinMain or wWinMain is defined, WINDOWS is the default.
- The NATIVE subsystem is used for device drivers.

The optional *major* and *minor* version numbers specify the minimum required version of the subsystem. The arguments are decimal numbers in the range 0 - 65535. The default is version 3.10 for CONSOLE and WINDOWS and 1.0 for NATIVE.

The choice of subsystem affects the default starting address for the program. For more information, see the [/ENTRY](#) option.

/VERBOSE

Syntax:

/VERBOSE[:LIB]

The linker sends information about the progress of the linking session to the Output window. If specified on the command line, the information is sent to standard output and can be redirected to a file.

The displayed information includes the library search process and lists each library and object name (with full path), the symbol being resolved from the library, and the list of objects that reference the symbol.

Adding :LIB to the /VERBOSE option restricts progress messages to those indicating the libraries searched.

You can select this option in Microsoft Developer Studio by filling in the Print Progress Messages box in the Customize category of the Link tab of the Project Settings dialog box.

/VERSION

Syntax:

/VERSION:*major* [*.minor*]

This option tells the linker to put a version number in the header of the executable file or DLL.

The *major* and *minor* arguments are decimal numbers in the range 0 - 65535. The default is version 0.0.

An alternate way to insert a version number is with the **VERSION** module-definition statement.

You can select this option in Microsoft Developer Studio by typing version information in the Major and Minor boxes in the Output category of the Link tab of the Project Settings dialog box.

/WARN

Syntax:

/WARN[:*level*]

Allows you to determine the output of LINK warnings. Specify the *level* as one of the following:

<i>level</i>	Meaning
0	Suppress all warnings.
1	Displays most warnings. Overrides a /WARN:<i>level</i> specified earlier on the LINK command line or in the LINK environment variable. Default if /WARN:<i>level</i> is not used.
2	Displays additional warnings. Default if /WARN is specified without <i>level</i> .

Module-Definition Files

A module-definition (.DEF) file is a text file that contains statements that define an executable file or DLL. (These should not be confused with module program units, described in Program Units and Procedures.) The following sections describe the statements in a .DEF file.

Because LINK provides equivalent command-line options for most module-definition statements, a typical program for Win32 does not usually require a .DEF file. In contrast, 16-bit programs for Windows almost *always* must be linked using a .DEF file.

You can use one or more of the following statements in a .DEF file:

- **DESCRIPTION**
- **EXPORTS**
- **LIBRARY**
- **NAME**
- **STACKSIZE**
- **VERSION**

The section describing each module-definition statement gives its command-line equivalent.

Rules for Module-Definition Statements

The following syntax rules apply to all statements in a .DEF file. Other rules that apply to specific statements are described with each statement.

- Statements and attribute keywords *are not* case sensitive. User-specified identifiers *are* case sensitive.

- Use one or more spaces, tabs, or newline characters to separate a statement keyword from its arguments and to separate statements from each other. A colon (:) or equal sign (=) that designates an argument is surrounded by zero or more spaces, tabs, or newline characters.
- A **NAME** or **LIBRARY** statement, if used, must precede all other statements.
- Most statements appear only once in the .DEF file and accept one specification of arguments. The arguments follow the statement keyword on the same or subsequent line(s). If the statement is repeated with different arguments later in the file, the latter statement overrides the former.
- The **EXPORTS** statement can appear more than once in the .DEF file. Each statement can take multiple specifications, which must be separated by one or more spaces, tabs, or newline characters. The statement keyword must appear once before the first specification and can be repeated before each additional specification.
- Comments in the .DEF file are designated by a semicolon (;) at the beginning of each comment line. A comment cannot share a line with a statement, but it can appear between specifications in a multiline statement. (**EXPORTS** is a multiline statement.)
- Numeric arguments are specified in decimal or in C-language notation.
- If a string argument matches a reserved word, it must be enclosed in double quotation (") marks.

Many statements have an equivalent LINK command-line option. See the [Linker options](#) for additional details.

DESCRIPTION

Syntax:

DESCRIPTION "*text*"

This statement writes a string into an .rdata section. Enclose the specified *text* in single or double quotation marks (' or "). To use a literal quotation mark (either single or double) in the string, enclose the string with the other type of mark.

This feature differs from the comment specified with the [/COMMENT](#) linker option.

EXPORTS

Syntax:

EXPORTS

This statement makes one or more definitions available as exports to other programs.

EXPORTS marks the beginning of a list of export *definitions*. Each definition must be on a separate line. The **EXPORTS** keyword can be on the same line as the first definition or on a preceding line. The .DEF file can contain one or more **EXPORTS** statements.

The syntax for an export definition is:

entryname[=*internalname*] [*@ordinal* [NONAME]] [DATA]

For information on the *entryname*, *internalname*, *ordinal*, NONAME, and DATA arguments, see the [/EXPORT](#) option.

There are three methods for exporting a definition, listed in recommended order of use:

- **ATTRIBUTES** DLLEXPORT in the source code
- An [/EXPORT](#) specification in a LINK command
- An **EXPORTS** statement in a .DEF file

All three methods can be used in the same program. When LINK builds a program that contains exports, it also creates an import library, unless the build uses an .EXP file.

LIBRARY

Syntax:

```
LIBRARY [library] [BASE=address]
```

This statement tells LINK to create a DLL. LINK creates an import library at the same time, unless you use an .EXP file in the build.

The *library* argument specifies the internal name of the DLL. (Use the [Output File Name \(/OUT\)](#) option to specify the DLL's output name.)

The BASE=*address* argument sets the base address that the operating system uses to load the DLL. This argument overrides the default DLL location of 0x10000000. See the description of the [Base Address \(/BASE\)](#) option for details about base addresses.

You can also use the [/DLL](#) linker option to specify a DLL build, and the [/BASE](#) option to set the base address.

NAME

Syntax:

```
NAME [application] [BASE=address]
```

This statement specifies a name for the main output file. An equivalent way to specify an output filename is with the [/OUT](#) option, and an equivalent way to set the base address is with the [/BASE](#) option. If both are specified, [/OUT](#) overrides **NAME**. See the [Base Address \(/BASE\)](#) and [Output File Name \(/OUT\)](#) options for details about output filenames and base addresses.

STACKSIZE

Syntax:

```
STACKSIZE reserve [,commit]
```

This statement sets the size of the stack in bytes. An equivalent way to set the stack is with the [/STACK](#) option. See the [/STACK](#) option for details about the *reserve* and *commit* arguments.

VERSION

Syntax:

VERSION *major* [*.minor*]

This statement tells LINK to put a number in the header of the executable file or DLL. The *major* and *minor* arguments are decimal numbers in the range 0 - 65535. The default is version 0.0.

An equivalent way to specify a version number is with the Version Information (/VERSION) option.

Linker Reserved Words

The following words are reserved by the linker. You can use these names as arguments in module-definition statements only if you enclose the name in double quotation marks ("").

APPLoader	INITINSTANCE	PRELOAD
BASE	IOPL	PROTMODE
CODE	LIBRARY	PURE
CONFORMING	LOADONCALL	READONLY
DATA	LONGNAMES	READWRITE
DESCRIPTION	MOVABLE	REALMODE
DEV386	MOVEABLE	RESIDENT
DISCARDABLE	MULTIPLE	RESIDENTNAME
DYNAMIC	NAME	SEGMENTS
EXECUTE-ONLY	NEWFILES	SHARED
EXECUTEONLY	NODATA	SINGLE
EXECUTEREAD	NOIOPL	STACKSIZE
EXETYPE	NONAME	STUB
EXPORTS	NONCONFORMING	VERSION
FIXED	NONDISCARDABLE	WINDOWAPI
FUNCTIONS	NONE	WINDOWCOMPAT
HEAPSIZE	NONSHARED	WINDOWS
IMPORTS	NOTWINDOWCOMPAT	
IMPURE	OBJECTS	
INCLUDE	OLD	

Microsoft Fortran Powerstation Command-Line Compatibility

This section provides compatibility information for FL32 command-line users of Microsoft Fortran Powerstation Version 4. It includes the following topics:

- [Using the DF or FL32 Command Line](#)
- [Equivalent Visual Fortran Compiler Options](#)

Using the DF or FL32 Command Line

You can use either the DF or FL32 commands to compile (and link) your application. The main difference between the DF and FL32 commands is the defaults set for certain command-line options:

- FL32 requests no optimization (/Od on x86 systems, /optimize:0 on Alpha systems). See [/\[no\]optimize](#).
- FL32 requests checking of arguments passed to and results from the math library (/math_library:check or /Od). Math library checking applies to x86 systems only. See [/\[no\]math_library](#).
- FL32 provides minimal debug information (/debug:minimal or /Zd). See [/\[no\]debug](#).
- FL32 requests full Microsoft® Fortran Powerstation compatibility (/fpscomp:all). See [/\[no\]fpscomp](#).
- FL32 disallows alternative PARAMETER syntax (/noaltparam). See [/\[no\]altparam](#).
- FL32 requests record length units for unformatted files to be in bytes (/assume:byterecl). See [/assume](#).
- FL32 requests warnings for mismatched arguments (/warn:argument_checking). See [/\[no\]warn](#).
- FL32 compiles each source unit individually and retains intermediate files that would otherwise be deleted (/keep). This prevents interprocedural optimizations at higher optimization levels. See [/keep](#).
- FL32 does not display an informational message related to compiling multiple files individually. See [/warn:fileopts](#).
- FL32 requests no inlining (/inline:none). See [/\[no\]inline](#).
- FL32 places module files in the same directory as the object files. See [/module:path](#).

The DF and FL32 commands both:

- Recognize the *same* set of command-line options. For example, the following commands are supported:

```
DF    /Odx test2.for
FL32 /Odx test2.for
```

Both DF and FL32 command lines allow most Microsoft Fortran Powerstation style options (such as /Ox) and all Visual Fortran options (such as /optimize:4). For a detailed list of equivalent Microsoft Fortran Powerstation style compiler options and Visual Fortran compiler options, see [Equivalent Visual Fortran Compiler Options](#).

- Activate the *same* compiler, the DIGITAL Fortran compiler.

For new programs and most existing applications, use the DIGITAL Fortran compiler (default). The DIGITAL Fortran compiler and language used by Visual Fortran provides a superset of the Fortran 90 standard with extensions for compatibility with previous versions of DIGITAL Fortran (DEC Fortran[™]), VAX FORTRAN[™], and Microsoft Fortran Powerstation Version 4.

- Pass options specified after /LINK to the LINK command.

The LINK command options after /link are passed directly to the Linker. These options are described in [Linker Options](#).

- Allow the use of indirect command files.

For example, assume the file text.txt contains the following:

```
/pdbfile:testout.pdb /exe:testout.exe /debug:full /optimize:0 test.f90 rest.f9
```

Either of the following (DF or FL32) commands executes the contents of file text.txt as an indirect command file to create a debugging version of the executable program and its associated PDB file:

```
DF @test.txt
FL32 @test.txt
```

To request Microsoft Fortran Powerstation V4 compatibility, specify the `/[no]fpscomp` option.

For information about using the DF command, see [Using the Compiler and Linker from the Command Line](#)".

Equivalent Visual Fortran Compiler Options

The following table lists the Microsoft Fortran Powerstation options and their Visual Fortran equivalents. The Microsoft Fortran Powerstation options (such as `/FAc`) are *case-sensitive*; other Visual Fortran options (such as `/asmfile`) are not case-sensitive/.

Fortran Powerstation Option (and Category)	Visual Fortran Command-Line Option
Listing Options	
<code>/FA</code>	Assembly listing. Specify <code>/noasmattributes</code> with <code>/asmfile[:file]</code> or <code>/FA</code> .
<code>/FAc</code>	Assembly listing with machine code. Specify <code>/asmattributes:machine</code> with <code>/asmfile[:file]</code> or <code>/FAc</code> .
<code>/FAs</code>	Assembly listing with source code. Specify <code>/asmattributes:source</code> with <code>/asmfile[:file]</code> or <code>/FAs</code> .
<code>/FAcs</code>	Assembly listing with machine instructions and source code. Specify <code>/asmattributes:all</code> with <code>/asmfile[:file]</code> or <code>/FAcs</code> .
<code>/Fa[file]</code>	Assembly listing to file <i>file</i> . Specify <code>/asmfile[:file]</code> with <code>/noasmattributes</code> or specify <code>/Fa[file]</code> .
<code>/Fc[file]</code>	Assembly listing with source and machine code to file <i>file</i> . Specify <code>/asmfile[:file]</code> with <code>/asmattributes:all</code> or specify <code>/Fc[file]</code> .
<code>/Fl[file]</code>	Assembly listing with machine instructions to file <i>file</i> . Specify <code>/asmfile[:file]</code> with <code>/asmattributes:machine</code> or specify <code>/Fl[file]</code> .
<code>/Fs[file]</code>	Source listing with compiled code. Specify <code>/list[:file]</code> with <code>/show:map</code> or specify <code>/Fs[file]</code> .
Code Generation Options	
<code>/FR[file]</code>	Generates extended Source Browser information. Specify <code>/browser[:file]</code> or <code>/FR[file]</code> .
<code>/G3 /G4 /G5</code>	Ignored on x86 systems. On Alpha systems, use <code>/tune:keyword</code> .
<code>/Ob2</code>	Automatic inlining of code, use with <code>/Ox</code> . Specify <code>/inline:speed</code> or <code>/Ob2</code> .

/Od	No code optimization (default for FL32 command). Specify <u>/optimize:0</u> with <u>/math_library:check</u> , or specify /Od.
/Op	Improved floating-point consistency. Specify <u>/fltconsistency</u> or /Op.
/Ox	Full optimization with no error checking. Specify <u>/optimize:4</u> with <u>/math_library:fast</u> and <u>/assume:nodummy_aliases</u> , or specify /Ox.
/Oxp	Speed optimization and denoted inlining; error checking. Specify <u>/optimize:4</u> with <u>/assume:nodummy_aliases</u> and <u>/math_library:check</u> with <u>/fltconsistency</u> (x86 systems), or specify /Oxp.
/Zp[n]	Packs structures on n-byte boundary (<i>n</i> is 1, 2, or 4). Specify <u>/alignment[:keyword]</u> or /Zp[n].
Language Extension Options	
/4Lnn	Line length for Fortran 90 fixed-form source (<i>nn</i> is 72, 80, or 132). Specify <u>/extend_source[:nn]</u> or /4Lnn.
/4Yb or /4Nb	Enable/disable extended error checking. Specify <u>/check[:keyword]</u> , /4Yb, or /4Nb.
/4Yd or /4Nd	Warnings about undeclared variables. Specify <u>/warn: [no]declarations</u> , /4Yd or /4Nd.
/W0	Suppress warnings. Specify <u>/nowarn</u> or /W0.
/W1	Show warnings (default). Specify <u>/warn:general</u> or /W1.
/WX	Interpret all warnings as errors. Specify <u>/warn:(general,errors)</u> or /WX.
Language Standard, Source Form, and Data Options	
/4Ya or /4Na	Makes all variables AUTOMATIC. Specify <u>/[no]automatic</u> , <u>/[no]static</u> , /4Ya, or /4Na.
/4Yaltparam /4Naltparam	Use the alternate syntax for PARAMETER statements. Specify <u>/[no]altparam</u> , /4Yaltparam, or /4Naltparam.
/4Yf or /4Nf	Use free-form source format. Specify <u>/[no]free</u> , <u>/[no]fixed</u> , /4Yf, or /4Nf.
/4I2	Change default KIND for INTEGER and LOGICAL declarations. Specify <u>/integer_size:nn</u> (<i>nn</i> is 16 for KIND=2) or /4I2.
/4R8	Change default KIND for REAL declarations. Specify <u>/real_size:nn</u> (<i>nn</i> is 32 for KIND=4) or /4R8.
/4Ys or /4Ns	Strict Fortran 90 syntax. Specify <u>/stand:f90</u> , <u>/warn:stderrs</u> , /4Ys, or /4Ns.
Compiler Directive Options	
/Dsymbol[=int]	Define preprocessor symbol. Specify <u>/define:symbol[=int]</u> or Dsymbol[=int]
/4ccstring	Treat lines with d or D in column 1 as comments. Specify <u>/d_lines</u> or /4ccd or /4ccD (partial support)
Build Control Options	
/4Yportlib or /4Nportlib	Specify /4Yportlib or /4Nportlib
/Fd[file]	Controls creation of compiler PDB files. Specify <u>/[no]pdbfile[:file]</u> or /Fd[file]
/Fe[file]	Specifies file name of executable or DLL file. Specify <u>/exe:file</u> , <u>/dll:file</u> , or /Fe[file]
/Fm[file]	Controls creation of link map file. Specify <u>/map[:file]</u> or /Fm[file]
/Fo[file]	Controls creation of object file. Specify <u>/object[:file]</u> or /Fo[file]
/GNa	Keep external names as is and treat source code identifiers as case

	sensitive. Specify <u>/names:as_is</u> or <u>/GNa</u>
<u>/GNI</u>	Make external names lowercase and ignore the case of source code identifiers. Specify <u>/names:lowercase</u> or <u>/GNI</u>
<u>/GNu</u>	Make external names uppercase and ignore the case of source code identifiers. Specify <u>/names:uppercase</u> or <u>/GNu</u>
<u>/Ipath</u>	Control search path for module or include files. Specify <u>/[no]include[:path]</u> or <u>/Ipath</u>
<u>/LD</u>	Create dynamic-link library. Specify <u>/dll</u> or <u>/LD</u>
<u>/MD</u>	Link against multithreaded DLL libraries. Specify <u>/libs:dll</u> with <u>/threads</u> or <u>/MD</u>
<u>/MDd</u>	Link against multithreaded DLL libraries. Specify <u>/libs:dll</u> with <u>/threads</u> and <u>/dbglibs</u> or specify <u>/MDd</u>
<u>/MDs</u>	Link against single threaded DLL libraries. Specify <u>/libs:dll</u> or <u>/MDs</u>
<u>/MG</u>	Link against libraries for windows applications. Specify <u>/winapp</u> or <u>/MG</u>
<u>/ML</u>	Link against single threaded static libraries. Specify <u>/libs:static</u> or <u>/ML</u>
<u>/MLd</u>	Link against single threaded static libraries. Specify <u>/libs:static</u> with <u>/dbglibs</u> or <u>/MLd</u>
<u>/MT</u>	Link against multithreaded static libraries. Specify <u>/libs:static</u> with <u>/threads</u> or <u>/MT</u>
<u>/MTd</u>	Link against multithreaded static libraries. Specify <u>/libs:static</u> with <u>/threads</u> and <u>/dbglibs</u> or specify <u>/MTd</u>
<u>/MW</u>	Link against quickwin multidoc libraries. Specify <u>/libs:qwin</u> or <u>/MW</u>
<u>/MWs</u>	Link against quickwin single doc libraries. Specify <u>/libs:qwins</u> or <u>/MWs</u>
<u>/Tffile</u>	Request that <i>file</i> be treated as a Fortran source file. Specify <u>/source:filename</u> or <u>/Tffile</u>
<u>/V"string"</u>	Place <i>string</i> in object file. Specify <u>/bintext:string</u> or <u>/V"string"</u>
<u>/Z7</u>	Request full debug information in object file. Specify <u>/debug:full</u> with <u>/nopdbfile</u> or <u>/Z7</u>
<u>/Zd</u>	Request minimal debug information. Specify <u>/debug:minimal</u> with <u>/pdbfile</u> or <u>/Zd</u>
<u>/Zi</u>	Request full debug information and create PDB file. Specify <u>/debug:full</u> with <u>/pdbfile</u> or <u>/Zi</u>
<u>/Zla</u>	Do not insert <i>any</i> library names in object file. Specify <u>/nolibdir</u> or <u>/Zla</u>
<u>/Zl</u>	Do not insert <i>default</i> library names in object file. Specify <u>/libdir:noautomatic</u> or <u>/Zl</u>
<u>/Zs</u>	Perform syntax check only (no object). Specify <u>/syntax_only</u> or <u>/Zs</u>
<u>/link [option]</u>	Begin specifying linker options. Specify <u>/link [option]</u>
Command-Line Specific Options	
<u>/?</u> , <u>/help</u>	Display command help. Specify <u>/?</u> or <u>/help</u>
<u>/nologo</u>	Prevent display of copyright information. Specify <u>/nologo</u>

Performance: Making Programs Run Faster

This chapter discusses the following topics related to improving run-time performance of DIGITAL Visual Fortran programs:

- Important software environment suggestions that apply to nearly all applications, including using the most recent version of the compiler, related performance tools, and efficient ways to compile using the DF command ([Software Environment and Efficient Compilation](#))
- Analyzing program performance, including using profiling tools ([Analyze Program Performance](#))
- Guidelines related to avoiding unaligned data ([Data Alignment Considerations](#))
- Guidelines for efficient array use ([Use Arrays Efficiently](#))
- Guidelines related to improving overall I/O performance ([Improve Overall I/O Performance](#))
- Additional performance guidelines related to source code ([Additional Source Code Guidelines for Run-Time Efficiency](#))
- Understanding the DF /optimize:num optimization level options and the types of optimizations performed ([Optimization Levels: the /optimize:num Option](#))
- Understanding other DF optimization options (besides the /optimize:num options ([Other Options Related to Optimization](#)))

Software Environment and Efficient Compilation

Before you attempt to analyze and improve program performance, you should:

- Obtain and install the latest version of Visual Fortran, along with performance products that can improve application performance.
- Use the DF command and its options in a manner that lets the DIGITAL Visual Fortran compiler perform as many optimizations as possible to improve run-time performance.
- Use certain performance capabilities provided by the operating system.

For more information, see:

- [Install the Latest Version of Visual Fortran and Performance Products](#)
- [Compile With Appropriate Options and Multiple Source Files](#)

Install the Latest Version of Visual Fortran and Performance Products

To ensure that your software development environment can significantly improve the run-time performance of your applications, obtain and install the following optional software products:

- The latest version of Visual Fortran

New releases of the DIGITAL Visual Fortran compiler and its associated run-time libraries may provide new features that improve run-time performance.

For information on more recent Visual Fortran releases, access the DIGITAL Fortran web page at the following URL: <http://www.digital.com/fortran>.

If you have the appropriate technical support contract, you can also contact the DIGITAL

technical support center for information on new releases (see [Visual Fortran Technical Support](#)).

- Performance profiling tools

The Developer Studio profiling tools allow function and line profiling. For more information on profiling, see [Analyze Program Performance](#).

- System-wide performance products

Other products are not specific to a particular programming language or application, but can improve system-wide performance, such as minimizing disk device I/O and handling capacity planning.

When running large programs, such as those accessing large arrays, adequate process limits and virtual memory space as well as proper system tuning are especially important.

Compile With Appropriate Options and Multiple Source Files

During the earlier stages of program development (such as for the debug configuration in Developer Studio), you can use compilation with minimal optimization. For example:

```
% DF /compile_only /optimize:1 sub2.f90
% DF /compile_only /optimize:1 sub3.f90
% DF /exe:main.exe /debug /optimize:0 main.f90 sub2.obj sub3.obj
```

During the later stages of program development (such as for the release configuration), you should:

- Specify multiple source files together and use an optimization level of at least `/optimize:4` on the DF command line to allow more interprocedural optimizations to occur. For instance, the following command compiles all three source files together using the default level of optimization (`/optimize:4`):

```
% DF /exe:main.exe main.f90 sub2.f90 sub3.f90
```

- Avoid using incremental linking.
- Consider building from the command line to allow multiple source files to be compiled together. For information on creating (exporting) makefile for command-line use, see [Files in a Project](#); for information about using NMAKE, see [Building Projects with NMAKE](#).

Compiling multiple source files lets the compiler examine more code for possible optimizations, which results in:

- Inlining more procedures
- More complete data flow analysis
- Reducing the number of external references to be resolved during linking

When compiling all source files together is not feasible (such as for very large programs), consider compiling related source files together using multiple DF commands rather than compiling source files individually.

If you use the `/compile_only` option to prevent linking, also use the `/object:file` option so that multiple sources files are compiled into a single object file, allowing more optimizations to occur.

Visual Fortran performs certain optimizations unless you specify the appropriate DF command-line options or corresponding Developer Studio options in the Optimization category of the Fortran tab (see [Categories of Compiler Options](#)). Additional optimizations can be enabled or disabled using DF command options or the Fortran tab in Developer Studio.

The following table shows DF options that can directly improve run-time performance on both *x86* and Alpha systems. Most of these options do not affect the accuracy of the results, while others improve run-time performance but can change some numeric results.

Options Related to Run-Time Performance

Option Names	Description	For More Information
<code>/align: keyword</code>	Controls whether padding bytes are added between data items within common blocks, derived-type data, and DIGITAL Fortran record structures to make the data items naturally aligned.	See Data Alignment Considerations .
<code>/fast</code>	Sets the following performance-related options: <code>/align:dcommons</code> , <code>/assume:noaccuracy_sensitive</code> , and <code>/math_library:fast</code> .	See description of each option
<code>/assume:noaccuracy_sensitive</code>	Allows the compiler to reorder code based on algebraic identities to improve performance, enabling certain optimizations. The numeric results can be slightly different from the default (<code>accuracy_sensitive</code>) because of the way intermediate results are rounded. This slight difference in numeric results is acceptable to most programs.	Arithmetic Reordering Optimizations
<code>/inline:all</code>	Inlines every call that can possibly be inlined while generating correct code. Certain recursive routines are not inlined to prevent infinite loops.	Controlling the Inlining of Procedures
<code>/inline:speed</code>	Inlines procedures that will improve run-time performance with a likely significant increase in program size.	Controlling the Inlining of Procedures
<code>/inline:size</code>	Inlines procedures that will improve run-time performance without a significant increase in program size. This type of inlining occurs with optimization level <code>/optimize:4</code> (or, on Alpha systems, <code>/optimize:4</code> or <code>/optimize:5</code>).	Controlling the Inlining of Procedures
<code>/math_library:fast</code>	On <i>x86</i> systems, requests that arguments to the math library routines are not checked to improve performance. On Alpha systems, requests the use of certain math library routines (used by intrinsic functions) that provide faster speed. Using this option may cause a slight loss of accuracy and provides less reliable arithmetic exception checking to get significant performance improvements in those functions.	/math_library
<code>/optimize:level</code>	Controls the optimization level and thus the types of optimizations performed. The default optimization level is <code>/optimize:4</code> , unless you specify <code>/debug</code> , which changes the	Optimization Levels: the /optimize Option

	default to /optimize:0 (no optimizations). On Alpha systems, use /optimize:5 to activate loop transformation optimizations and the software pipelining optimizations.	
/unroll: <i>num</i>	Specifies the number of times a loop is unrolled (<i>num</i>) when specified with optimization level /optimize:3 or higher. If you omit /unroll: <i>num</i> , the optimizer determines how many times loops are unrolled.	<u>Loop Unrolling</u>

The following table lists the DF options that can directly improve run-time performance on Alpha systems only.

Options Related to Run-Time Performance for Alpha Systems Only

Option Names	Description	For More Information
/architecture: <i>keyword</i>	Requests code generation for a specific Alpha chip generation. Certain Alpha chip generations use new instructions that provide improved performance for certain applications, but those instructions are not supported by older Alpha chip generations.	<u>/architecture</u>
/math_library:fast	On Alpha systems, requests the use of certain math library routines (used by intrinsic functions) that provide faster speed. Using this option may cause a slight loss of accuracy and provides less reliable arithmetic exception checking to get significant performance improvements in those functions.	<u>/math_library</u>
/optimize: <i>level</i>	Controls the optimization level and thus the types of optimization performed. The default optimization level is /optimize:4, unless you specify /debug, which changes the default to /optimize:0 (no optimizations). On Alpha systems, use /optimize:5 to activate loop transformation optimizations and the software pipelining optimization.	<u>Optimization Levels: the /optimize Option</u>
/pipeline	Activates the software pipelining optimization (a subset of /optimize:5).	<u>/[no]pipeline</u>
/transform_loops	Activates a group of loop transformation optimizations (a subset of /optimize:5).	<u>/[no]transform_loops</u>
/tune: <i>keyword</i>	Specifies the target processor generation (chip) architecture on which the program will be run, allowing the optimizer to make decisions about instruction tuning optimizations needed to create the most efficient code. Keywords allow specifying one particular Alpha processor generation type, multiple processor generation types, or the processor generation type currently in use during compilation. Regardless of the setting of /tune <i>keyword</i> , the generated code will run correctly on all implementations of the Alpha architecture.	<u>Requesting Optimized Code for a Specific Processor Generation</u>

The following table lists options that can slow program performance on x86 and Alpha systems.

Some applications that require floating-point exception handling might need to use a different `/fpe:n` option. Other applications might need to use the `/assume: dummy_aliases` or `/vms` options for compatibility reasons. Other options listed in the table are primarily for troubleshooting or debugging purposes.

Options that Slow Run-Time Performance

Option Names	Description	For More Information
<code>/assume:dummy_aliases</code>	Forces the compiler to assume that dummy (formal) arguments to procedures share memory locations with other dummy arguments or with variables shared through use association, host association, or common block use. These program semantics slow performance, so you should specify <code>/assume: dummy_aliases</code> only for the called subprograms that depend on such aliases. The use of dummy aliases violates the FORTRAN-77 and Fortran 90 standards but occurs in some older programs.	Dummy Aliasing Assumption
<code>/compile_only</code>	If you use <code>/compile_only</code> when compiling multiple source files, also specify <code>/object:file</code> to compile many source files together into one object file. Separate compilations prevent certain interprocedural optimizations, the same as using multiple DF commands or using <code>/compile_only</code> <i>without</i> the <code>/object:file</code> option.	Compile Using Multiple Source Files and Appropriate DF Options
<code>/check:bounds</code>	Generates extra code for array bounds checking at run time.	/check
<code>/check:overflow</code>	Generates extra code to check integer calculations for arithmetic overflow at run time. Once the program is debugged, you may want to omit this option to reduce executable program size and slightly improve run-time performance.	/check
<code>/fpe:n</code> values	On x86 systems, <code>/fpe:3</code> provides the best performance. Using other <code>/fpe</code> values slows program execution. On Alpha systems, using <code>/fpe:0</code> provides the best performance. Using other <code>/fpe</code> values (or using the <code>for_set_fpe</code> routine) to set equivalent exception handling slows program execution. For programs that specify <code>/fpe:3</code> , the impact on run-time	/fpe

	performance can be significant.	
<code>/rounding_mode:dynamic</code> (Alpha only)	Certain rounding modes and changing the rounding mode can slow program execution slightly.	<code>/rounding_mode</code> (Alpha only)
<code>/debug:full</code> , <code>/debug</code> , or equivalent	Generates extra symbol table information in the object file. Specifying <code>/debug</code> also reduces the default level of optimization to <code>/optimize:0</code> .	<code>/debug</code>
<code>/inline: none</code> <code>/inline: manual</code>	Prevents the inlining of all procedures (except statement functions).	Controlling the Inlining of Procedures
<code>/optimize:0</code> , <code>/optimize:1</code> , <code>/optimize:2</code> , or <code>/optimize:3</code>	Reduces the optimization level (and types of optimizations). Use during the early stages of program development or when you will use the debugger.	/[no]optimize and Optimization Levels: the /optimize Option
<code>/synchronous_exceptions</code> (Alpha only)	Generates extra code to associate an arithmetic exception with the instruction that causes it, slowing instruction execution. Use this option only when troubleshooting, such as when identifying the source of an exception.	<code>/[no]synchronous_exceptions</code>
<code>/vms</code>	Controls certain VMS-related run-time defaults, including alignment. If you specify the <code>/vms</code> option, you may need to also specify the <code>/align:records</code> option to obtain optimal run-time performance.	<code>/[no]vms</code>

For More Information:

- On compiling multiple files, see [Compiling and Linking for Optimization](#).

Analyze Program Performance

This section describes how you can analyze program performance using timings and profiling tools.

Before you analyze program performance, make sure any errors you might have encountered during the early stages of program development have been corrected. Only profile code that is stable and has been debugged.

The following topics are covered:

- [Timing Your Application](#)
- [Profiling and Performance Tools](#)

Timing Your Application

The following considerations apply to timing your application:

- Run program timings when other users are not active. Your timing results can be affected by one or more CPU-intensive processes also running while doing your timings.

- Try to run the program under the same conditions each time to provide the most accurate results, especially when comparing execution times of a previous version of the same program. Use the same system (processor model, amount of memory, version of the operating system, and so on) if possible.
- If you do need to change systems, you should measure the time using the same version of the program on both systems, so you know each system's effect on your timings.
- For programs that run for less than a few seconds, run several timings to ensure that the results are not misleading. Certain overhead functions like loading DLLs might influence short timings considerably.
- If your program displays a lot of text, consider redirecting the output from the program. Redirecting output from the program will change the times reported because of reduced screen I/O.

Methods of Timing Your Application

To perform application timings, use a version of the `TIME` command in a `.BAT` file (or the function timing profiling option). You might consider modifying the program to call routines within the program to measure execution time (possibly using conditionally compiled lines). For example:

- DIGITAL Fortran intrinsic procedures, such as `CPU TIME`, `SYSTEM CLOCK`, `DATE AND TIME`, and `TIME`.
- Library routines, such as `etime` or `time`.

Visual Fortran programs created in a Windows 95 development environment can be run and analyzed on Windows NT x86 systems. Whenever possible, perform detailed performance analysis on a system that closely resembles the system(s) that will be used for actual application use.

Sample Command Procedure that Uses TIME and Performance Monitor

The following example shows a `.BAT` command procedure that uses the `TIME` command and the Performance Monitor (`perfmon`) tool available on Windows NT systems. The `kill` command that stops the `perfmon` tool is included on the Windows NT Resource kit; if the `kill` tool is not available on your system, manually end the `perfmon` task (use the task manager).

This `.BAT` procedure assumes that the program to be timed is `myprog.exe`.

Before using this batch file, start the performance monitor to setup logging of the statistics that you are interested in:

1. At the DOS prompt type: `Perfmon`
2. In the View menu, select Log
3. In the Edit menu, select Add to Log and select some statistics
4. In the Options menu, select Log. In the dialog box:
 - Name the log file. This following `.BAT` procedure assumes that you have named the logfile `myprog.log`.
 - Consider adjusting the Log Interval.
 - As the last step, be sure to select "Start Log".
5. In the File menu, select Save Workspace to save the setup information. The following `.BAT` procedure assumes you have saved the workspace as `my_perfmon_setup.pmw`.

The command procedure follows:

```
echo off
rem Sample batch file to record performance statistics for later analysis.
rem This .bat file assumes that you have the utility "kill" available, which is
rem distributed with the NT resource kit.

rem Delete previous logs, then start up the Performance Monitor.
rem We use start so that control returns instantly to this batch file.
del myprog.log
start perfmon my_perfmon_setup.pmw

rem print the time we started
time <nul | findstr current

rem start the program we are interested in, this time using
rem cmd /c so that the batch file waits for the program to finish.
echo on
cmd /c myprog.exe
echo off

rem print the time we stopped
time <nul | findstr current

rem all done logging statistics
kill perfmon
rem if kill is not available, end the perfmon task manually
```

After the run, analyze your data by using Performance Monitor:

1. If it is not currently running, start Performance Monitor.
2. In the View menu, select Chart.
3. In the Options menu, select Data From and specify the name of the logfile.
4. In the Edit menu, select Add To Chart to display the counters.

For more information:

- About the optimizations that improve application performance without source code modification, see [Compile With Appropriate Options and Multiple Source Files](#).
- About profiling your application, see [Profiling and Performance Tools](#).

Profiling and Performance Tools

To generate profiling information, you use the compiler, linker, and the profiler from either Developer Studio or the command line.

Select those parts of your application that make the most sense to profile. For example, routines that perform user interaction may not be worth profiling. Consider profiling routines that perform a series of complex calculations or call multiple user-written subprograms.

Profiling identifies areas of code where significant program execution time is spent. It can also show areas of code that are not executed. Visual Fortran programs created in a Windows 95 or Windows NT x86 development environment can be run and analyzed on a Windows NT x86 or Windows 95 system. Whenever possible, perform detailed performance analysis on a system that closely resembles the system(s) that will be used to run the actual application.

For detailed information about profiling from the command line, see [Profiling Code from the Command Line](#).

There are two main types of profiling: **function profiling** and **line profiling**.

Function Profiling

Function profiling helps you locate areas of inefficient code. It can show:

- The time spent in functions and the number of times a function was called (function timing).
- Only the number of times a function was called (function counting).
- A list of functions executed or not executed (function coverage).
- Information about the stack when each function is called (function attribution).

Function profiling does not require debug information (it obtains addresses from a .MAP file). Since function profiling (except function attribution) uses the stack, routines that modify the stack cannot be profiled. Exclude object files for routines that modify the stack.

To perform function profiling:

1. In the Project menu, select Settings.
2. Click the Link tab.
3. In the General category, click the Enable profiling checkbox (this turns off incremental linking).
4. In the General category, click the Generate mapfile checkbox.
5. Click OK to accept the current project settings.
6. Build your application.
7. After building your application, profile your project.

Line Profiling

Line profiling collects more information than function profiling. It shows how many times a line is executed and whether certain lines are not executed. Line profiling requires debug information.

To perform line profiling:

1. In the Project menu, select Settings.
2. Click the Link tab.
3. In the General category, click the Enable profiling checkbox (this turns off incremental linking).
4. In the General category, click the Generate debug information checkbox.
5. Click on the Fortran tab.
6. In the category drop-down list, select Debug.
7. In the Debugging level drop-down list, select Full.
8. In the Debugging level drop-down list, click the Use Program Database for Debug Information checkbox.
9. Click OK to accept the current project settings.
10. Build your application
11. After building your application, profile your project.

Performance Tools

Tools that you can use to analyze performance include:

- Process Viewer (Pview) lets you view process and thread characteristics.
- Spy++ provides a graphical view of system use.
- On Windows NT systems, the Windows NT Performance Monitor can help identify performance bottlenecks.
- Other performance tools are available in the Microsoft® Win32 SDK (see the online *Platform SDK Tools Guide*, Tuning section in InfoViewer).

You can also purchase separate products to perform performance analysis and profiling.

Efficient Source Code

Once you have determined those sections of code where most of the program execution time is spent, examine these sections for coding efficiency. Suggested guidelines for improving source code efficiency are provided in the following sections:

- [Data Alignment Considerations](#)
- [Use Arrays Efficiently](#)
- [Improve Overall I/O Performance](#)
- [Additional Source Code Guidelines for Run-Time Efficiency](#)

For information about timing your application and for an example command procedure that uses the Windows NT Performance Monitor, see [Timing Your Application](#).

Data Alignment Considerations

For optimal performance, make sure your data is aligned naturally.

A natural boundary is a memory address that is a multiple of the data item's size (data type sizes are described in [Data Representation](#)). For example, a REAL (KIND=8) data item aligned on natural boundaries has an address that is a multiple of 8. An array is aligned on natural boundaries if all of its elements are so aligned.

All data items whose starting address is on a natural boundary are *naturally aligned*. Data not aligned on a natural boundary is called *unaligned data*.

Although the DIGITAL Fortran compiler naturally aligns individual data items when it can, certain DIGITAL Fortran statements (such as EQUIVALENCE) can cause data items to become unaligned (see [Causes of Unaligned Data and Ensuring Natural Alignment](#)).

Although you can use the DF command `/align: keyword` options to ensure naturally aligned data, you should check and consider reordering data declarations of data items within common blocks and structures. Within each common block, derived type, or record structure, carefully specify the order and sizes of data declarations to ensure naturally aligned data. Start with the largest size numeric items first, followed by smaller size numeric items, and then nonnumeric (character) data.

The following sections discuss data alignment considerations in more detail:

- [Causes of Unaligned Data and Ensuring Natural Alignment](#)
- [Checking for Inefficient Unaligned Data](#)
- [Ordering Data Declarations to Avoid Unaligned Data](#)
- [Options Controlling Alignment](#)

Causes of Unaligned Data and Ensuring Natural Alignment

Common blocks (**COMMON** statement), derived-type data, and DIGITAL Fortran [record structures](#) (**RECORD** statement) usually contain multiple items within the context of the larger structure.

The following declaration statements can force data to be unaligned:

- Common blocks (**COMMON** statement)

The order of variables in the **COMMON** statement determines their storage order.

Unless you are sure that the data items in the common block will be naturally aligned, specify either the `/align: commons` or `/align: dcommons` option, depending on the largest data size used.

For examples and more information, see [Arranging Data in Common Blocks](#).

- Derived-type (user-defined) data

Derived-type data members are declared after a **TYPE** statement.

If your data includes derived-type data structures and you specify the `/vms` option, you should also specify the `/align: records` option, unless you are sure that the data items in derived-type data structures will be naturally aligned.

If you omit the **SEQUENCE** statement, the `/align: records` option (default unless the `/vms` option is specified) ensures all data items are naturally aligned.

If you specify the **SEQUENCE** statement, the `/align: records` option is prevented from adding necessary padding to avoid unaligned data (data items are packed). When you use **SEQUENCE**, you should specify data declaration order such that all data items are naturally aligned.

For an example and more information, see [Arranging Data Items in Derived-Type Data](#).

- DIGITAL Fortran [record structures](#) (**RECORD** and **STRUCTURE** statements)

DIGITAL Fortran [record structures](#) usually contain multiple data items. The order of variables in the **STRUCTURE** statement determines their storage order. The **RECORD** statement names the [record structure](#). For examples and more information, see [Arranging Data Items in DIGITAL Fortran Record Structures](#).

If your data includes DIGITAL Fortran [record structures](#) and you specify the `/vms` option, you should also specify the `/align: records` option, unless you are sure that the data items in derived-type data and DIGITAL Fortran [record structures](#) will be naturally aligned.

For an example and more information, see [Arranging Data Items in DIGITAL Fortran Record Structures](#).

- **EQUIVALENCE** statements

EQUIVALENCE statements can force unaligned data or cause data to span natural boundaries.

To avoid unaligned data in a common block, derived-type data, or [record structures](#), use one or both of the following:

- For new programs or for programs where the source code declarations can be modified easily, plan the order of data declarations with care. For example, you should order variables in a **COMMON** statement such that numeric data is arranged from largest to smallest, followed by any character data (see the data declaration rules in [Ordering Data Declarations to Avoid Unaligned Data](#)).
- For existing programs where source code changes are not easily done or for array elements containing derived-type or [record structures](#), you can use command line options to request that the compiler align numeric data by adding padding spaces where needed.

Other possible causes of unaligned data include unaligned actual arguments and arrays that contain a derived-type structure or DIGITAL Fortran record structure.

When actual arguments from outside the program unit are not naturally aligned, unaligned data access will occur. DIGITAL Fortran assumes all passed arguments are naturally aligned and has no information at compile time about data that will be introduced by actual arguments during program execution.

For arrays where each array element contains a derived-type structure or DIGITAL Fortran [record structure](#), the size of the array elements may cause some elements (but not the first) to start on an unaligned boundary.

Even if the data items are naturally aligned within a derived-type structure without the **SEQUENCE** statement or a [record structures](#), the size of an array element might require use of `DF /align` options to supply needed padding to avoid some array elements being unaligned.

If you specify `/align: norecords` or specify `/vms` without `/align: records`, no padding bytes are added between array elements. If array elements each contain a derived-type structure with the **SEQUENCE** statement, array elements are packed without padding bytes regardless of the `DF` command options specified. In this case, some elements will be unaligned.

When `/align: records` option is in effect, the number of padding bytes added by the compiler for each array element is dependent on the size of the largest data item within the structure. The compiler determines the size of the array elements as an exact multiple of the largest data item in the derived-type structure without the **SEQUENCE** statement or a record structure. The compiler then adds the appropriate number of padding bytes.

For instance, if a structure contains an 8-byte floating-point number followed by a 3-byte character variable, each element contains five bytes of padding (16 is an exact multiple of 8). However, if the structure contains one 4-byte floating-point number, one 4-byte integer, followed by a 3-byte character variable, each element would contain one byte of padding (12 is an exact multiple of 4).

For More Information:

On the `/align:keyword` options, see [Options Controlling Alignment](#).

Checking for Inefficient Unaligned Data

During compilation, the DIGITAL Fortran compiler naturally aligns as much data as possible. Exceptions that can result in unaligned data are described in [Causes of Unaligned Data and Ensuring Natural Alignment](#).

Because unaligned data can slow run-time performance, it is worthwhile to:

- Double-check data declarations within common block, derived-type data, or **record structures** to ensure all data items are naturally aligned (see the data declaration rules in [Ordering Data Declarations to Avoid Unaligned Data](#)). Using modules to contain data declarations can ensure consistent alignment and use of such data.
- Avoid the **EQUIVALENCE** statement or use it in a manner that cannot cause unaligned data or data spanning natural boundaries.
- Ensure that passed arguments from outside the program unit are naturally aligned.
- Check that the size of array elements containing at least one derived-type data or **record structure** cause array elements to start on aligned boundaries (see [Causes of Unaligned Data and Ensuring Natural Alignment](#)).

During compilation, warning messages are issued for any data items that are known to be unaligned (unless you specify the `/warn:noalignments` option).

Ordering Data Declarations to Avoid Unaligned Data

For new programs or when the source declarations of an existing program can be easily modified, plan the order of your data declarations carefully to ensure the data items in a common block, derived-type data, record structure, or data items made equivalent by an **EQUIVALENCE** statement will be naturally aligned.

Use the following rules to prevent unaligned data:

- Always define the largest size numeric data items first.
- Add small data items of the correct size (or padding) before otherwise unaligned data to ensure natural alignment for the data that follows.
- If your data includes a mixture of character and numeric data, place the numeric data first.

Using the suggested data declaration guidelines minimizes the need to use the `/align keyword` options to add padding bytes to ensure naturally aligned data. In cases where the `/align keyword` options are still needed, using the suggested data declaration guidelines can minimize the number of padding bytes added by the compiler.

Arranging Data Items in Common Blocks

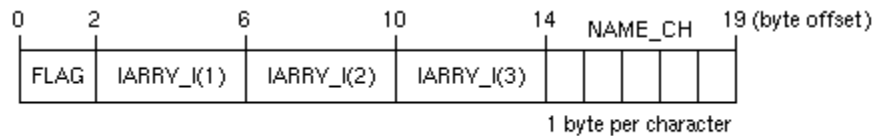
The order of data items in a **COMMON** statement determine the order in which the data items are

stored. Consider the following declaration of a common block named X:

```
LOGICAL (KIND=2) FLAG
INTEGER          IARRY_I(3)
CHARACTER(LEN=5) NAME_CH
COMMON /X/ FLAG, IARRY_I(3), NAME_CH
```

As shown in the following figure, if you omit the appropriate DF command options, the common block will contain unaligned data items beginning at the first array element of IARRY_I.

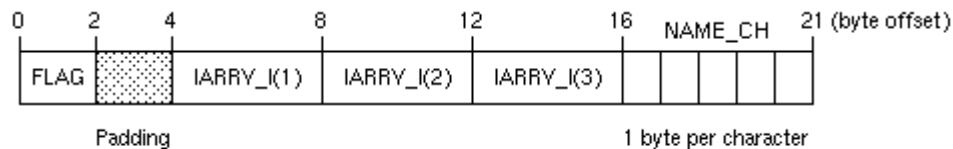
Common Block with Unaligned Data



ZK-6659A-GE

As shown in the following figure, if you compile the program units that use the common block with the /align:commons options, data items will be naturally aligned.

Common Block with Naturally Aligned Data



ZK-6660A-GE

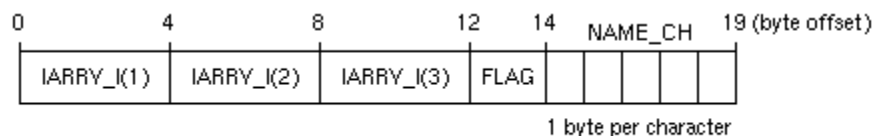
Because the common block X contains data items whose size is 32 bits or smaller, specify /align:commons. If the common block contains data items whose size might be larger than 32 bits (such as REAL (KIND=8) data), use /align:dcommons.

If you can easily modify the source files that use the common block data, define the numeric variables in the COMMON statement in descending order of size and place the character variable last to provide more portability and ensure natural alignment without padding or the DF command options /align:commons or /align:dcommons:

```
LOGICAL (KIND=2) FLAG
INTEGER          IARRY_I(3)
CHARACTER(LEN=5) NAME_CH
COMMON /X/ IARRY_I(3), FLAG, NAME_CH
```

As shown in the following figure, if you arrange the order of variables from largest to smallest size and place character data last, the data items will be naturally aligned.

Common Block with Naturally Aligned Reordered Data



ZK-7915A-GE

When modifying or creating all source files that use common block data, consider placing the common block data declarations in a module so the declarations are consistent. If the common block is not needed for compatibility (such as file storage or DIGITAL Fortran 77 use), you can place the data declarations in a module without using a common block.

Arranging Data Items in Derived-Type Data

Like common blocks, derived-type data may contain multiple data items (members).

Data item components within derived-type data will be naturally aligned on up to 64-bit boundaries, with certain exceptions related to the use of the **SEQUENCE** statement and DF options.

DIGITAL Fortran stores a derived data type as a linear sequence of values, as follows:

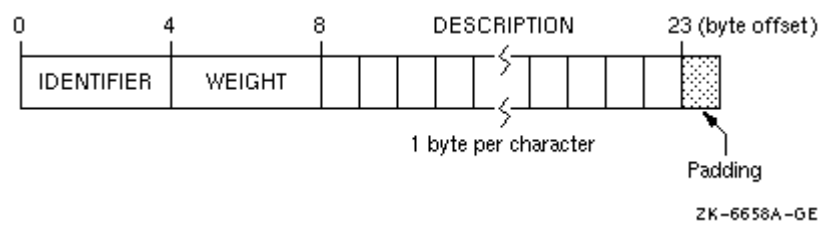
- If you specify the **SEQUENCE** statement, the first data item is in the first storage location and the last data item is in the last storage location. The data items appear in the order in which they are declared. The DF options have no effect on unaligned data, so data declarations must be carefully specified to naturally align data.
- If you omit the **SEQUENCE** statement, DIGITAL Fortran adds the padding bytes needed to naturally align data item components, unless you specify the /align:norecords option or the /vms option without /align:records.

Consider the following declaration of array CATALOG_SPRING of derived-type PART_DT:

```
MODULE DATA_DEFS
  TYPE PART_DT
    INTEGER      IDENTIFIER
    REAL         WEIGHT
    CHARACTER(LEN=15) DESCRIPTION
  END TYPE PART_DT
  TYPE (PART_DT) CATALOG_SPRING(30)
  .
  .
  .
END MODULE DATA_DEFS
```

As shown in the following figure, the largest numeric data items are defined first and the character data type is defined last. There are no padding characters between data items and all items are naturally aligned. The trailing padding byte is needed because CATALOG_SPRING is an array; it is inserted by the compiler when the /align records option is in effect.

Derived-Type Naturally Aligned Data (in CATALOG_SPRING())



Arranging Data Items in DIGITAL Fortran Record Structures

DIGITAL Fortran supports **record structures** provided by DIGITAL Fortran. DIGITAL Fortran **record structures** use the **RECORD** statement and optionally the **STRUCTURE** statement, which are extensions to the FORTRAN 77 and Fortran 90 standards. The order of data items in a **STRUCTURE** statement determine the order in which the data items are stored.

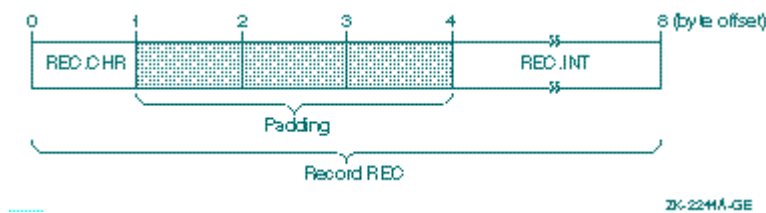
DIGITAL Fortran stores a **record** in memory as a linear sequence of values, with the record's first element in the first storage location and its last element in the last storage location. Unless you specify the /vms option without the /align:records option or specify /align:norecords, padding bytes are added if needed to ensure data fields are naturally aligned.

The following example contains a structure declaration, a **RECORD** statement, and diagrams of the resulting **records** as they are stored in memory:

```
STRUCTURE /STRA/
  CHARACTER*1 CHR
  INTEGER*4 INT
END STRUCTURE
.
.
.
RECORD /STRA/ REC
```

The following figure shows the memory diagram of **record REC** for naturally aligned **records**.

Memory Diagram of REC for Naturally Aligned Records



Options Controlling Alignment

The following options control whether the DIGITAL Fortran compiler adds padding (when needed) to naturally align multiple data items in common blocks, derived-type data, and DIGITAL Fortran **record structures**:

- The /align:commons option

Requests that data in common blocks be aligned on up to 4-byte boundaries, by adding padding bytes as needed. Unless you specify /fast, the default is /align:nocommons or arbitrary byte alignment of common block data. In this case, unaligned data can occur unless the order of data items specified in the COMMON statement places the largest numeric data item first, followed by the next largest numeric data (and so on), followed by any character data.

- The /align:dcommons option

Requests that data in common blocks be aligned on up to 8-byte boundaries, by adding padding bytes as needed. Unless you specify /fast, the default is /align:nocommons or arbitrary byte alignment of data items in a common data.

Specify the `/align:dcommons` option for applications that use common blocks, unless your application has no unaligned data or, if the application might have unaligned data, all data items are four bytes or smaller. For applications that use common blocks where all data items are four bytes or smaller, you can specify `/align:commons` instead of `/align:dcommons`.

- The `/align:norecords` option

Requests that multiple data items in derived-type data and **record structures** (a DIGITAL Fortran extension) be aligned arbitrarily on byte boundaries instead of being naturally aligned. If you omit the `/vms` option, the default is `/align:records`. If you specify the `/vms` option, `/align:norecords` is used (unless you also specify `/align: records`).

- The `/align:records` option

Requests that multiple data items in **record structures** and derived-type data without the `SEQUENCE` statement be naturally aligned, by adding padding bytes as needed. You only need to specify `/align records` if you specify the `/vms` option.

- The `/vms` option

Controls certain VMS-related run-time defaults, including alignment (sets `/align:norecords`) option. If you specify the `/vms` option, you may need to also specify the `/align:records` option to ensure that padding bytes are added.

The default behavior is that multiple data items in derived-type data and **record structures** *will* be naturally aligned; data items in common blocks *will not* be naturally aligned (`/align:records`) with `/align:nocommons`. In derived-type data, using the **SEQUENCE** statement prevents `/align:records` from adding needed padding bytes to naturally align data items.

Use Arrays Efficiently

On Alpha systems, many of the array access efficiency techniques described in this section are applied automatically by the DIGITAL Fortran loop transformation optimizations.

Several aspects of array use can improve run-time performance:

- The fastest array access occurs when contiguous access to the whole array or most of an array occurs. Perform one or a few array operations that access all of the array or major parts of an array rather than numerous operations on scattered array elements.

Rather than use explicit loops for array access, use elemental array operations, such as the following line that increments all elements of array variable A:

```
A = A + 1.
```

When reading or writing an array, use the array name and not a **DO** loop or an implied DO-loop that specifies each element number. Fortran 90 array syntax allows you to reference a whole array by using its name in an expression. For example:

```

REAL :: A(100,100)
A = 0.0
A = A + 1.          ! Increment all elements of A by 1
.
.
.

WRITE (8) A        ! Fast whole array use

```

Similarly, you can use derived-type array structure components, such as:

```

TYPE X
  INTEGER A(5)
END TYPE X
.
.
.
TYPE (X) Z
WRITE (8) Z%A      ! Fast array structure component use

```

- Make sure multidimensional arrays are referenced using proper array syntax and are traversed in the "natural" ascending order *column major* for Fortran. With column-major order, the leftmost subscript varies most rapidly with a stride of one. Writing a whole array uses column-major order.

Avoid *row-major* order, as is done by C, where the rightmost subscript varies most rapidly.

For example, consider the nested **DO** loops that access a two-dimension array with the J loop as the innermost loop:

```

INTEGER X(3,5), Y(3,5), I, J
Y = 0
DO I=1,3          ! I outer loop varies slowest
  DO J=1,5        ! J inner loop varies fastest
    X (I,J) = Y(I,J) + 1 ! Inefficient row-major storage order
  END DO          ! (rightmost subscript varies fastest)
END DO
.
.
.
END PROGRAM

```

Because J varies the fastest and is the second array subscript in the expression X (I,J), the array is accessed in row-major order.

To make the array accessed in natural column-major order, examine the array algorithm and data being modified.

Using arrays X and Y, the array can be accessed in natural column-major order by changing the nesting order of the **DO** loops so the innermost loop variable corresponds to the leftmost array dimension:

```

INTEGER X(3,5), Y(3,5), I, J
Y = 0
DO J=1,5          ! J outer loop varies slowest

```

```

      DO I=1,3                ! I inner loop varies fastest
        X (I,J) = Y(I,J) + 1 ! Efficient column-major storage order
      END DO                 ! (leftmost subscript varies fastest)
    END DO
  .
  .
  .
END PROGRAM

```

Fortran whole array access ($X = Y + I$) uses efficient column major order. However, if the application requires that J vary the fastest or if you cannot modify the loop order without changing the results, consider modifying the application program to use a rearranged order of array dimensions. Program modifications include rearranging the order of:

- Dimensions in the declaration of the arrays $X(5,3)$ and $Y(5,3)$
- The assignment of $X(J,I)$ and $Y(J,I)$ within the DO loops
- All other references to arrays X and Y

In this case, the original **DO** loop nesting is used where J is the innermost loop:

```

INTEGER  X(5,3), Y(5,3), I, J
Y = 0
DO I=1,3                ! I outer loop varies slowest
  DO J=1,5              ! J inner loop varies fastest
    X (J,I) = Y(J,I) + 1 ! Efficient column-major storage order
  END DO               ! (leftmost subscript varies fastest)
END DO
.
.
.
END PROGRAM

```

Code written to access multidimensional arrays in row-major order (like C) or random order can often make inefficient use of the CPU memory cache. For more information on using natural storage order during record I/O operations, see [Write Array Data in the Natural Storage Order](#).

- Use the available Fortran 90 array intrinsic procedures rather than create your own.

Whenever possible, use Fortran 90 array intrinsic procedures instead of creating your own routines to accomplish the same task. Fortran 90 array intrinsic procedures are designed for efficient use with the various Visual Fortran run-time components.

Using the standard-conforming array intrinsics can also make your program more portable.

- With multidimensional arrays where access to array elements will be noncontiguous, avoid left-most array dimensions that are a power of two (such as 256, 512). At higher levels of optimization (/optimize=3 or higher), the compiler pads certain power-of-two array sizes to minimize possible inefficient use of the cache.

Because the *cache sizes* are a power of two, *array dimensions* that are also a power of two may make inefficient use of cache when array access is noncontiguous. On Alpha systems, if the cache size is *an exact multiple* of the leftmost dimension, your program will probably make little use of the cache. This does not apply to contiguous sequential access or whole array access.

One work-around is to increase the dimension to allow some unused elements, making the leftmost dimension larger than actually needed. For example, increasing the leftmost dimension of A from 512 to 520 would make better use of cache:

```
REAL A (512,100)
DO I = 2,511
  DO J = 2,99
    A(I,J)=(A(I+1,J-1) + A(I-1, J+1)) * 0.5
  END DO
END DO
```

In this code, array A has a leftmost dimension of 512, a power of two. The innermost loop accesses the rightmost dimension (row major), causing inefficient access. Increasing the leftmost dimension of A to 520 (REAL A (520,100)) allows the loop to provide better performance, but at the expense of some unused elements.

Because loop index variables I and J are used in the calculation, changing the nesting order of the **DO** loops changes the results.

For More Information:

On arrays and their data declaration statements, see [Specifying Arrays](#).

Improve Overall I/O Performance

Improving overall I/O performance can minimize both device I/O and actual CPU time. The techniques listed in this section can greatly improve performance in many applications.

A *bottleneck* limits the maximum speed of execution by being the slowest process in an executing program. In some programs, I/O is the bottleneck that prevents an improvement in run-time performance. The key to relieving I/O bottlenecks is to reduce the actual amount of CPU and I/O device time involved in I/O. Bottlenecks may be caused by one or more of the following:

- A dramatic reduction in CPU time without a corresponding improvement in I/O time results in an I/O bottleneck.
- By such coding practices as:
 - Unnecessary formatting of data and other CPU-intensive processing
 - Unnecessary transfers of intermediate results
 - Inefficient transfers of small amounts of data
 - Application requirements

Improved coding practices can minimize actual device I/O, as well as the actual CPU time.

You can also consider solutions to system-wide problems like minimizing device I/O delays.

The following sections discuss I/O performance considerations in more detail:

- [Use Unformatted Files Instead of Formatted Files](#)
- [Write Whole Arrays or Strings](#)
- [Write Array Data in the Natural Storage Order](#)

- [Use Memory for Intermediate Results](#)
- [Enable Implied-DO Loop Collapsing](#)
- [Use of Variable Format Expressions](#)
- [Efficient Use of Record Buffers and Disk I/O](#)
- [Specify RECL](#)
- [Use the Optimal Record Type](#)

Use Unformatted Files Instead of Formatted Files

Use unformatted files whenever possible. Unformatted I/O of numeric data is more efficient and more precise than formatted I/O. Native unformatted data does not need to be modified when transferred and will take up less space on an external file.

Conversely, when writing data to formatted files, formatted data must be converted to character strings for output, less data can transfer in a single operation, and formatted data may lose precision if read back into binary form.

To write the array $A(25,25)$ in the following statements, S_1 is more efficient than S_2 :

```
S1          WRITE (7) A
S2          WRITE (7,100) A
            100  FORMAT (25(' ',25F5.21))
```

Although formatted data files are more easily ported to other systems, DIGITAL Fortran can convert unformatted data in several formats (see [Converting Unformatted Numeric Data](#)).

Write Whole Arrays or Strings

The general guidelines about array use discussed in [Use Arrays Efficiently](#) also apply to reading or writing an array with an I/O statement.

To eliminate unnecessary overhead, write whole arrays or strings at one time rather than individual elements at multiple times. Each item in an I/O list generates its own calling sequence. This processing overhead becomes most significant in implied-DO loops. When accessing whole arrays, use the array name (Fortran 90 array syntax) instead of using implied-DO loops.

Write Array Data in the Natural Storage Order

Use the *natural* ascending storage order whenever possible. This is column-major order, with the leftmost subscript varying fastest and striding by 1 (see [Use Arrays Efficiently](#)). If a program must read or write data in any other order, efficient block moves are inhibited.

If the whole array is not being written, natural storage order is the best order possible.

If you must use an *unnatural* storage order, in certain cases it might be more efficient to transfer the data to memory and reorder the data before performing the I/O operation.

Use Memory for Intermediate Results

Performance can improve by storing intermediate results in memory rather than storing them in a file on a peripheral device. One situation that may not benefit from using intermediate storage is when there is a disproportionately large amount of data in relation to physical memory on your system. Excessive page faults can dramatically impede virtual memory performance.

Enable Implied-DO Loop Collapsing

DO loop collapsing reduces a major overhead in I/O processing. Normally, each element in an I/O list generates a separate call to the DIGITAL Fortran RTL. The processing overhead of these calls can be most significant in implied-DO loops.

DIGITAL Fortran reduces the number of calls in implied-DO loops by replacing up to seven nested implied-DO loops with a single call to an optimized run-time library I/O routine. The routine can transmit many I/O elements at once.

Loop collapsing can occur in formatted and unformatted I/O, but only if certain conditions are met:

- The control variable must be an integer. The control variable cannot be a dummy argument or contained in an **EQUIVALENCE** or **VOLATILE** statement. DIGITAL Fortran must be able to determine that the control variable does not change unexpectedly at run time.
- The format must not contain a variable format expression.

For More Information:

- See the **VOLATILE** attribute and statement.
- On loop optimizations, see Optimization Levels: the /optimize:num Option.

Use of Variable Format Expressions

Variable format expressions (a DIGITAL Fortran extension) are almost as flexible as run-time formatting, but they are more efficient because the compiler can eliminate run-time parsing of the I/O format. Only a small amount of processing and the actual data transfer are required during run time.

On the other hand, run-time formatting can impair performance significantly. For example, in the following statements, S₁ is more efficient than S₂ because the formatting is done once at compile time, not at run time:

```

S1      WRITE (6,400) (A(I), I=1,N)
        400  FORMAT (1X, <N> F5.2)
           .
           .
S2      WRITE (CHFMT,500) '(1X, ',N, 'F5.2)'
        500  FORMAT (A,I3,A)
           WRITE (6,FMT=CHFMT) (A(I), I=1,N)

```

Efficient Use of Record Buffers and Disk I/O

Records being read or written are transferred between the user's program buffers and one or more disk block I/O buffers, which are established when the file is opened by the DIGITAL Fortran RTL. Unless very large records are being read or written, multiple logical records can reside in the disk block I/O buffer when it is written to disk or read from disk, minimizing physical disk I/O.

You can specify the size of the disk block I/O buffer by using the **OPEN** statement **BLOCKSIZE** specifier. If you omit the **BLOCKSIZE** specifier in the **OPEN** statement, it is set for optimal I/O use with the type of device the file resides on.

The default for **BUFFERCOUNT** is 1. Any experiments to improve I/O performance should increase the **BUFFERCOUNT** value and not the **BLOCKSIZE** value, to increase the amount of data read by each disk I/O.

Specify RECL

The sum of the record length (**RECL** specifier in an **OPEN** statement) and its overhead is a multiple or divisor of the blocksize, which is device specific. For example, if the **BLOCKSIZE** is 8192 then **RECL** might be 24576 (a multiple of 3) or 1024 (a divisor of 8).

The **RECL** value should fill blocks as close to capacity as possible (but not over capacity). Such values allow efficient moves, with each operation moving as much data as possible; the least amount of space in the block is wasted. Avoid using values larger than the block capacity, because they create very inefficient moves for the excess data only slightly filling a block (allocating extra memory for the buffer and writing partial blocks are inefficient).

The **RECL** value unit for formatted files is always 1-byte units. For unformatted files, the **RECL** unit is 4-byte units, unless you specify the `/assume:byterecl` option to request 1-byte units.

When porting unformatted data files from non-DIGITAL systems, see [Converting Unformatted Numeric Data](#).

Use the Optimal Record Type

Unless a certain record type is needed for portability reasons, choose the most efficient type, as follows:

- For sequential files of a consistent record size, the fixed-length record type gives the best performance.
- For sequential unformatted files when records are not fixed in size, the variable-length record type gives the best performance, particularly for **BACKSPACE** operations.
- For sequential formatted files when records are not fixed in size, the `Stream_LF` record type gives the best performance.

For More Information:

- On **OPEN** statement specifiers and defaults, see [I/O Statements](#) and [OPEN](#).
- On Visual Fortran data files, see [Devices and Files](#).

Additional Source Code Guidelines for Run-Time Efficiency

In addition to data alignment and the efficient use of arrays and I/O, other source coding guidelines can be implemented to improve run-time performance.

The amount of improvement in run-time performance is related to the number of times a statement is executed. For example, improving an arithmetic expression executed within a loop many times has the potential to improve performance, more than improving a similar expression executed once outside a loop.

Suggested guidelines for improving source code efficiency are provided in the following sections:

- [Avoid Small Integer and Small Logical Data Items](#)
- [Avoid Mixed Data Type Arithmetic Expressions](#)
- [Use Efficient Data Types](#)
- [Avoid Using Slow Arithmetic Operators](#)
- [Avoid EQUIVALENCE Statement Use](#)
- [Use Statement Functions and Internal Subprograms](#)
- [Code DO Loops for Efficiency](#)

Avoid Small Integer and Small Logical Data Items

To minimize data storage and memory cache misses with arrays, use 32-bit data rather than 64-bit data, unless you require the greater range and precision of double precision floating-point numbers or, on Alpha systems, the numeric range of 8-byte integers.

On Alpha systems, avoid using integer or logical data less than 32 bits (KIND=4). Accessing a 16-bit (KIND=2) or 8-bit (KIND=1) data type can result in a sequence of machine instructions to access the data, rather than a single, efficient machine instruction for a 32-bit data item.

Avoid Mixed Data Type Arithmetic Expressions

Avoid mixing integer and floating-point (REAL) data in the same computation. Expressing all numbers in a floating-point arithmetic expression (assignment statement) as floating-point values eliminates the need to convert data between fixed and floating-point formats. Expressing all numbers in an integer arithmetic expression as integer values also achieves this. This improves run-time performance.

For example, assuming that I and J are both INTEGER variables, expressing a constant number (2.) as an integer value (2) eliminates the need to convert the data:

Original Code: INTEGER I, J

I= J / 2.

Efficient Code: INTEGER I, J

I= J / 2

For applications with numerous floating-point operations, consider using the /assume: accuracy_sensitive option (see [Arithmetic Reordering Optimizations](#)) if a small difference in the result is acceptable.

You can use different *sizes* of the same general data type in an expression with minimal or no effect

on run-time performance. For example, using REAL, DOUBLE PRECISION, and COMPLEX floating-point numbers in the same floating-point arithmetic expression has minimal or no effect on run-time performance.

Use Efficient Data Types

In cases where more than one data type can be used for a variable, consider selecting the data types based on the following hierarchy, listed from most to least efficient:

- Integer (for Alpha systems, also see [Avoid Small Integer and Small Logical Data Items](#))
- Single-precision real, expressed explicitly as REAL, REAL (KIND=4), or REAL*4
- Double-precision real, expressed explicitly as DOUBLE PRECISION, REAL (KIND=8), or REAL*8

However, keep in mind that in an arithmetic expression, you should avoid mixing integer and floating-point (REAL) data (see [Avoid Mixed Data Type Arithmetic Expressions](#)).

Avoid Using Slow Arithmetic Operators

Before you modify source code to avoid slow arithmetic operators, be aware that optimizations convert many slow arithmetic operators to faster arithmetic operators. For example, the compiler optimizes the expression $H=J**2$ to be $H=J*J$.

Consider also whether replacing a slow arithmetic operator with a faster arithmetic operator will change the accuracy of the results or impact the maintainability (readability) of the source code.

Replacing slow arithmetic operators with faster ones should be reserved for critical code areas. The following hierarchy lists the DIGITAL Fortran arithmetic operators, from fastest to slowest:

- Addition (+), subtraction (-), and floating-point multiplication (*)
- Integer multiplication (*)
- Division (/)
- Exponentiation (**)

Avoid EQUIVALENCE Statement Use

Avoid using **EQUIVALENCE** statements. **EQUIVALENCE** statements can:

- Force unaligned data or cause data to span natural boundaries.
- Prevent certain optimizations, including:
 - Global data analysis under certain conditions (see [Global Optimizations](#))
 - Implied-DO loop collapsing when the control variable is contained in an **EQUIVALENCE** statement

Use Statement Functions and Internal Subprograms

Whenever the DIGITAL Visual Fortran compiler has access to the use and definition of a subprogram during compilation, it may choose to inline the subprogram. Using statement functions and internal subprograms maximizes the number of subprogram references that will be inlined,

especially when multiple source files are compiled together at optimization level `/optimize:4` (or, on Alpha systems, optimization level `/optimize:4` or `/optimize:5`).

For more information, see [Compile With Appropriate Options and Multiple Source Files](#).

Code DO Loops for Efficiency

Minimize the arithmetic operations and other operations in a **DO** loop whenever possible. Moving unnecessary operations outside the loop will improve performance (for example, when the intermediate nonvarying values within the loop are not needed).

For More Information:

- On loop optimizations, see [Software Pipelining](#) (Alpha systems only) and [Controlling Loop Unrolling](#).
- On the **DO** statement, see [DO](#) and your language reference manual.

Optimization Levels: the `/optimize` Option

Visual Fortran performs many optimizations by default. You do not have to recode your program to use them. However, understanding how optimizations work helps you remove any inhibitors to their successful function.

If an operation can be performed, eliminated, or simplified at compile time, Visual Fortran does so, rather than have it done at run time. The time required to compile the program usually increases as more optimizations occur.

The program will likely execute faster when compiled at `/optimize:4`, but will require more compilation time than if you compile the program at a lower level of optimization.

The size of object files varies with the optimizations requested. Factors that can increase object file size include an increase of loop unrolling or procedure inlining.

The following table lists the levels of DIGITAL Fortran optimization with different `/optimize:num` options (for example, `/optimize:0` specifies no selectable optimizations); some optimizations always occur. On x86 systems, `/optimize:4` specifies all levels of optimizations. On Alpha systems, `/optimize:5` specifies all levels of optimizations, including loop transformation and software pipelining.

Levels of Optimization with Different `/optimize:num` Options

Optimization Type	Option					
	<code>/optimize:0</code>	<code>/optimize:1</code>	<code>/optimize:2</code>	<code>/optimize:3</code>	<code>/optimize:4</code>	<code>/optimize:5</code>
Loop transformation and software pipelining						x (Alpha only)
Automatic inlining					x	x
Additional global optimizations			x	x	x	x
Global optimizations		x	x	x	x	x

Local (minimal) optimizations		x	x	x	x	x
-------------------------------	--	---	---	---	---	---

The default is `/optimize:4`. However, if `/debug` is also specified, the default is `/optimize:0` (no optimizations).

In the table, the following terms are used to describe the levels of optimization (described in detail in the following sections):

- *Local (minimal) optimizations* (`/optimize:1`) or higher occur within the source program unit and include recognition of common subexpressions and the expansion of multiplication and division.
- *Global optimizations* (`/optimize:2`) or higher include such optimizations as data-flow analysis, code motion, strength reduction, split-lifetime analysis, and instruction scheduling.
- *Additional global optimizations* (`/optimize:3`) or higher improve speed at the cost of extra code size. These optimizations include loop unrolling, code replication to eliminate branches, and padding certain power-of-two array sizes for more efficient cache use.
- *Automatic inlining* (`/optimize:4`) or higher applies interprocedural analysis and inline expansion of small procedures, usually by using heuristics that limit extra code.
- On Alpha systems, *Loop transformation and Software pipelining* (`/optimize:5`), include a group of loop transformation optimizations and the software pipelining optimization. The loop transformation optimizations apply to array references within loops and can apply to multiple nested loops. Loop transformation optimizations can improve the performance of the memory system.

Software pipelining applies instruction scheduling to certain innermost loops, allowing instructions within a loop to "wrap around" and execute in a different iteration of the loop. This can reduce the impact of long-latency operations, resulting in faster loop execution. Software pipelining also enables the prefetching of data to reduce the impact of cache misses.

The following sections discuss I/O performance considerations in more detail:

- [Optimizations Performed at All Optimization Levels](#)
- [Local \(Minimal\) Optimizations](#)
- [Global Optimizations](#)
- [Additional Global Optimizations](#)
- [Automatic Inlining](#)
- [Loop Transformation and Software Pipelining \(Alpha only\)](#)

Optimizations Performed at All Optimization Levels

The following optimizations occur at any optimization level (`/optimize:0` through `/optimize:5`):

- Space optimizations

Space optimizations decrease the size of the object or executing program by eliminating unnecessary use of memory, thereby improving speed of execution and system throughput. Visual Fortran space optimizations are as follows:

- Constant Pooling

Only one copy of a given constant value is ever allocated memory space. If that constant value is used in several places in the program, all references point to that value.

- Dead Code Elimination

If operations will never execute or if data items will never be used, Visual Fortran eliminates them. Dead code includes unreachable code and code that becomes unused as a result of other optimizations, such as value propagation.

- Inlining arithmetic statement functions and intrinsic procedures

Regardless of the optimization level, Visual Fortran inserts arithmetic statement functions directly into a program instead of calling them as functions. This permits other optimizations of the inlined code and eliminates several operations, such as calls and returns or stores and fetches of the actual arguments. For example:

```
SUM(A,B) = A+B
      .
      .
      .
Y = 3.14
X = SUM(Y,3.0)    ! With value propagation, becomes: X = 6.14
```

Many intrinsic procedures are automatically inlined.

Inlining of other subprograms, such as contained subprograms, occurs at optimization level /optimize:4 (or /optimize:5 on Alpha systems).

- Implied-DO loop collapsing

DO loop collapsing reduces a major overhead in I/O processing. Normally, each element in an I/O list generates a separate call to the Visual Fortran RTL. The processing overhead of these calls can be most significant in implied-DO loops.

If Visual Fortran can determine that the format will not change during program execution, it replaces the series of calls in up to seven nested implied-DO loops with a single call to an optimized RTL routine (see [Enable Implied-Do Loop Collapsing](#)). The optimized RTL routine can transfer many elements in one operation.

Visual Fortran collapses implied-DO loops in formatted and unformatted I/O operations, but it is more important with unformatted I/O, where the cost of transmitting the elements is a higher fraction of the total cost.

- Array temporary elimination and **FORALL** statements

Certain array store operations are optimized. For example, to minimize the creation of array temporaries, Visual Fortran can detect when no overlap occurs between the two sides of an array assignment. This type of optimization occurs for some assignment statements in **FORALL** constructs.

Certain array operations are also candidates for loop unrolling optimizations (see [Loop Unrolling](#)).

Local (Minimal) Optimizations

To enable local optimizations, use /optimize:1 or a higher optimization level /optimize:2, /optimize:3, /optimize:4, or (on Alpha systems) /optimize:5.

To prevent local optimizations, specify the /optimize:0 option.

The following sections discuss the local optimizations:

- [Common Subexpression Elimination](#)
- [Integer Multiplication and Division Expansion](#)
- [Compile-Time Operations](#)
- [Value Propagation](#)
- [Dead Store Elimination](#)
- [Register Usage](#)
- [Mixed Real/Complex Operations](#)

Common Subexpression Elimination

If the same subexpressions appear in more than one computation and the values do not change between computations, Visual Fortran computes the result once and replaces the subexpressions with the result itself:

```
DIMENSION A(25,25), B(25,25)
A(I,J) = B(I,J)
```

Without optimization, these statements can be coded as follows:

```
t1 = ((J-1)*25+(I-1))*4
t2 = ((J-1)*25+(I-1))*4
A(t1) = B(t2)
```

Variables t1 and t2 represent equivalent expressions. Visual Fortran eliminates this redundancy by producing the following:

```
t = ((J-1)*25+(I-1))*4
A(t) = B(t)
```

Integer Multiplication and Division Expansion

Expansion of multiplication and division refers to bit shifts that allow faster multiplication and division while producing the same result. For example, the integer expression (I*17) can be calculated as I with a 4-bit shift plus the original value of I. This can be expressed using the DIGITAL Fortran **ISHFT** intrinsic function:

```
J1 = I*17
J2 = ISHFT(I,4) + I      ! equivalent expression for I*17
```

The optimizer uses machine code that, like the **ISHFT** intrinsic function, shifts bits to expand

multiplication and division by literals.

Compile-Time Operations

Visual Fortran does as many operations as possible at compile time rather than having them done at run time.

Constant Operations

Visual Fortran can perform many operations on constants (including **PARAMETER** constants):

- Constants preceded by a unary minus sign are negated.
- Expressions involving +, -, *, or / operators are evaluated; for example:

```
PARAMETER (NN=27)
I = 2*NN+J           ! Becomes: I = 54 + J
```

Evaluation of some constant functions and operators is performed at compile time. This includes certain functions of constants, concatenation of string constants, and logical and relational operations involving constants.

- Lower-ranked constants are converted to the data type of the higher-ranked operand:

```
REAL X, Y
X = 10 * Y           ! Becomes: X = 10.0 * Y
```

- Array address calculations involving constant subscripts are simplified at compile time whenever possible:

```
INTEGER I(10,10)
I(1,2) = I(4,5)     ! Compiled as a direct load and store
```

Algebraic Reassociation Optimizations

Visual Fortran delays operations to see whether they have no effect or can be transformed to have no effect. If they have no effect, these operations are removed. A typical example involves unary minus and .NOT. operations:

```
X = -Y * -Z         ! Becomes: Y * Z
```

Value Propagation

Visual Fortran tracks the values assigned to variables and constants, including those from **DATA** statements, and traces them to every place they are used. Visual Fortran uses the value itself when it is more efficient to do so.

When compiling subprograms, Visual Fortran analyzes the program to ensure that propagation is safe if the subroutine is called more than once.

Value propagation frequently leads to more value propagation. Visual Fortran can eliminate run-time

operations, comparisons and branches, and whole statements.

In the following example, constants are propagated, eliminating multiple operations from run time:

Original Code	Optimized Code
<i>PI</i> = 3.14	.
.	.
.	<i>PIOVER2</i> = 1.57
<i>PIOVER2</i> = <i>PI</i> /2	.
.	.
.	<i>I</i> = 100
<i>I</i> = 100	.
.	.
.	10 <i>A</i> (100) = 3.0* <i>Q</i>
.	.
<i>IF</i> (<i>I</i> .GT.1) <i>GOTO</i> 10	.
10 <i>A</i> (<i>I</i>) = 3.0* <i>Q</i>	.

Dead Store Elimination

If a variable is assigned but never used, Visual Fortran eliminates the entire assignment statement:

```
X = Y*Z
.
.
.      !If X is not used in between, X=Y*Z is eliminated.
X = A(I,J)* PI
```

Some programs used for performance analysis often contain such unnecessary operations. When you try to measure the performance of such programs compiled with Visual Fortran, these programs may show unrealistically good performance results. Realistic results are possible only with program units using their results in output statements.

Register Usage

A large program usually has more data that would benefit from being held in registers than there are registers to hold the data. In such cases, Visual Fortran typically tries to use the registers according to the following descending priority list:

1. For temporary operation results, including array indexes
2. For variables
3. For addresses of arrays (base address)
4. All other usages

Visual Fortran uses heuristic algorithms and a modest amount of computation to attempt to determine an effective usage for the registers.

Holding Variables in Registers

Because operations using registers are much faster than using memory, Visual Fortran generates code

that uses the integer and floating-point registers instead of memory locations. Knowing when Visual Fortran uses registers may be helpful when doing certain forms of debugging.

Visual Fortran uses registers to hold the values of variables whenever the Fortran language does not require them to be held in memory, such as holding the values of temporary results of subexpressions, even if `/optimize:0` (no optimization) was specified.

Visual Fortran may hold the same variable in different registers at different points in the program:

```
V = 3.0*Q
.
.
X = SIN(Y)*V
.
.
V = PI*X
.
.
Y = COS(Y)*V
```

Visual Fortran may choose one register to hold the first use of *V* and another register to hold the second. Both registers can be used for other purposes at points in between. There may be times when the value of the variable does not exist anywhere in the registers. If the value of *V* is never needed in memory, it might not ever be assigned.

Visual Fortran uses registers to hold the values of *I*, *J*, and *K* (so long as there are no other optimization effects, such as loops involving the variables):

```
A(I) = B(J) + C(K)
```

More typically, an expression uses the same index variable:

```
A(K) = B(K) + C(K)
```

In this case, *K* is loaded into only one register, which is used to index all three arrays at the same time.

Mixed Real/Complex Operations

In mixed REAL/COMPLEX operations, Visual Fortran avoids the conversion and performs a simplified operation on:

- Add (+), subtract (-), and multiply (*) operations if either operand is REAL
- Divide (/) operations if the divisor is REAL

For example, if variable *R* is REAL and *A* and *B* are COMPLEX, no conversion occurs with the following:

```
COMPLEX A, B
```

```

      .
      .
      .
B = A + R

```

Global Optimizations

To enable global optimizations, use `/optimize:2` or a higher optimization level. Using `/optimize:2` or higher also enables local optimizations (`/optimize:1`).

Global optimizations include:

- Data-flow analysis
- Split lifetime analysis
- Strength reduction (replaces a CPU-intensive calculation with one that uses fewer CPU cycles)
- Code motion (also called code hoisting)
- Instruction scheduling

Data-flow and split lifetime analysis (global data analysis) traces the values of variables and whole arrays as they are created and used in different parts of a program unit. During this analysis, Visual Fortran assumes that any pair of array references to a given array might access the same memory location, unless constant subscripts are used in both cases.

To eliminate unnecessary recomputations of invariant expressions in loops, Visual Fortran hoists them out of the loops so they execute only once.

Global data analysis includes which data items are selected for analysis. Some data items are analyzed as a group and some are analyzed individually. Visual Fortran limits or may disqualify data items that participate in the following constructs, generally because it cannot fully trace their values.

Data items in the following constructs can make global optimizations less effective:

- **VOLATILE** declarations

VOLATILE declarations are needed to use certain run-time features of the operating system. Declare a variable as **VOLATILE** if the variable can be accessed using rules in addition to those provided by the Fortran 90 language. Examples include:

- **COMMON** data items or entire common blocks that can change value by means other than direct assignment or during a routine call. For such applications, you must declare the variable or the **COMMON** block to which it belongs as volatile.
- An address not saved by the **%LOC** built-in function.
- Variables read or written by a signal handler, including those in a common block or module.

As requested by the **VOLATILE** statement, Visual Fortran disqualifies any volatile variables from global data analysis.

- Subroutine calls or external function references

Visual Fortran cannot trace data flow in a called routine that is not part of the program unit being compiled, unless the same DF command compiled multiple program units (see [Compile](#)

With Appropriate Options and Multiple Source Files). Arguments passed to a called routine that are used again in a calling program are assumed to be modified, unless the proper **INTENT** is specified in an interface block (the compiler must assume they are referenced by the called routine).

- Common blocks

Visual Fortran limits optimizations on data items in common blocks. If common block data items are referenced inside called routines, their values might be altered. In the following example, variable I might be altered by FOO, so Visual Fortran cannot predict its value in subsequent references.

```
COMMON /X/ I

DO J=1,N
  I = J
  CALL FOO
  A(I) = I
ENDDO
```

- Variables in Fortran 90 modules

Visual Fortran limits optimizations on variables in Fortran 90 modules. Like common blocks, if the variables in Fortran 90 modules are referenced inside called routines, their values might be altered.

- Variables referenced by a %LOC built-in function or variables with the **TARGET** attribute

Visual Fortran limits optimizations on variables indirectly referenced by a %LOC function or on variables with the TARGET attribute, because the called routine may dereference a pointer to such a variable.

- Equivalence groups

An *equivalence group* is formed explicitly with the **EQUIVALENCE** statement or implicitly by the **COMMON** statement. A program section is a particular common block or local data area for a particular routine. Visual Fortran combines equivalence groups within the same program section and in the same program unit.

The equivalence groups in separate program sections are analyzed separately, but the data items within each group are not, so some optimizations are limited to the data within each group.

Additional Global Optimizations

To enable additional global optimizations, use /optimize:3 or a higher optimization level. Using /optimize:3 or higher also enables local optimizations (/optimize:1) and global optimizations (/optimize:2).

Additional global optimizations improve speed at the cost of longer compile times and possibly extra code size. These optimizations include:

- Loop unrolling, including instruction scheduling (see [Loop Unrolling](#))
- Code replication to eliminate branches (see [Code Replication to Eliminate Branches](#))
- Padding the size of certain power-of-two arrays to allow more efficient cache use (see [Use Arrays Efficiently](#))

Loop Unrolling

At optimization level /optimize:3 or above, Visual Fortran attempts to unroll certain innermost loops, minimizing the number of branches and grouping more instructions together to allow efficient overlapped instruction execution (instruction pipelining). The best candidates for loop unrolling are innermost loops with limited control flow.

As more loops are unrolled, the average size of basic blocks increases. Loop unrolling generates multiple copies of the code for the loop body (loop code iterations) in a manner that allows efficient instruction pipelining.

The loop body is replicated a certain number of times, substituting index expressions. An initialization loop might be created to align the first reference with the main series of loops. A remainder loop might be created for leftover work.

The number of times a loop is unrolled can be determined either by the optimizer or by using the [/unroll](#) option, which can specify the limit for loop unrolling. Unless the user specifies a value, the optimizer unrolls a loop four times for most loops or two times for certain loops (large estimated code size or branches out of the loop).

Array operations are often represented as a nested series of loops when expanded into instructions. The innermost loop for the array operation is the best candidate for loop unrolling (like **DO** loops). For example, the following array operation (once optimized) is represented by nested loops, where the innermost loop is a candidate for loop unrolling:

```
A(1:100,2:30) = B(1:100,1:29) * 2.0
```

Code Replication to Eliminate Branches

In addition to loop unrolling and other optimizations, the number of branches are reduced by replicating code that will eliminate branches. Code replication decreases the number of basic blocks (a stream of instructions entered only at the beginning and exited only at the end) and increases instruction-scheduling opportunities.

Code replication normally occurs when a branch is at the end of a flow of control, such as a routine with multiple, short exit sequences. The code at the exit sequence gets replicated at the various places where a branch to it might occur.

For example, consider the following unoptimized routine and its optimized equivalent that uses code replication, where R0 (EAX on x86 systems) is register 0:

Unoptimized Instructions Optimized (Replicated) Instructions

```

:
:

```


- One of the `/optimize:num` options to control the optimization level. For example, specifying `/optimize:4` or higher enables interprocedural optimizations.

Different `/optimize:num` options set different `/inline:keyword` options. For example, `/optimize:4` sets `/inline:speed`.

- One of the `/inline:keyword` options to directly control the inlining of procedures (see [Controlling the Inlining of Procedures](#)). For example, `/inline:speed` inlines more procedures than `/inline:size`.

Loop Transformation and Software Pipelining (Alpha only)

A group of optimizations known as loop transformation optimizations and software pipelining with its associated additional software dependence analysis are enabled by using the `/optimize:5` option on Alpha systems. In certain cases, this improves run-time performance.

The loop transformation optimizations apply to array references within loops and can apply to multiple nested loops. These optimizations can improve the performance of the memory system.

Software pipelining applies instruction scheduling to certain innermost loops, allowing instructions within a loop to "wrap around" and execute in a different iteration of the loop. This can reduce the impact of long-latency operations, resulting in faster loop execution.

Software pipelining also enables the prefetching of data to reduce the impact of cache misses.

For More Information:

- On loop transformations, see [Loop Transformation](#).
- On software pipelining, see [Software Pipelining](#).

Other Options Related to Optimization

In addition to the `/optimize:num` options (discussed in [Optimization Levels: the /optimize:num Option](#)), several other DF command options can prevent or facilitate improved optimizations, as discussed in the following sections:

- [Options Set by the /fast Option](#)
- [Controlling Loop Unrolling](#)
- [Controlling the Inlining of Procedures](#)
- [Arithmetic Reordering Optimizations](#)
- [Dummy Aliasing Assumption](#)
- [Requesting Optimized Code for a Specific Processor Generation \(Alpha only\)](#)
- [Requesting Code Generation for a Specific Processor Generation \(Alpha only\)](#)
- [Loop Transformation \(Alpha only\)](#)
- [Software Pipelining \(Alpha only\)](#)

Options Set by the /fast Option

Specifying the `/fast` option sets the following options:

- `/align:(dcommons,records)` (see [Data Alignment Considerations](#))

- `/assume:noaccuracy_sensitive` (see [Arithmetic Reordering Optimizations](#))
- `/math_library: fast` (see [/math_library](#))

Controlling Loop Unrolling

You can specify the number of times a loop is unrolled by using the `/unroll:num` option.

Although unrolling loops usually improves run-time performance, the size of the executable program may increase.

On Alpha systems, the `/unroll: num` option can also influence the run-time results of software pipelining optimizations performed when you specify `/optimize:5`.

For More Information:

On loop unrolling, see [Loop Unrolling](#).

Controlling the Inlining of Procedures

To specify the types of procedures to be inlined, use the `/inline:keyword` options. Also, compile multiple source files together and specify an adequate optimization level, such as `/optimize:4`.

If you omit `/noinline` and the `/inline:keyword` options, the optimization level `/optimize:num` option used determines the types of procedures that are inlined.

The `/inline:keyword` options are as follows:

- `/inline:none` (same as `/noinline`) inlines statement functions but not other procedures. This type of inlining occurs if you specify `/optimize:0` or `/optimize:1` and omit `/inline: xxxx` options.
- `/inline>manual` inlines statement functions but not other procedures. This type of inlining occurs if you omit `/inline: xxxx` options.
- In addition to inlining statement functions, `/inline:size` inlines any procedures that the DIGITAL Fortran optimizer expects will improve run-time performance with no likely significant increase in program size.
- In addition to inlining statement functions, `/inline:speed` inlines any procedures that the DIGITAL Fortran optimizer expects will improve run-time performance with a likely significant increase in program size. This type of inlining occurs if you specify `/optimize:4` (or `/optimize:4` or `/optimize:5` on Alpha systems) and omit `/inline: xxxx` options.
- `/inline:all` inlines every call that can possibly be inlined while generating correct code, including the following:
 - Statement functions (always inlined)
 - Any procedures that DIGITAL Fortran expects will improve run-time performance with a likely significant increase in program size.
 - Any other procedures that can possibly be inlined and generate correct code. Certain recursive routines are not inlined to prevent infinite expansion.

For information on the inlining of other procedures (inlined at optimization level `/optimize:4` or higher), see [Inlining Procedures](#).

Maximizing the types of procedures that are inlined usually improves run-time performance, but

compile-time memory usage and the size of the executable program may increase.

To determine whether using `/inline` all benefits your particular program, time program execution for the same program compiled with and without `/inline:all`.

Arithmetic Reordering Optimizations

If you use the `/assume:noaccuracy_sensitive` option, DIGITAL Fortran may reorder code (based on algebraic identities) to improve performance. For example, the following expressions are mathematically equivalent but may not compute the same value using finite precision arithmetic:

```
X = (A + B) + C
```

```
X = A + (B + C)
```

The results can be slightly different from the default `/assume:accuracy_sensitive` because of the way intermediate results are rounded. However, the `/assume:noaccuracy_sensitive` results are not categorically less accurate than those gained by the default. In fact, dot product summations using `/assume:noaccuracy_sensitive` can produce more accurate results than those using `/assume:accuracy_sensitive`.

The effect of `/assume:noaccuracy_sensitive` is important when DIGITAL Fortran hoists divide operations out of a loop. If `/assume:noaccuracy_sensitive` is in effect, the unoptimized loop becomes the optimized loop:

Unoptimized Code Optimized Code

```
DO I=1,N          T= 1/V
.                DO I=1,N
.                .
.                .
.                .
B(I)= A(I)/V      B(I)= A(I)*T
END DO           END DO
```

The transformation in the optimized loop increases performance significantly, and loses little or no accuracy. However, it does have the potential for raising overflow or underflow arithmetic exceptions.

Dummy Aliasing Assumption

Some programs compiled with Visual Fortran (or DIGITAL Fortran 77) may have results that differ from the results of other Fortran compilers. Such programs may be aliasing dummy arguments to each other or to a variable in a common block or shared through use association, and at least one variable access is a store.

This program behavior is prohibited in programs conforming to the Fortran 90 standard, but not by Visual Fortran. Other versions of Fortran allow dummy aliases and check for them to ensure correct results. However, Visual Fortran assumes that no dummy aliasing will occur, and it can ignore potential data dependencies from this source in favor of faster execution.

The Visual Fortran default is safe for programs conforming to the Fortran 90 standard. It will

improve performance of these programs because the standard prohibits such programs from passing overlapped variables or arrays as actual arguments if either is assigned in the execution of the program unit.

The `/assume:dummy_aliases` option allows dummy aliasing. It ensures correct results by assuming the exact order of the references to dummy and common variables is required. Program units taking advantage of this behavior can produce inaccurate results if compiled with `/assume:nodummy_aliases`.

The following example is taken from the DAXPY routine in the Fortran-77 version of the Basic Linear Algebra Subroutines (BLAS).

Example: Using the `/assume:dummy_aliases` Option

```

SUBROUTINE DAXPY(N,DA,DX,INCX,DY,INCY)
C      Constant times a vector plus a vector.
C      uses unrolled loops for increments equal to 1.

DOUBLE PRECISION DX(1), DY(1), DA
INTEGER I,INCX,INCY,IX,IY,M,MP1,N
C
IF (N.LE.0) RETURN
IF (DA.EQ.0.0) RETURN
IF (INCX.EQ.1.AND.INCY.EQ.1) GOTO 20

C      Code for unequal increments or equal increments
C      not equal to 1.
.
.
.
RETURN
C      Code for both increments equal to 1.
C      Clean-up loop

20    M = MOD(N,4)
      IF (M.EQ.0) GOTO 40
      DO I=1,M
         DY(I) = DY(I) + DA*DX(I)
      END DO

      IF (N.LT.4) RETURN
40    MP1 = M + 1
      DO I = MP1, N, 4
         DY(I) = DY(I) + DA*DX(I)
         DY(I + 1) = DY(I + 1) + DA*DX(I + 1)
         DY(I + 2) = DY(I + 2) + DA*DX(I + 2)
         DY(I + 3) = DY(I + 3) + DA*DX(I + 3)
      END DO

      RETURN
END SUBROUTINE

```

The second DO loop contains assignments to DY. If DY is overlapped with DA, any of the assignments to DY might give DA a new value, and this overlap would affect the results. If this overlap is desired, then DA must be fetched from memory each time it is referenced. The repetitious fetching of DA degrades performance.

Linking Routines with Opposite Settings

You can link routines compiled with the `/assume:dummy_aliases` option to routines compiled with `/assume:nodummy_aliases`. For example, if only one routine is called with dummy aliases, you can use `/assume:dummy_aliases` when compiling that routine, and compile all the other routines with `/assume:nodummy_aliases` to gain the performance value of that option.

Programs calling DAXPY with DA overlapping DY do not conform to the FORTRAN-77 and Fortran 90 standards. However, they are accommodated if `/assume:dummy_aliases` was used to compile the DAXPY routine.

Requesting Optimized Code for a Specific Processor Generation (Alpha systems)

You can specify the types of optimized code to be generated by using the `/tune:keyword` option. Regardless of the specified *keyword*, the generated code will run correctly on all implementations of the Alpha architecture. Tuning for a specific implementation can improve run-time performance; it is also possible that code tuned for a specific target may run slower on another target.

Specifying the correct keyword for `/tune:keyword` for the target Alpha processor generation type usually slightly improves run-time performance. Unless you request software pipelining, the run-time performance difference for using the wrong keyword for `/tune:keyword` (such as using `/tune ev4` for an ev5 processor) is usually less than 5%. When using software pipelining (using `/optimize:5`) with `/tune:keyword`, the difference can be more than 5%.

The combination of the specified keyword for `/tune:keyword` and the type of processor generation used has no effect on producing the expected correct program results.

The `/tune:keyword` keywords are described in [/tune \(Alpha only\)](#).

To request a specific set of instructions for an Alpha architecture generation, see the [/architecture](#) option.

Requesting Code Generation for a Specific Processor Generation (Alpha only)

You can specify whether the code to be generated will run on an ev56, pca56, or earlier Alpha processor. The `/architecture (/arch)` option determines the type of Alpha chip code that will be generated for this program. The `/arch:keyword` option uses the same keywords as the `/tune:keyword` option. This option is ignored on x86 processor systems.

For more information, see [/architecture \(Alpha only\)](#).

Loop Transformation (Alpha only)

The loop transformation optimizations are enabled by using the `/transform_loops` option or the `/optimize:5` option. Loop transformation attempts to improve performance by rewriting loops to make better use of the memory system. By rewriting loops, the loop transformation optimizations can increase the number of instructions executed, which can degrade the run-time performance of some programs.

To request loop transformation optimizations without software pipelining, do one of the following:

- Specify `/optimize:5` with `/nopipeline` (preferred method)
- Specify `/transform_loops` with `/optimize:4`, `/optimize:3`, or `/optimize:2`. This optimization is not performed at optimization levels below `/optimize:2`.

The loop transformation optimizations apply to array references within loops. These optimizations can improve the performance of the memory system and usually apply to multiple nested loops. The loops chosen for loop transformation optimizations are always *counted loops*. Counted loops are those loops that use a variable to count iterations in a manner that the number of iterations can be determined before entering the loop. For example, most DO loops are counted loops.

Conditions that typically prevent the loop transformation optimizations from occurring include subprogram references that are not inlined (such as an external function call), complicated exit conditions, and uncounted loops.

The types of optimizations associated with `/transform_loops` include the following:

- **Loop blocking**
Can minimize memory system use with multidimensional array elements by completing as many operations as possible on array elements currently in the cache. Also known as loop tiling.
- **Loop distribution**
Moves instructions from one loop into separate, new loops. This can reduce the amount of memory used during one loop so that the remaining memory may fit in the cache. It can also create improved opportunities for loop blocking.
- **Loop fusion**
Combines instructions from two or more adjacent loops that use some of the same memory locations into a single loop. This can avoid the need to load those memory locations into the cache multiple times and improves opportunities for instruction scheduling.
- **Loop interchange**
Changes the nesting order of some or all loops. This can minimize the stride of array element access during loop execution and reduce the number of memory accesses needed. Also known as loop permutation.
- **Scalar replacement**
Replaces the use of an array element with a scalar variable under certain conditions.
- **Outer loop unrolling**
Unrolls the outer loop inside the inner loop under certain conditions to minimize the number of instructions and memory accesses needed. This also improves opportunities for instruction scheduling and scalar replacement.

For More Information:

On the interaction of command-line options and timing programs compiled with the loop transformation optimizations, see [/\[no\]transform_loops](#).

Software Pipelining (Alpha only)

Software pipelining and additional software dependence analysis are enabled by using the `/pipeline` option or by the `/optimize:5` option. Software pipelining in certain cases improves run-time

performance.

The software pipelining optimization applies instruction scheduling to certain innermost loops, allowing instructions within a loop to "wrap around" and execute in a different iteration of the loop. This can reduce the impact of long-latency operations, resulting in faster loop execution.

Loop unrolling (enabled at `/optimize:3` or above) *cannot* schedule across iterations of a loop. Because software pipelining *can* schedule across loop iterations, it can perform more efficient scheduling to eliminate instruction stalls within loops.

For instance, if software dependence analysis of data flow reveals that certain calculations can be done before or after that iteration of the loop, software pipelining reschedules those instructions ahead of or behind that loop iteration, at places where their execution can prevent instruction stalls or otherwise improve performance.

Software pipelining also enables the prefetching of data to reduce the impact of cache misses.

Software pipelining can be more effective when you combine `/pipeline` (or `/optimize:5`) with the appropriate `/tune keyword` for the target Alpha processor generation (see [Requesting Optimized Code for a Specific Processor Generation](#)).

To specify software pipelining without loop transformation optimizations, do one of the following:

- Specify `/optimize:5` with `/nottransform_loops` (preferred method)
- Specify `/pipeline` with `/optimize:4`, `/optimize:3`, or `/optimize:2`. This optimization is not performed at optimization levels below `/optimize:2`.

For this version of Visual Fortran, loops chosen for software pipelining:

- Are always innermost loops (those executed the most).
- Do not contain branches or procedure calls.
- Do not use COMPLEX floating-point data.

By modifying the unrolled loop and inserting instructions as needed before and/or after the unrolled loop, software pipelining generally improves run-time performance, except where the loops contain a large number of instructions with many existing overlapped operations. In this case, software pipelining may not have enough registers available to effectively improve execution performance. Run-time performance using `/optimize:5` (or `/pipeline`) may not improve performance, as compared to using `/optimize:4`.

For programs that contain loops that exhaust available registers, longer execution times may result with `/optimize:5` or `/pipeline`. In cases where performance does not improve, consider compiling with the `/unroll 1` option along with `/optimize:5` or `/pipeline`, to possibly improve the effects of software pipelining.

For More Information:

On the interaction of command-line options and timing programs compiled with software pipelining, see [/\[no\]pipeline](#).

Creating Multithread Applications

Visual Fortran provides support for creating multithread applications. You should consider using more than one thread if your application needs to manage multiple activities, such as simultaneous keyboard input and calculations. One thread can process keyboard input while a second thread performs data transformation calculations. A third thread can update the display screen based on data from the keyboard thread. At the same time, other threads can access disk files, or get data from a communications port.

For resources about threads, processes, and multithreading, see [Other Sources of Information](#).

For more information, see:

- [Basic Concepts of Multithreading](#)
- [Writing a Multithread Program](#)
- [Compiling and Linking Multithread Programs](#)
- [Other Sources of Information](#)

Basic Concepts of Multithreading

A *thread* is a path of execution through a program. It is an executable entity that belongs to one and only one process. Each process has at least one thread of execution, automatically created when the process is created. Your main program runs in the first thread. A Win32 thread consists of a stack, the state of the CPU registers, a security context, and an entry in the execution list of the system scheduler. Each thread shares all of the process's resources.

A *process* consists of one or more threads and the code, data, and other resources of a program in memory. Typical program resources are open files, semaphores (a method of interthread communication), and dynamically allocated memory. A program executes when the system scheduler gives one of its threads execution control. The scheduler determines which threads should run and when they should run. Threads of lower priority might need to wait while higher priority threads complete their tasks. On multiprocessor machines, the scheduler can move individual threads to different processors to balance the CPU load.

Because threads require less system overhead and are easier to create than an entire process, they are useful for time- or resource-intensive operations that can be performed concurrently with other tasks. Threads can be used for operations such as background printing, monitoring a device for input, or backing up data while it is being edited.

When threads, processes, files, and communications devices are opened, the function that creates them returns a *handle*. Each handle has an associated Access Control List (ACL) that is used to check the security credentials of the process. Processes and threads can inherit a handle or give one away using functions described in this section. Objects and handles regulate access to system resources. For more information on handles and security, see the *Reference* for the Win32 Application Programming Interface.

All threads in a process execute independently of one another. Unless you take special steps to make them communicate with each other, each thread operates while completely unaware of the existence of other threads in a process. Threads sharing common resources must coordinate their work by

using semaphores or another method of interthread communication. For more information on interthread communication, see [Sharing Resources](#).

Writing a Multithread Program

Multiple threads are best used for:

- Background tasks such as data calculations, database queries, and input gathering, which do not directly involve window management or user interface.
- Operations that are independent from one another that can benefit from concurrent processing.
- Asynchronous tasks such as polling on a serial port.

If your application contains tasks that require a private address space and private resources, you can protect them from the activities of other threads by creating multiple processes rather than multiple threads. See [Working with Multiple Processes](#).

The sections that follow discuss the steps you need to consider in creating a multithread application:

- [Modules for Multithread Programs](#)
- [Starting and Stopping Threads](#)
- [Thread Routine Format](#)
- [Sharing Resources](#)
- [Thread Local Storage \(TLS\)](#)
- [Synchronizing Threads](#)
- [Handling Errors in Multithread Programs](#)
- [Working with Multiple Processes](#)
- [Table of Multithread Routines](#)

Modules for Multithread Programs

A module called DFMT.MOD is supplied with Visual Fortran. It contains interface statements to the underlying Win32 API routines as well as parameter and structure definitions used by the routines. You need to include a **USE DFMT** statement in the declarations section of every Fortran program unit (program, subroutine, function, or module) that uses multithread APIs.

The source code for the DFMT module (file name DFMT.F90) contains type definitions and external function declarations. You can use it as an added reference for the calling syntax, number, and type of arguments for a multithread procedure.

Other Windows APIs that support multithreading tasks (such as window management functions) are included in the DFWIN.F90 module, available to your programs with the **USE DFWIN** statement. For information about creating a Windows application, see [Creating Windows Applications](#).

Starting and Stopping Threads

When you add threads to a process, you need to consider the costs to your process. Create only the number of threads that help your application respond and perform better. You can save time by multitasking, but remember that additional CPU time is needed to keep track of multiple threads. When you are deciding how many threads to create, you also need to consider what data can be

process-specific, and what data is thread-specific. [Sharing Resources](#) discusses synchronizing access to variables and data.

One single call to the **CreateThread** function creates a thread, specifies security attributes and memory stack size, and names the routine for the thread to run. Windows allocates memory for the thread stack in the virtual address space of the application that contains the thread. Once a thread has finished processing, the **CloseHandle** routine frees the resources used by the thread.

For more information, see:

- [Starting Threads](#)
- [Stopping Threads](#)
- [Other Thread Support Functions](#)

Starting Threads

The function **CreateThread** creates a new thread. Its return value is an INTEGER(4) thread handle, used in communicating to the thread and when closing it. The syntax for this function is:

CreateThread (*security, stack, thread_func, argument, flags, thread_id*)

All arguments are INTEGER(4) variables except for *thread_func*, which names the routine for **CreateThread** to run. Minimum requirements for *thread_func* are discussed in [Thread Routine Format](#).

The first argument, *security*, is the SECURITY_ATTRIBUTES type, defined in DFMT.F90. If *security* is zero, the thread has the default security attributes of the parent process. For more information about setting security attributes for processes and threads, see the *Win32 API Reference*.

The second argument, *stack*, defines the stack size of the new thread. All of an application's default stack space is allocated to the first thread of execution. As a result, you must specify how much memory to allocate for a separate stack for each additional thread your program needs. The **CreateThread** call allows you to specify the value for the stack size on each thread you create. A value of zero indicates the stack has the same size as the application's primary thread. The size of the stack is increased dynamically, if necessary, up to a limit of 1 MB.

The third parameter, *thread_func*, is the starting address for the thread function.

The fourth parameter, *argument*, is an optional argument for *thread_func*. Your program defines this parameter and how it is used.

You can create a thread that will not begin processing until you signal it. The fifth parameter, *flags*, can take either of two values: 0, or CREATE_SUSPENDED. If you specify CREATE_SUSPENDED, the thread is created, but does not run until you call the **ResumeThread** function.

The last argument, *thread_id*, is returned by **CreateThread**. It is a unique identifier for the thread, which you can use when calling other multithread routines. While the thread is running, no other thread has the same identifier. However, the operating system may use the identifier again for other threads once this one has completed.

A thread can be referred to by its handle as well as its unique thread identifier. Synchronization functions such as **WaitForSingleObject** and **WaitForMultipleObjects** take the thread handle as an argument.

Stopping Threads

The **ExitThread** routine allows a thread to stop its own execution. The syntax is:

```
CALL EXITTHREAD ( [ Termination Status ] )
```

Termination status may be queried by another thread. A termination status of 0 indicates normal termination. You can assign other termination status values and their meaning in your program.

When the called thread is no longer needed, the calling thread needs to close the handle for the thread. Use the **CloseHandle** routine to free memory used by the thread. A thread object is not deleted until the last thread handle is closed.

It is possible for more than one handle to be open to a thread: for example, if a program creates two threads, one of which waits for information from the other. In this case, two handles are open to the first thread: one from the thread requesting information, the other from the thread that created it. All handles are closed implicitly when the enclosing process terminates.

The **TerminateThread** routine allows one thread to terminate another, if the security attributes are set appropriately for both threads. DLLs attached to the thread are not notified that the thread is terminating, and its initial stack is not deallocated. Use **Terminate Thread** for emergencies only.

Other Thread Support Functions

Scheduling thread priorities is supported through the functions **GetThreadPriority** and **SetThreadPriority**. Use the priority class of a thread to differentiate between applications that are time critical and those that have normal or below normal scheduling requirements. If you need to manipulate priorities, be very careful not to give a thread too high a priority, or it can consume all of the available CPU time. A thread with a base priority level above 11 interferes with the normal operation of the operating system. Using `REALTIME_PRIORITY_CLASS` may cause disk caches to not flush, hang the mouse, and so on.

When communicating with other threads, a thread uses a *pseudohandle* to refer to itself. A pseudohandle is a special constant that is interpreted as the current thread handle. Pseudohandles are only valid for the calling thread; they cannot be inherited by other threads. The **GetCurrentThread** function returns a pseudohandle for the current thread. The calling thread can use this handle to specify itself whenever a thread handle is required. Pseudohandles are not inherited.

To get the thread's identifier, use the **GetCurrentThreadId** function. The identifier uniquely identifies the thread in the system until it terminates. You can use the identifier to specify the thread itself whenever an identifier is required.

Use **GetExitCodeThread** to find out if a thread is still active, or if it is not, to find its exit status. Call **GetLastError** for more detailed information on the exit status. If one routine depends on a task being performed by a different thread, use the wait functions described in [Synchronizing Threads](#) instead of **GetExitCodeThread**.

Thread Routine Format

A function or subroutine that runs in a separate thread from the main program can take an argument. The code below shows a skeleton for a function and a subroutine:

```
INTEGER(4) FUNCTION thrdfnc(arg)
USE DFMT
integer(4) arg
arg = arg + 1      ! Sample only; real work goes here.
thrdfnc = 0       ! Sets exit code to 0.
END FUNCTION

SUBROUTINE thrdfnc2 (arg2)
USE DFMT
integer(4) arg2
                ! Subroutine work goes here.
Call exitthread(0) ! Exit code is 0.
END
```

The arguments `arg` or `arg2` are passed to the function or subroutine when the main program calls **CreateThread**, as the fourth argument.

Threads automatically terminate when the function or subroutine terminates.

Sharing Resources

Each thread has its own stack and its own copy of the CPU registers. Other resources, such as files, units, static data, and heap memory, are shared by all threads in the process. Threads using these common resources must coordinate their work. There are several ways to synchronize resources:

- Critical section--A block of code that accesses a non-shareable resource. Critical sections are typically used to restrict access to data or code that can only be used by one thread at a time within a process (for example, modification of shared data in a common block).
- MUTual EXclusion object (Mutex)--A mechanism that allows only one thread at a time to access a resource. Mutexes are typically used to restrict access to a system resource that can only be used by one thread at a time (for example, a printer), or when sharing might produce unpredictable results.
- Semaphore--A counter that regulates the number of threads that can use a resource. Semaphores are typically used to control access to a specified number of identical resources.
- Event--An event object announces that an event has happened to one or more threads.

The state of each of these objects is either signaled or not-signaled. A signaled state indicates a resource is available for a process or thread to use it. A not-signaled state indicates the resource is in use. The routines described in the following sections manage the creation, initialization, and termination of resource sharing mechanisms. Some of them change the state to signaled from not-signaled. The routines **WaitForSingleObject** and **WaitForMultipleObjects** also change the signal status of an object. For information on these functions, see Synchronizing Threads.

For resources about coordinating and synchronizing Win32 threads, see Other Sources of Information.

For more information, see:

- [Thread Stacks](#)
- [I/O Operations](#)

Critical Sections

Before you can synchronize threads with a critical section, you must initialize it by calling **InitializeCriticalSection**, passing to it the address of a global variable or COMMON block that different threads have access to. Call **EnterCriticalSection** when beginning to process the global variable, and **LeaveCriticalSection** when the application is finished with it. Both **EnterCriticalSection** and **LeaveCriticalSection** can be called several times within an application. For a Multithreaded Sample that uses Critical Sections, see PEEKAPP.F90.

Mutexes

CreateMutex creates a mutex object. It returns an error if the mutex already exists (one by the same name was created by another process or thread). Call **GetLastError** after calling **CreateMutex** to look for the error status ERROR_ALREADY_EXISTS. You can also use the **OpenMutex** function to determine whether or not a named mutex object exists. When called, **OpenMutex** returns the object's handle if it exists, or null if a mutex with the specified name is not found. Using **OpenMutex** does not change a mutex object to a signaled state; this is accomplished by one of the wait routines described in [Synchronizing Threads](#).

ReleaseMutex changes a mutex from the not-signaled state to the signaled state. This function only has an effect if the thread calling it also owns the mutex. When the mutex is in a signaled state, any thread waiting for it can acquire it and begin executing.

Semaphores

Functions for handling semaphores are nearly identical to functions that manage mutexes.

CreateSemaphore creates a semaphore, specifying an initial as well as a maximum count for the number of threads that can access the resource. **OpenSemaphore**, like **OpenMutex**, returns the handle of the named semaphore object, if it exists. The handle can then be used in any function that requires it (such as one of the wait functions described in [Synchronizing Threads](#)). Calling **OpenSemaphore** does not reduce a resource's available count; this is accomplished by the function waiting for the resource.

Use **ReleaseSemaphore** to increase the available count for a resource by a specified amount. You can call this function when the thread is finished with the resource. Another possible use is to call **CreateSemaphore**, specifying an initial count of zero to protect the resource from access during an initialization process. When the application has finished its initialization, call **ReleaseSemaphore** to increase the resource's count to its maximum.

Events

Event objects can trigger execution of other threads. You can use events if one thread provides data to several other threads. An event object is created by the **CreateEvent** function. The creating thread specifies the initial state of the object and whether it is a manual-reset or auto-reset event. A manual-reset event is one whose state remains signaled until it is explicitly reset by a call to **ResetEvent**. An auto-reset event is automatically reset by the system when a single waiting thread is released.

Use either **SetEvent** or **PulseEvent** to set an event object's state to signaled. **OpenEvent** returns a handle to the event, which can be used in other function calls. **ReleaseEvent** releases ownership of the event.

Memory Use and Thread Stacks

Because each thread has its own stack, you can avoid potential collisions over data items by using as little static data as possible. Design your program to use automatic stack variables for all data that can be private to a thread. All the variables declared in a multithread routine are by default static and shared among the threads. If you do not want one thread to overwrite a variable used by another, you can do one of the following:

- Declare the variable as **AUTOMATIC**.
- Create a vector of variable values, one for each thread, so that the variable values for different threads are in different storage locations. (You can use the single integer parameter passed by **CREATETHREAD** as an index to identify the thread.)
- Use Thread Local Storage (TLS).

Variables declared as automatic are placed on the stack, which is part of the thread context saved with the thread. Automatic variables within procedures are discarded when the procedure completes execution.

I/O Operations

Although files and units are shared between threads, you may not need to coordinate the use of these shared resources by threads. Fortran treats each input/output statement as an atomic operation. If two separate threads try to write to the same unit and one thread's output operation has started, the operation will complete before the other thread's output operation can begin.

The operating system does not impose an ordering on threads' access to units or files. For example, the non-determinate nature of multithread applications can cause records in a sequential file to be written in a different order on each execution of the application as each thread writes to the file. Direct access files might be a better choice than sequential files in such a case. If you cannot use direct access files, use mutexes to impose an ordering constraint on input or output of sequential files.

Certain restrictions apply to blocking functions for input procedures in QuickWin programs. For details on these restrictions, see [Using QuickWin](#).

Thread Local Storage

Thread Local Storage (TLS) calls allow you to store per-thread data. TLS is the method by which each thread in a multithreaded process can allocate locations in which to store thread-specific data.

Dynamically bound (run-time) thread-specific data is supported by routines such as **TlsAlloc** (allocates an index to store data), **TlsGetValue** (retrieves values from an index), **TlsSetValue** (stores values into an index), and **TlsFree** (frees the dynamic storage). Threads allocate dynamic storage and use **TlsSetValue** to associate the index with a pointer to that storage. When a thread needs to access the storage, it calls **TlsGetValue**, specifying the index.

When all threads have finished using the index, **TlsFree** frees the dynamic storage.

Synchronizing Threads

The routines **WAITFORSINGLEOBJECT** and **WAITFORMULTIPLEOBJECTS** enable threads to wait for a variety of different occurrences, such as thread completion or signals from other threads. They enable threads and processes to wait efficiently, consuming no CPU resources, either indefinitely or until a specified timeout interval has elapsed.

WAITFORSINGLEOBJECT takes an object handle as the first parameter and does not return until the object referenced by the handle either reaches a signaled state or until a specified timeout value elapses. The syntax is:

```
WaitResult = WAITFORSINGLEOBJECT (ObjectHandle, [ Timeout ] )
```

If you are using a timeout, specify the value in milliseconds as the second parameter. The value **WAIT_INFINITE** represents an infinite timeout, in which case the function waits until *ObjectHandle* completes.

WAITFORMULTIPLEOBJECTS is similar, except that its second parameter is an array of Windows object handles. Specify the number of handles to wait for in the first parameter. This can be less than the total number of threads created, and its maximum is 64. The function can either wait until all events have completed, or resume as soon as any one of the objects completes.

Deadlocks occur when a thread waits for objects that never become available. Use the timeout parameter when there is a chance that the thread you are waiting for may never terminate. See "Detecting Deadlocks in Multithreaded Win32 Applications," by Ruediger Asche, in *Microsoft Systems Journal*, vol. 8, for a discussion of how to find and avoid potential resource collisions.

Suspending and Resuming Threads

You can use **SuspendThread** to stop a thread from executing. **SuspendThread** is not particularly useful for synchronization because it does not control the point in the code at which the thread's execution is suspended. However, you could suspend a thread if you need to confirm a user's input that would terminate the work of the thread. If confirmed, the thread is terminated; otherwise, it resumes.

If a thread is created in a suspended state, it does not begin to run until **Resume Thread** is called with a handle to the suspended thread. This can be useful for initializing the thread's state before it begins to run. Suspending a thread at creation can be useful for one-time synchronization, because **ResumeThread** ensures that the suspended thread will resume running at the starting point of its code.

Handling Errors in Multithread Programs

Use the **GetLastError** function to obtain error information if any of the multithreading routines returns an error code. Remember that it returns the error code of the last error, not necessarily the error status of the last call.

Error codes are 32-bit values. Bit 29 is reserved for application-defined error codes. You can set this bit and use **SetLastError** if you are creating your own dynamic-link library, to emulate Win32 API behavior. Win32 functions only call **SetLastError** when they fail, not when they succeed.

The last error code value is kept in Thread Local Storage, so that multiple threads do not overwrite each other's values.

Working with Multiple Processes

The multithread libraries provide a number of routines for working with multiple processes. An application can use multiple processes for functions that require a private address space and private resources, to protect them from the activities of other threads. It is usually more efficient to implement multitasking by creating several threads in one process, rather than by creating multiple processes, for these reasons:

- The system can create and execute threads more quickly than it can create processes, since the code for threads has already been mapped into the address space of the process, while the code for a new process must be loaded.
- All threads of a process share the same address space and can access the process's global variables, which can simplify communications between threads.
- All threads of a process can use open handles to resources such as files and pipes.

If you want to create an independent process that runs concurrently with the current one, use **CreateProcess**. **CreateProcess** returns a 32-bit process identifier that is valid until the process terminates. **ExitProcess** stops the process and notifies all DLLs the process is terminating.

Different processes can share mutexes, events, and semaphores (but not critical sections). Processes can also optionally inherit handles from the process that created them (see Help for **CreateProcess**).

You can obtain information about the current process by calling **GetCurrentProcess** (returns a pseudohandle to its own process), and **GetCurrentProcessId** (returns the process identifier). The value returned by these functions can be used in calls to communicate with other processes. **GetExitCodeProcess** returns the exit code of a process, or an indication that it is still running.

The **OpenProcess** function opens a handle to a process specified by its process identifier. **OpenProcess** allows you to specify the handle's access rights and inheritability.

A process terminates whenever one of the following occurs:

- Any thread of the process calls **ExitProcess**
- The primary thread of the process returns
- The last thread of the process terminates
- **TerminateProcess** is called with a handle to the process

ExitProcess is the preferred way to terminate a process because it notifies all attached DLLs of the termination, and ensures that all threads of the process terminate. DLLs are not notified after a call to **TerminateProcess**.

Table of Multithread Routines

The following table lists routines available for multithread programs. For information about the calling syntax of these routines, see the *Platform SDK* Reference section in InfoViewer.

Routine	Description
CloseHandle	Closes an open object handle.
CreateEvent	Creates a named or unnamed event object.
CreateMutex	Creates a named or unnamed mutex object.
CreateProcess	Creates a new process and its primary thread.
CreateSemaphore	Creates a named or unnamed semaphore object.
CreateThread	Creates a thread to execute within the address space of the calling process.
DeleteCriticalSection	Releases all resources used by an unowned critical section object.
DuplicateHandle	Duplicates an object handle.
EnterCriticalSection	Waits for ownership of the specified critical section object.
ExitProcess	Ends a process and all its threads.
ExitThread	Ends a thread.
GetCurrentProcess	Returns a pseudohandle for the current process.
GetCurrentProcessId	Returns the process identifier of the calling process.
GetCurrentThread	Returns a pseudohandle for the current thread.
GetCurrentThreadId	Returns the thread identifier of the calling thread.
GetExitCodeProcess	Retrieves the termination status of the specified process.
GetExitCodeThread	Retrieves the termination status of the specified thread.
GetLastError	Returns the calling thread's last-error code value.
GetPriorityClass	Returns the priority class for the specified process.
GetThreadPriority	Returns the priority value for the specified thread.
InitializeCriticalSection	Initializes a critical section object.
LeaveCriticalSection	Releases ownership of the specified critical section object.
OpenEvent	Returns a handle of an existing named event object.
OpenMutex	Returns a handle of an existing named mutex object.
OpenProcess	Returns a handle of an existing process object.
OpenSemaphore	Returns a handle of an existing named semaphore object.
PulseEvent	As a single operation, sets (to signaled) and then resets the state of the specified event object after releasing the appropriate number of waiting threads.
ReleaseMutex	Releases ownership of the specified mutex object.
ReleaseSemaphore	Increases the count of the specified semaphore object by a specified amount.
ResetEvent	Sets the state of the specified event object to nonsignaled.
ResumeThread	Decrements a thread's suspend count. When the suspend count is zero, execution of the thread resumes.
SetEvent	Sets the state of the specified event object to signaled.
SetLastError	Sets the last-error code for the calling thread.
SetPriorityClass	Sets the priority class for the specified process.
SetThreadPriority	Sets the priority value for the specified thread.
SuspendThread	Suspends the specified thread.

TerminateProcess	Terminates the specified process and all of its threads.
TerminateThread	Terminates a thread.
TlsAlloc	Allocates a thread local storage (TLS) index.
TlsFree	Releases a thread local storage (TLS) index, making it available for reuse.
TlsGetValue	Retrieves the value in the calling thread's thread local storage (TLS) slot for a specified TLS index.
TlsSetValue	Stores a value in the calling thread's thread local storage (TLS) slot for a specified TLS index.
WaitForMultipleObjects	Returns either any one or all of the specified objects are in the signaled state or when the time-out interval elapses.
WaitForSingleObject	Returns when the specified object is in the signaled state or the time-out interval elapses.

If a function mentioned in this section is not listed in the preceding table, then it is only available through the **USE DFWIN** statement.

Compiling and Linking Multithread Programs

The support library DFORMAT.LIB is a re-entrant library for creating statically linked multithread programs. The DFORMD.LIB library, which calls code in the shared DFORMD.DLL, is also re-entrant. Programs built with DFORMAT.LIB do not share Fortran run-time library code or data with any dynamic-link libraries they call. You must link with DFORMD.LIB if you plan to call a DLL.

To build a multithread application that uses the Fortran run-time libraries, you must tell the linker to use a special version of the libraries. You can specify the /threads compiler option from the command line, or in Microsoft Developer Studio in the Project Settings window, as described in the following paragraph.

A sample multithread project and source file, THREADS.MAK and THREADS.F90, are included in the \DF\SAMPLES\ADVANCED\WIN32\THREADS subdirectory and \DF\SAMPLES\TUTORIAL subdirectory, respectively. To build this sample, open the project THREADS.MAK and choose Build All from the Build menu. Listed following are the steps for compiling and linking your own multithread program using the Microsoft Developer Studio.

► To compile and link your multithread program:

1. Create a new project. Choose the Project tab, then specify the Project type. (The sample THREADS.F90 is a QuickWin project.)
2. Add the file containing the source code to the project.
3. From the Project menu, select Settings.
The Project Settings dialog box appears.
4. Choose the Fortran tab, Fortran Libraries category, and set the Runtime Libraries to Multithread Libraries (DFORMAT.LIB) or Multithread Libraries in a DLL (DFORMD.LIB).
5. Create the executable file by choosing Build All from the Build menu.

The following steps describe how to compile and link the sample multithread program from the command line.

► **To compile and link the sample multithread program from the command line:**

1. Make sure the library files directory is specified in your LIB environment variable.
2. Compile and link the program with the DF command-line option `/threads`.

For example:

```
DF /threads MYTHREAD.F90
```

The `/threads` compiler option (automatically set when you specify a multithread application in Developer Studio) tells the linker to use `DFORMAT.LIB` as a default library.

To compile and link the `THREADS.F90` sample, the command is:

```
DF /libs=qwin THREADS.F90
```

The `/threads` compiler option causes the linker to search the multithread library; the `/libs=qwin` requests a Quickwin multiple window application.

Select the compiler options `/libs=dll` and `/threads` if you are using both multithread code and DLLs. You can use the `/libs=dll` and `/threads` options only with console projects, not QuickWin applications.

Other Sources of Information

For a thorough discussion of threads, processes, and multithreading, see Helen Custer's *Inside Windows NT*, available from Microsoft Press. Articles on how to accomplish multithreading have also been published in the Microsoft Developer Network CD and the *Microsoft Systems Journal*:

- The Microsoft Developer Network CD contains several articles on multithreading:
 - "Multiple Threads in the User Interface," by Nancy Winnick Cluts, discusses the ramifications of adding multiple threads to the user interface. This article not only offers alternatives to multiple threads, but also covers window management and message loops for multithreading.
 - "Multithreading for Rookies," by Ruediger R. Asche, focuses on practical applications of multithreading.
 - "Detecting Deadlocks in Multithreaded Win32 Applications," by Ruediger R. Asche, presents deadlock detection techniques. A deadlock is a condition in which the application hangs because two or more threads are waiting for each other to release a shared resource before resuming execution.
 - "Moving Unix Applications to Windows NT," provides an overview of Windows multithreading calls, contrasting them with Unix `fork()` calls.
- The Microsoft Systems Journal is also a source of information on multithreading:
 - "Coordinate Win32 threads using manual-reset and auto-reset events," by Jeffrey Richter. October 1993, v8 n10.
 - "Synchronizing Win32 threads using critical sections, semaphores, and mutexes," by Jeffrey Richter. August 1993, v8 n8.

Programming with Mixed Languages

Mixed-language programming is the process of building programs in which the source code is written in two or more languages. It allows you to:

- Call existing code that is written in another language
- Use procedures that may be difficult to implement in a particular language
- Gain advantages in processing speeds

Mixed-language programming is possible among the 32-bit languages Visual Fortran, Visual C/C++, Visual Basic, and MASM. Mixed-language programming in Win32 is different from that in 16-bit environments, and in many respects it is easier.

To properly create mixed-language programs, rules must be established for naming variables and procedures, for stack use, and for argument passing among routines written in different languages. These rules, as a whole, are the calling convention.

A calling convention includes:

- Stack considerations
 - Does a routine receive a varying or fixed number of arguments?
 - Which routine clears the stack after a call?
- Naming conventions
 - Is lowercase or uppercase significant or not significant?
 - Are names decorated (as in Visual C++)?
- Argument passing protocol
 - Are arguments passed by value or by reference?
 - What are the equivalent data types and data structures among languages?

This section provides information on the calling conventions available when writing routines written in Fortran, C, Visual C++, Visual Basic, and x86 assembly language. It is organized into the following topics:

- [Overview of Mixed-Language Issues](#)
- [Exchanging and Accessing Data in Mixed-Language Programming](#)
- [Handling Data Types in Mixed-Language Programming](#)
- [Visual Fortran/Visual C++ Mixed-Language Programs](#)
- [Fortran/Visual Basic Mixed-Language Programs](#)
- [Fortran/MASM Mixed-Language Programs](#)

Overview of Mixed-Language Issues

Mixed-language programming involves a call from a routine written in one language to a function, procedure, or subroutine written in another language. For example, a Fortran main program may need to execute a specific task that you want to program separately in an assembly-language procedure, or you may need to call an existing DLL or system procedure.

Mixed-language programming is possible with Visual Fortran, Visual C/C++, Visual Basic, and assembly language (MASM) because each language implements functions, subroutines, and

procedures in approximately the same way. The following table shows how different kinds of routines from each language correspond to each other. For example, a C main program could call an external **void** function, which is actually implemented as a Fortran subroutine.

Language Equivalents for Calls to Routines

Language	Call with return value	Call with no return value
Fortran	FUNCTION	SUBROUTINE
C and Visual C++	function	(void) function
Visual Basic	Function	Sub
Assembly language	Procedure	Procedure

There are some important differences in the way languages implement routines. Argument passing, naming conventions and other interface issues must be thoughtfully and consistently reconciled between any two languages to prevent program failure and indeterminate results. However, the advantages of mixed-language programming often make the extra effort worthwhile.

A summary of a few mixed-language advantages and restrictions follows:

- **Fortran/Assembly Language**

Assembly-language routines are small and execute very quickly because they don't require initialization as do high-level languages like Fortran and C. Also, they allow access to hardware instructions unavailable to the high-level language user. In a Fortran/assembly-language program, compiling the main routine in Fortran gives the assembly code access to Fortran high-level procedures and library functions, yet allows freedom to tune the assembly-language routines for maximum speed and efficiency. The main program can also be an assembly-language program.

- **Fortran/Visual Basic**

A mix of Fortran and Visual Basic 4.0 or higher (32-bit) allows you to use the easy-to-implement user-interface features of Visual Basic, yet do all your computation, especially floating-point math, in Fortran routines. In a Fortran/Visual Basic program, the main routine must be Visual Basic. It is not possible to call Basic routines from Fortran.

- **Fortran/C (or C++)**

Generally, Fortran/C programs are mixed to allow one to use existing code written in the other language. Either Fortran or C can call the other, so the main routine can be in either language.

To use the same Developer Studio environment for multiple languages, you must have the same version of Developer Studio for your languages (see [Mixed-Language Development Support](#)).

This section provides an explanation of the keywords, attributes, and techniques you can use to reconcile differences between Fortran and other languages. Adjusting calling conventions, adjusting naming conventions and writing interface procedures are discussed in the next sections:

- [Adjusting Calling Conventions in Mixed-Language Programming](#)
- [Adjusting Naming Conventions in Mixed-Language Programming](#)
- [Prototyping a Procedure in Fortran](#)

After establishing a consistent interface between mixed-language procedures, you then need to reconcile any differences in the treatment of individual data types (strings, arrays, and so on). This is discussed in [Exchanging and Accessing Data in Mixed-Language Programming](#).

Note: This section uses the term "routine" in a generic way, to refer to functions, subroutines, and procedures from different languages.

Adjusting Calling Conventions in Mixed-Language Programming

The calling convention determines how a program makes a call to a routine, how the arguments are passed, and how the routines are named (discussed in the section on [Adjusting Naming Conventions in Mixed-Language Programming](#)). In a single-language program, calling conventions are nearly always correct, because there is one default for all routines and because header files or Fortran module files with interface blocks enforce consistency between the caller and the called routine.

In a mixed-language program, different languages cannot share the same header files. If, as a result, you link Fortran and C routines that use different calling conventions, the error isn't apparent until the bad call is made at run-time. During execution, the bad call causes indeterminate results and/or a fatal error, often somewhere in the program that has no apparent relation to the actual cause: memory/stack corruption due to calling errors. Therefore, you should check carefully the calling conventions for each mixed-language call.

The discussion of calling conventions between languages applies only to external procedures. You cannot call internal procedures from outside the program unit that contains them.

A calling convention affects programming in five ways:

- The caller routine uses a calling convention to determine the order in which to pass arguments to another routine; the called routine uses a calling convention to determine the order in which to receive the arguments passed to it. In Fortran, you can specify these conventions in a mixed-language interface with the **INTERFACE** statement or in a data or function declaration. 32-bit Visual C/C++ and Fortran both pass arguments in order from left to right.
- The caller routine and the called routine use a calling convention to determine which of them is responsible for adjusting the stack in order to remove arguments when the execution of the called routine is complete. You can specify these conventions with **ATTRIBUTES** (**cDEC\$ ATTRIBUTES** compiler directive) options such as C or STDCALL.
- The caller routine and the called routine use a calling convention to select the option of passing a variable number of arguments.
- The caller routine and the called routine use a calling convention to pass arguments by value (values passed) or by reference (addresses passed). Individual Fortran arguments can also be designated with **ATTRIBUTES** option VALUE or REFERENCE.
- The caller routine and the called routine use a calling convention to establish naming conventions for procedure names. You can establish any procedure name you want, regardless of its Fortran name, with the **ALIAS** directive (or **ATTRIBUTES** option ALIAS). This is useful because C is case sensitive, while Fortran is not.

Different Fortran calling conventions can be specified by declaring the Fortran procedure to have certain attributes. For example, on x86 systems:

```

INTERFACE
  SUBROUTINE MY_SUB (I)
    !DEC$ ATTRIBUTES C, ALIAS:'_My_Sub' :: MY_SUB    ! x86 systems
    INTEGER I
  END SUBROUTINE MY_SUB
END INTERFACE

```

This code declares a subroutine named MY_SUB with the C property and the external name _My_Sub set with the ALIAS property on x86 systems.

On Alpha systems, the leading underscore for the external name _My_Sub is removed, so the !DEC\$ ATTRIBUTES line is as follows:

```

!DEC$ ATTRIBUTES C, ALIAS:'My_Sub' :: MY_SUB    ! Alpha systems

```

To write code for both x86 and Alpha platforms, use the conditional compilation features of the IF Directive Construct, perhaps with the platform-specific preprocessor macros (such as _X86_ and _ALPHA_) defined under the /define option.

The **ATTRIBUTES** options C, STDCALL, REFERENCE, VALUE, and VARYING all affect the calling convention of routines. By default, Fortran passes all data by reference (except the hidden length argument of strings, which is a special case). If the C or STDCALL option is used, the default changes to passing almost all data by value except arrays. However, in addition to the calling-convention options C and STDCALL, you can specify argument options, VALUE and REFERENCE, to pass arguments by value or by reference, regardless of the calling convention option. Arrays can only be passed by reference.

The following table summarizes the effect of the most common Fortran calling-convention directives.

Calling Conventions for ATTRIBUTES Options

	Default	C	STDCALL	C, REFERENCE	STDCALL, REFERENCE
Argument					
Scalar	Reference	Value	Value	Reference	Reference
Scalar [value]	Value	Value	Value	Value	Value
Scalar [reference]	Reference	Reference	Reference	Reference	Reference
String	Reference, either Len: Mixed or Len:End	String(1:1)	String(1:1)	Reference, either Len: Mixed or Len:End	Reference, No Len
String [value]	Error	String(1:1)	String(1:1)	String(1:1)	String(1:1)
String [reference]	Reference, either Len: Mixed or No Len	Reference, No Len	Reference, No Len	Reference, No Len	Reference, No Len
Array	Reference	Reference	Reference	Reference	Reference
Array [value]	Error	Error	Error	Error	Error

Array [reference]	Reference	Reference	Reference	Reference	Reference
Derived Type	Reference	Value, size dependent	Value, size dependent	Reference	Reference
Derived Type [value]	Value, size dependent	Value, size dependent	Value, size dependent	Value, size dependent	Value, size dependent
Derived Type [reference]	Reference	Reference	Reference	Reference	Reference
F90 Pointer	Descriptor	Descriptor	Descriptor	Descriptor	Descriptor
F90 Pointer [value]	Error	Error	Error	Error	Error
F90 Pointer [reference]	Descriptor	Descriptor	Descriptor	Descriptor	Descriptor
Procedure Name					
Suffix	@n (x86 systems)	none	@n (x86 systems)	none	@n (x86 systems)
Case	Upper Case	Lower Case	Lower Case	Lower Case	Lower Case
Stack Cleanup	Callee	Caller	Callee	Caller	Callee

The terms in the above table mean the following:

[value]	Assigned the VALUE property.
[reference]	Assigned the REFERENCE property.
Value	The argument value is pushed on the stack. All values are padded to the next 4-byte boundary.
Reference	The 4-byte argument address is pushed on the stack.
Len: Mixed or Len: End	For certain string arguments: <ul style="list-style-type: none"> • Len: Mixed applies when <code>/iface:mixed str len arg</code> is set. The length of the string is pushed (by value) on the stack immediately after the address of the beginning of the string. • Len: End applies when <code>/iface:nomixed str len arg</code> is set. The length of the string is pushed (by value) on the stack after all of the other arguments.
Len: Mixed or No Len	For certain string arguments: <ul style="list-style-type: none"> • Len: Mixed applies when <code>/iface:mixed str len arg</code> is set. The length of the string is pushed (by value) on the stack immediately after the address of the beginning of the string. • No Len applies when <code>//iface:nomixed str len arg</code> is set. The length of the string is not available to the called procedure.
No Len	For string arguments, the length of the string is not available to the called procedure.
String(1:1)	For string arguments, the first character is converted to INTEGER(4) as in ICHAR(string(1:1)) and pushed on the stack by value.
Error	Produces a compiler error.
Descriptor	4-byte address of the array descriptor.
@n	On x86 systems, the at sign (@) followed by the number of bytes (in decimal) required for the argument list.
Size	Derived-type arguments specified by value are passed as follows:

dependent	<ul style="list-style-type: none"> Arguments from 1 to 4 bytes are passed by value. Arguments from 5 to 8 bytes are in two registers (two arguments). Arguments more than 8 bytes provide value semantics by passing a temporary storage address by reference.
Upper Case	Procedure name in all uppercase.
Lower Case	Procedure name in all lowercase.
Callee	The procedure being called is responsible for removing arguments from the stack before returning to the caller.
Caller	The procedure doing the call is responsible for removing arguments from the stack after the call is over.

The following table shows which Fortran **ATTRIBUTES** options match other language calling conventions.

Matching Calling Conventions

Other language calling convention	Matching ATTRIBUTES option
Visual C/C++ cdecl (default)	C option
Visual C/C++ __stdcall	STDCALL option
Visual Basic	none
Visual Basic CDECL keyword	C option
MASM C (in PROTO and PROC declarations)	C option
MASM STDCALL (in PROTO and PROC declarations)	STDCALL option

Note that the ALIAS option can be used with any other Fortran calling-convention option to preserve mixed-case names.

Note: When interfacing to the Windows graphical user interface or making API calls, you will typically use STDCALL. See [Advanced Applications](#), for more information on Windows programming.

Specific calling-convention issues are discussed in the following sections:

- [Stack Considerations in Calling Conventions](#)
- [Fortran/C Calling Conventions](#)
- [Fortran/Visual Basic Calling Conventions](#)
- [Fortran/MASM Calling Conventions](#)

Stack Considerations in Calling Conventions

In the **C** calling convention, the calling routine always adjusts the stack immediately after the called routine returns control. This produces slightly larger object code because the code that restores the stack must exist at every point a procedure is called. In the STDCALL calling convention, the called procedure controls the stack. The code to restore the stack resides in the called procedure, so the code needs to appear only once.

However, the **C** calling convention makes calling with a variable number of arguments possible. Since in the **C** calling convention the caller cleans up the stack, it is possible to write a routine with a variable number of arguments. Therefore, it has the same address relative to the frame pointer, regardless of how many arguments are actually passed. Because of this, when the calling routine

controls the stack, it knows how many arguments it passed, how big they are and where they reside in the stack. It can thus skip passing an argument and still keep track.

You can call routines with a variable number of arguments by including the **ATTRIBUTES C** and **VARYING** options in your interface to a routine. The **VARYING** option prevents Fortran from enforcing a matching number of arguments in routines. The **VARYING** option is not necessary with intrinsic Fortran 90 routines with optional arguments, where argument order and/or keywords determine which arguments are present and which are absent.

In MASM, stack control is also set by the **C** or **STDCALL** convention declared for the procedure, but you can write MASM code to control the stack within the procedure any way you wish. In addition, you can specify the **USES** option in the **PROC** directive to save and restore certain registers automatically.

Fortran/C Calling Conventions

In C and Visual C++ modules, you can specify the **STDCALL** calling convention by using the **__stdcall** keyword in a function prototype or definition. The **__stdcall** convention is also used by window procedures and API functions. As an example, the following C language prototype sets up a function call to a subroutine using the **STDCALL** calling convention:

```
extern void __stdcall FORTRAN_ROUTINE (int n);
```

Alternatively, instead of changing the calling convention of the C code, you can adjust the Fortran source code by using the **C** option. This is set with the **ATTRIBUTES** directive. For example, the following declaration assumes the subroutine is called with the **C** calling convention:

```
SUBROUTINE CALLED_FROM_C (A)
  !DEC$ ATTRIBUTES C :: CALLED_FROM_C
  INTEGER A
```

Fortran/Visual Basic Calling Conventions

You establish Fortran subroutines and functions in Visual Basic forms and the Fortran routines are then invoked from a Basic module. A Fortran routine has to be a DLL (dynamic-link library) to be called from Basic. For more information on DLLs, see [Advanced Applications](#) and [Dynamic-Link Library Projects](#).

The calling-convention **ALIAS** directive (or **ATTRIBUTES** option **ALIAS**) is needed if mixed-case names are to be preserved (by default Fortran translates names to all uppercase). However, two special cases require different treatment:

- If a varying number of arguments are to be passed, the **C** and **VARYING** options are needed in the Fortran procedure definition and the **CDECL** keyword needed in the Basic **DECLARE** statement in order to establish the **C** calling and naming convention.
- When passing character arguments, the Fortran routine must use the **ATTRIBUTES** option **STDCALL** so it does not expect the hidden length of the character arguments. Since **STDCALL** also lowercases Fortran names, the Fortran subprogram name should be referenced in lowercase from the Visual Basic program.

The following Fortran and Visual Basic statements establish an example Fortran function to be

called from Basic:

```
! Fortran Subprogram establishing Fortran function.
  INTERFACE
    DOUBLE PRECISION FUNCTION GetFVal (r1)
      !DEC$ ATTRIBUTES ALIAS:'GetFVal' :: GetFVal
      !DEC$ ATTRIBUTES VALUE :: r1
      REAL r1
    END FUNCTION
  END INTERFACE
'FORM.FRM Basic Form to establish Fortran function
Declare Function GetFVal Lib "C:\f90\FVAL.DLL" (ByVal r1 As Single) As Double
```

Fortran/MASM Calling Conventions

You specify the calling convention for a MASM procedure in the **PROTO** and **PROC** directives. The **STDCALL** option in the **PROTO** and **PROC** directives tells the procedure to use the **STDCALL** calling convention. The **C** option in the **PROTO** and **PROC** directives tells the procedure to use the **C** calling convention. The **USES** option in the **PROC** directive specifies which registers to save and restore in the called MASM routine. The **VARARG** option to the **PROTO** and **PROC** directives specifies that the procedure allows a variable number of arguments.

As an example, the following Fortran and MASM statements set up a MASM function that can be called from Visual Fortran, using the **STDCALL** calling convention:

```
!Fortran STDCALL prototype.
  INTERFACE
    INTEGER FUNCTION forfunc(I1, I2)
      !DEC$ ATTRIBUTES STDCALL :: forfunc
      INTEGER I1
      INTEGER(2) I2
    END INTERFACE
  WRITE (*,*) forfunc(I1,I2)

;MASM STDCALL Prototype
  .MODEL FLAT, STDCALL
forfunc PROTO STDCALL, forint: SDWORD, shorti: SWORD
  .CODE
forfunc PROC STDCALL, forint: SDWORD, shorti: SWORD
  ...
forfunc ENDP END
```

The following Fortran and MASM statements set up a Fortran-callable MASM function using the **C** calling convention:

```
!Fortran C prototype
  INTERFACE
    INTEGER FUNCTION Forfunc (I1, I2)
      !DEC$ATTRIBUTES C, ALIAS:'Forfunc' :: Forfunc
      INTEGER I1
      INTEGER(2) I2
    END INTERFACE
  WRITE(*,*) Forfunc (I1, I2)
  END

; MASM C PROTOTYPE
  .MODEL FLAT, C
Forfunc PROTO C, forint:SDWORD, shorti:SWORD
  .CODE
Forfunc PROC C, forint:SDWORD, shorti:SWORD
```

```

    ...
Forfunc ENDP END

```

Adjusting Naming Conventions in Mixed-Language Programming

C and STDCALL determine naming conventions as well as calling conventions. Calling conventions specify how arguments are moved and stored; naming conventions specify how symbol names are altered when placed in an .OBJ file. Names are an issue for external data symbols shared among parts of the same program as well as among external routines. Symbol names (such as the name of a subroutine) identify a memory location that must be consistent among all calling routines.

Parameter names (names given in a procedure definition to variables that are passed to it) are never affected. Names are altered because of case sensitivity (in C, Visual Basic, and MASM), lack of case sensitivity (in Fortran), name decoration (in Visual C++), or other issues. If naming conventions are not reconciled, the program cannot successfully link and you will receive an "unresolved external" error.

This section also discusses:

- [Visual C/C++ and Visual Basic Naming Conventions](#)
- [MASM Naming Conventions](#)
- [Summary of Naming Conventions](#)
- [How to reconcile names between languages in four common cases](#)

Visual C/C++ and Visual Basic Naming Conventions

Visual C/C++ and Visual Basic preserve case sensitivity in their symbol tables while Fortran by default does not, a difference that requires attention. Fortunately, you can use the Fortran **ATTRIBUTES ALIAS** option to resolve discrepancies between names, to preserve mixed-case names, or to override the automatic conversion of names to all uppercase by the Fortran default naming, or the automatic conversion to all lowercase by Fortran's STDCALL and C naming convention.

MASM Naming Conventions

In MASM (Microsoft Assembler, for x86 systems), specifying the C or STDCALL naming convention in **PROC** and **PROTO** statements preserves case sensitivity if no **CASEMAP** option exists. The MASM **OPTION CASEMAP** directive (and the command line option /C) also sets case sensitivity and overrides naming conventions specified within **PROTO** and **PROC** statements. **CASEMAP: NONE** (equivalent to /Cx) preserves the case of identifiers in **PUBLIC**, **COMM**, **EXTERNDEF**, **EXTERN**, **PROTO**, and **PROC** declarations. **CASEMAP: NOTPUBLIC** (equivalent to /Cp) preserves the case of all user identifiers; this is the default. **CASEMAP: ALL** (equivalent to /Cu) translates all identifiers to uppercase.

Summary of Naming Conventions

The following table summarizes how Fortran, Visual C/C++, Visual Basic and MASM handle procedure names. Note that for MASM, the table does not apply if the **CASEMAP: ALL** option is used.

Naming Conventions in Fortran, C, Visual C++, Visual Basic, and MASM

Language	Attributes	Name translated as	Case of name in .OBJ file
Fortran	<code>cDEC\$ ATTRIBUTES C</code>	<code>_name</code>	All lowercase
	<code>cDEC\$ ATTRIBUTES STDCALL</code>	<code>_name@n</code>	All lowercase
	default	<code>_name@n</code>	All uppercase
C	<code>cdecl</code> (default)	<code>_name</code>	Mixed case preserved
	<code>__stdcall</code>	<code>_name@n</code>	Mixed case preserved
Visual C++	Default	<code>_name@_@decoration</code>	Mixed case preserved
Visual Basic	Default	<code>_name@n</code>	Mixed case preserved
MASM	<code>C</code> (in <code>PROTO</code> and <code>PROC</code> declarations)	<code>_name</code>	Mixed case preserved
	<code>STDCALL</code> (in <code>PROTO</code> and <code>PROC</code> declarations)	<code>_name@n</code>	Mixed case preserved

In the preceding table:

- The leading underscore (such as `_name`) is used on *x86* systems only (not on Alpha systems).
- `@n` represents the stack space, in decimal notation, occupied by parameters on *x86* systems only (not on Alpha systems).

For example, assume a function is declared in C as:

```
extern int __stdcall Sum_Up( int a, int b, int c );
```

Each integer occupies 4 bytes, so the symbol name placed in the .OBJ file on *x86* systems is:

```
_Sum_Up@12
```

On Alpha systems, the symbol name placed in the .OBJ file is:

```
Sum_Up
```

The following sections describe how to reconcile names between languages in four common cases:

- [Calls to Fortran, where Fortran cannot be recompiled \(use uppercase names\)](#)
- [Symbol names that are all lowercase](#)
- [Mixed-case names](#)
- [Fortran module names](#)
- [Visual C++ names](#)

All-Uppercase Names

If you call a Fortran routine that uses Fortran defaults and cannot recompile the Fortran code, then in C and Visual Basic you must use an all-uppercase name to make the call. In MASM you must either use an all-uppercase name or set the **OPTION CASEMAP** directive to **ALL**, which translates all identifiers to uppercase. Use of the `__stdcall` convention in C code or **STDCALL** in MASM **PROTO** and **PROC** declarations is not enough, because `__stdcall` and **STDCALL** always preserve case. Fortran generates all-uppercase names by default and the C or MASM code must match it.

For example, these prototypes establish the Fortran function `FFARCTAN(angle)` where the argument

angle has the **ATTRIBUTES** VALUE property:

- In C,

```
extern float __stdcall FFARCTAN( float angle );
```

- In Visual Basic,

```
Declare Function FFARCTAN Lib "C:\f90ps\FBAS.DLL" (ByVal angle As Single) As S
```

- In MASM,

```
.MODEL FLAT, STDCALL
FFARCTAN PROTO STDCALL, angle: PTR REAL4
...
FFARCTAN PROC STDCALL, angle: PTR REAL4
```

All-Lowercase Names

If the name of the routine appears as all lowercase in C or MASM, then naming conventions are automatically correct when the C or STDCALL option is used in the Fortran declaration. Any case may be used in the Fortran source code, including mixed case since the C and STDCALL options change the name to all lowercase. In this way STDCALL differs from the Fortran default behavior. You cannot call a Visual Basic routine from Fortran directly (see [Fortran/Visual Basic Mixed-Language Programs](#)), so Basic routine names are never translated.

Mixed-case Names

If the name of a routine appears as mixed-case in C or MASM and you cannot change the name, then you can resolve this naming conflict by using the Fortran **ATTRIBUTES** ALIAS option. ALIAS is required in this situation because otherwise Fortran will not preserve the mixed-case name.

To use the ALIAS option, place the name in quotation marks exactly as it is to appear in the .OBJ file. The following is an example on x86 systems for referring to the C function My_Proc:

```
!DEC$ ATTRIBUTES ALIAS:'_My_Proc' :: My_Proc
```

On Alpha systems, this would be coded without the leading underscore as:

```
!DEC$ ATTRIBUTES ALIAS:'My_Proc' :: My_Proc
```

Fortran Module Names

Fortran module entities (data and procedures) have external names that differ from other external entities. Module names use the convention:

```
_MODULENAME_mp_ENTITY [ @stacksize ]
```

MODULENAME is the name of the module and is all uppercase by default. *ENTITY* is the name of the module procedure or module data contained within *MODULENAME*. *ENTITY* is also uppercase by default. *_mp_* is the separator between the module and entity names and is always lowercase.

For example:

```
MODULE mymod
  INTEGER a
```

```
CONTAINS
  SUBROUTINE b (j)
    INTEGER j
  END SUBROUTINE
END MODULE
```

results in the following symbols being defined in the compiled .OBJ file on x86 systems:

```
_MYMOD_mp_A
_MYMOD_mp_B@4
```

Or, on Alpha systems:

```
MYMOD_mp_A
MYMOD_mp_B
```

Compiler options can affect the naming of module data and procedures.

Note: Except for ALIAS, **ATTRIBUTES** options do not affect the module name, which remains uppercase.

The following table shows how each **ATTRIBUTES** option affects the subroutine in the previous example module.

Effect of ATTRIBUTES options on Fortran Module Names

ATTRIBUTES Option Given to Routine 'b'	Procedure name in .OBJ file on x86 systems	Procedure name in .OBJ file on Alpha systems
None	_MYMOD_mp_B@4	MYMOD_mp_B
C	_MYMOD_mp_b	MYMOD_mp_b
STDCALL	_MYMOD_mp_b@4	MYMOD_mp_b
ALIAS	Overrides all others, name as given in the alias	Overrides all others, name as given in the alias
VARYING	No effect on name	No effect on name

You can write code to call Fortran modules or access module data from other languages. As with other naming and calling conventions, the module name must match between the two languages. Generally, this means using the C or STDCALL convention in Fortran, and if defining a module in another language, using the ALIAS option to match the name within Fortran. Examples are given in the section [Using Modules in Mixed-Language Programming](#).

Visual C++ Names

Visual C++ uses the same calling convention and argument-passing techniques as C, but naming conventions are different because of Visual C++ decoration of external symbols. The **extern "C"** syntax makes it possible for a Visual C++ module to share data and routines with other languages by causing Visual C++ to drop name decoration.

The following example declares prn as an external function using the C naming convention. This declaration appears in Visual C++ source code:

```
extern "C" { void prn(); }
```

To call functions written in Fortran, declare the function as you would in C and use a "C" linkage

specification. For example, to call the Fortran function FACT from Visual C++, declare it as follows:

```
extern "C" { int __stdcall FACT( int n ); }
```

The **extern "C"** syntax can be used to adjust a call from Visual C++ to other languages, or to change the naming convention of Visual C++ routines called from other languages. However, **extern "C"** can only be used from within Visual C++. If the Visual C++ code does not use **extern "C"** and cannot be changed, you can call Visual C++ routines only by determining the name decoration and generating it from the other language. Such an approach should only be used as a last resort, because the decoration scheme is not guaranteed to remain the same between versions.

Use of **extern "C"** has some restrictions:

- You cannot declare a member function with **extern "C"**.
- You can specify **extern "C"** for only one instance of an overloaded function; all other instances of an overloaded function have Visual C++ linkage.

For more information on the **extern "C"** linkage specification, see the *Microsoft Visual C++ Language Reference*.

Prototyping a Procedure in Fortran

You define a prototype (interface block) in your Fortran source code to tell the Fortran compiler which language conventions you want to use for an external reference. The interface block is introduced by the **INTERFACE** statement. See [Program Units and Procedures](#), for a more detailed description of the **INTERFACE** statement.

The general form for the **INTERFACE** statement is:

INTERFACE

routine statement

[*routine* **ATTRIBUTE** *options*]

[*argument* **ATTRIBUTE** *options*]

formal argument declarations

END *routine name*

END INTERFACE

The *routine statement* defines either a **FUNCTION** or a **SUBROUTINE**, where the choice depends on whether a value is returned or not, respectively. The optional *routine* **ATTRIBUTE** *options* (such as C and STDCALL) determine the calling, naming, and argument-passing conventions for the routine in the prototype statement. The optional **ATTRIBUTE** *argument options* (such as VALUE and REFERENCE) are properties attached to individual arguments. The *formal argument declarations* are Fortran data type declarations. Note that the same **INTERFACE** block can specify more than one procedure.

For example, suppose you are calling a C function that has the following prototype:

```
extern void My_Proc (int i);
```

The Fortran call to this function should be declared with the following **INTERFACE** block on x86 systems:

```

INTERFACE
  SUBROUTINE my_Proc (I)
    !DEC$ ATTRIBUTES C, ALIAS: '_My_Proc' :: my_Proc
    INTEGER I
  END SUBROUTINE my_Proc
END INTERFACE

```

Note that:

- On Alpha systems, the leading underscore in `_My_Proc` is omitted. The `!DEC$ ATTRIBUTES` line on Alpha systems contains:

```
!DEC$ ATTRIBUTES C, ALIAS: 'My_Proc' :: my_Proc
```

- Except in the **ALIAS** string, the case of `My_Proc` in the Fortran program doesn't matter.

Exchanging and Accessing Data in Mixed-Language Programming

You can use several approaches to sharing data between mixed-language routines, which can be used within the individual languages as well. These approaches are:

- [Passing Arguments in Mixed-Language Programming](#)
- [Using Modules in Mixed-Language Programming](#)
- [Using Common External Data in Mixed-Language Programming](#)

Generally, if you have a large number of parameters to work with or you have a large variety of parameter types, you should consider using modules or external data declarations. This is true when using any given language, and to an even greater extent when using mixed languages.

Passing Arguments in Mixed-Language Programming

You can pass data between Fortran and C, Visual C++, Visual Basic and MASM through calling argument lists just as you can within each language (for example, the argument list `a, b` and `c` in `CALL MYSUB(a,b,c)`). There are two ways to pass individual arguments:

- *By value*, which passes the argument's value.
- *By reference*, which passes the address of the argument. (In Fortran, Visual Basic, C, and Visual C++, all addresses in Version 5.0 are 4 bytes on x86 and Alpha systems.)

You need to make sure that for every call, the calling program and the called routine agree on how each argument is passed. Otherwise, the called routine receives bad data.

The Fortran technique for passing arguments changes depending on the calling convention specified. By default, Fortran passes all data by reference (except the hidden length argument of strings, which is a special case). If the **ATTRIBUTES C** or **STDCALL** option is used, the default changes to passing all data by value except arrays. If the procedure has the **REFERENCE** option as well as the **C** or **STDCALL** option, all arguments by default are passed by reference.

In Fortran, in addition to establishing argument passing with the calling-convention options C and STDCALL, you can specify argument options, VALUE and REFERENCE, to pass arguments by value or by reference. In mixed-language programming, it is a good idea to specify the passing technique explicitly rather than relying on defaults.

Note: In addition to ATTRIBUTES, the compiler option `/iface` also establishes some default argument passing conventions (such as for hidden length of strings).

Examples of passing by reference and value for C, Visual Basic and MASM follow. All are interfaces to the example Fortran subroutine TESTPROC below. The definition of TESTPROC declares how each argument is passed. The REFERENCE option is not strictly necessary in this example, but using it makes the argument's passing convention conspicuous.

```
SUBROUTINE TESTPROC( VALPARM, REFPARM )
  !DEC$ ATTRIBUTES VALUE :: VALPARM
  !DEC$ ATTRIBUTES REFERENCE :: REFPARM
  INTEGER VALPARM
  INTEGER REFPARM
END
```

- Fortran/C example of arguments passed by value and reference

In C and Visual C++ all arguments are passed by value, except arrays, which are passed by reference to the address of the first member of the array. Unlike Fortran, C and Visual C++ do not have calling-convention directives to affect the way individual arguments are passed. To pass non-array C data by reference, you must pass a pointer to it. To pass a C array by value, you must declare it as a member of a structure and pass the structure. The following C declaration sets up a call to the example Fortran TESTPROC subroutine:

```
extern void __stdcall TESTPROC( int ValParm, int *RefParm );
```

- Fortran/Visual Basic example of arguments passed by value and reference

In Visual Basic, arguments are passed by reference by default. To pass arguments by value, you use the keyword **BYVAL** in front of the argument in the **DECLARE** statement. For example:

```
Declare Sub TESTPROC Lib "C:\f90\TESTPROC.DLL"
    (ByVal Valparm As Long, Refparm As Long)
```

Strings are a special case. See the discussion on character strings in [Handling Character Strings](#).

- Fortran/MASM example of arguments passed by value and reference

In MASM, arguments are passed by value by default. Arguments to be passed by reference are designated with **PTR** in the **PROTO** and **PROC** directives. For example:

```
TESTPROC PROTO STDCALL, valparm: SDWORD, refparm: PTR SDWORD
```

To use an argument passed by value, use the value of the variable. For example:

```
mov eax, valparm ; Load value of argument
```

This statement places the value of `valparm` into the EAX register.

To use an argument passed by reference, use the address of the variable. For example:

```
mov ecx, refparm ; Load address of argument
mov eax, [ecx]  ; Load value of argument
```

These statements place the value of `refparm` into the EAX register.

The following table summarizes how to pass arguments by reference and value. An array name in C is equated to its starting address because arrays are normally passed by reference. You can assign the REFERENCE property to a procedure, not only to individual arguments.

Passing Arguments by Reference and Value

Language	ATTRIBUTE	Argument Type	To pass by reference	To pass by value
Fortran	Default	Scalars and derived types	Default	VALUE option
	C or STDCALL option	Scalars and derived types	REFERENCE option	Default
	Default	Arrays	Default	Cannot pass by value
	C or STDCALL option	Arrays	Default	Cannot pass by value
Visual C/C++		Non-arrays	Pointer <code>argument_name</code>	Default
		Arrays	Default	Struct {type} <code>array_name</code>
Visual Basic		All types	Default	ByVal
Assembler (x86) MASM		All types	PTR	Default

This table does not describe argument passing of strings and Fortran 90 pointer arguments in Visual Fortran, which are constructed differently than other arguments. By default, Fortran passes strings by reference along with the string length. String length placement depends on whether the compiler option `/iface:mixed_str_len_arg` (immediately after the address of the beginning of the string) or `/iface:nomixed_str_len_arg` (after all arguments) is set.

Fortran 90 array pointers and assumed-shape arrays are passed by passing the address of the array descriptor.

For a discussion of the effect of attributes on passing Fortran 90 pointers and strings, see [Handling Fortran 90 Pointers and Allocatable Arrays](#) and [Handling Character Strings](#).

Using Modules in Mixed-Language Programming

Modules are the simplest way to exchange large groups of variables with C, because Visual Fortran modules are directly accessible from Visual C/C++. The following example declares a module in Fortran, then accesses its data from C:

```
! F90 Module definition
```

```

MODULE EXAMP
  REAL A(3)
  INTEGER I1, I2
  CHARACTER(80) LINE
  TYPE MYDATA
    INTEGER N
    CHARACTER(30) INFO
  END TYPE MYDATA
END MODULE EXAMP

\* C code accessing module data *\
extern float EXAMP_mp_A[3];
extern int EXAMP_mp_I1, EXAMP_mp_I2;
extern char EXAMP_mp_LINE[80];
extern struct {
    int N;
    char INFO[30];
} EXAMP_mp_MYDATA;

```

You can also define a module procedure in C and make that routine part of a Fortran module by using the `ALIAS` directive:

```

// C procedure
void pythagoras (float a, float b, float *c)
{
    *c = (float) sqrt(a*a + b*b);
}
! Fortran 90 Module including procedure
MODULE CPROC
  INTERFACE
    SUBROUTINE PYTHAGORAS (a, b, res)
      !DEC$ ATTRIBUTES C :: PYTHAGORAS
      !DEC$ ATTRIBUTES REFERENCE :: res
! res is passed by REFERENCE because its individual attribute
! overrides the subroutine's C attribute
      REAL a, b, res
! a and b have the VALUE attribute by default because
! the subroutine has the C attribute
    END SUBROUTINE
  END INTERFACE
END MODULE

```

Using Common External Data in Mixed-Language Programming

Common external data structures include Fortran common blocks, and C structures and variables that have been declared global or external. All of these data specifications create external variables, which are variables available to routines outside the routine that defines them.

This section applies only to Fortran/C and Fortran/MASM mixed-language programs because there is no way to share common data with Visual Basic. You must pass all data between Visual Basic and Fortran as arguments. This process can be streamlined by passing user-defined types between them, described in [Handling User-Defined Types](#).

External variables are case sensitive, so the cases must be matched between different languages, as discussed in the section on naming conventions. Common external data exchange is described in the following sections:

- [Using Global Variables](#)

- Using Fortran Common Blocks and C Structures

Using Global Variables in Mixed-Language Programming

A variable can be shared between Fortran and C or MASM by declaring it as global (or **COMMON**) in one language and accessing it as an external variable in the other language. Visual Basic cannot access another language's global data or share its own. In Fortran/Basic programs, variables must be passed as arguments.

In Fortran, a variable can access a global parameter by using the **EXTERN** option for **ATTRIBUTES**. For example:

```
!DEC$ ATTRIBUTES C, EXTERN :: idata
INTEGER idata (20)
```

EXTERN tells the compiler that the variable is actually defined and declared global in another source file. If Fortran declares a variable external with **EXTERN**, the language it shares the variable with must declare the variable global.

In C, a variable is declared global with the statement:

```
int idata[20]; // declared as global (outside of any function)
```

MASM declares a parameter global (**PUBLIC**) with the syntax:

```
PUBLIC [langtype] name
```

where *name* is the name of the global variable to be referenced, and the optional *langtype* is **STDCALL** or **C**. The option *langtype*, if present, overrides the calling convention specified in the **.MODEL** directive.

Conversely, Fortran can declare the variable global (**COMMON**) and other languages can reference it as external:

```
!Fortran declaring PI global
REAL PI
COMMON /PI/ PI ! Common Block and variable have the same name
```

In C, the variable is referenced as an external with the statement:

```
//C code with external reference to PI
extern float PI;
```

Note that the global name **C** references is the name of the Fortran common block, not the name of a variable within a common block. Thus, you cannot use blank common to make data accessible between C and Fortran. In the preceding example, the common block and the variable have the same name, which helps keep track of the variable between the two languages. Obviously, if a common block contains more than one variable they cannot all have the common block name. (See common block usage.)

MASM can also access Fortran global (**COMMON**) parameters with the **EXTERN** directive. The syntax is:

```
EXTERN [langtype] name
```

where *name* is the name of the global variable to be referenced, and the optional *langtype* is STDCALL or C.

Using Fortran Common Blocks and C Structures

In order to reference C structures from Fortran common blocks and vice versa, you must take into account the way the common blocks and structures differ in their methods of storing member variables in memory. Fortran places common block variables into memory in order as close together as possible, with the following rules:

- A single BYTE, INTEGER(1), LOGICAL(1), or CHARACTER variable in common block list begins immediately following the previous variable or array in memory.
- All other types of single variables begin at the next even address immediately following the previous variable or array in memory.
- All arrays of variables begin on the next even address immediately following the previous variable or array in memory, except for CHARACTER arrays which always follow immediately after the previous variable or array.
- All common blocks begin on a four-byte aligned address.

Because of these padding rules, you must consider the alignment of C structure elements with Fortran common block elements and assure matching either by making all variables exactly equivalent types and kinds in both languages (using only 4-byte and 8-byte data types in both languages simplifies this) or by using the C pack pragmas in the the C code around the C structure to make C data packing like Fortran's. For example:

```
#pragma pack(2)
struct {
    int N;
    char INFO[30];
} examp;
#pragma pack()
```

To restore the original packing, you must add `#pragma pack()` at the end of the structure. (Remember: Fortran module data can be shared directly with C structures with appropriate naming.)

Once you have dealt with alignment and padding, you can give C access to an entire common block or set of common blocks. Alternatively, you can pass individual members of a Fortran common block in an argument list, just as you can any other data item. Use of common blocks for mixed-language data exchange is discussed in the following sections:

- [Accessing Common Blocks and C Structures Directly](#)
- [Passing the Address of a Common Block](#)

Accessing Common Blocks and C Structures Directly

You can access Fortran common blocks directly from C by defining an external C structure with the appropriate fields, and making sure that alignment and padding between Fortran and C are compatible. The C and ALIAS ATTRIBUTES options can be used with a common block to allow mixed-case names.

As an example, suppose your Fortran code has a common block named `Really`, as shown:

```
!DEC$ ATTRIBUTES ALIAS:'Really' :: Really
```

```

REAL(4) x, y, z(6)
REAL(8) ydbl
COMMON / Really / x, y, z(6), ydbl

```

You can access this data structure from your C code with the following external data structure:

```

#pragma pack(2)
extern struct {
    float x, y, z[6];
    double ydbl;
} Really;
#pragma pack()

```

You can also access C structures from Fortran by creating common blocks that correspond to those structures. This is the reverse case from that just described. However, the implementation is the same because after common blocks and structures have been defined and given a common address (name), and assuming the alignment in memory has been dealt with, both languages share the same memory locations for the variables.

Passing the Address of a Common Block

To pass the address of a common block, simply pass the address of the first variable in the block, that is, pass the first variable by reference. The receiving C or Visual C++ module should expect to receive a structure by reference.

In the following example, the C function `initcb` receives the address of a user-defined type `n`, which it considers to be a pointer to a structure with three fields:

```

! Fortran SOURCE CODE
!
INTERFACE
  SUBROUTINE initcb (BLOCK)
    !DEC$ ATTRIBUTES C :: initcb
    !DEC$ ATTRIBUTES REFERENCE :: BLOCK
    INTEGER BLOCK
  END SUBROUTINE
END INTERFACE
!
INTEGER n
REAL(8) x, y
COMMON /CBLOCK/n, x, y
.
.
.
CALL initcb( n )
/* C source code */
//
#pragma pack(2)
struct block_type
{
    int n;
    double x;
    double y;
};
#pragma pack()
//
void initcb( struct block_type *block_hed )
{
    block_hed->n = 1;
    block_hed->x = 10.0;
    block_hed->y = 20.0;
}

```


Handling Data Types in Mixed-Language Programming

Even when you have reconciled calling conventions, naming conventions, and methods of data exchange, you must still be concerned with data types, because each language handles them differently. The following table lists the equivalent data types among Fortran, C, Visual Basic, and MASM:

Equivalent Data Types

Fortran data type	C data type	Visual Basic data type	MASM data type
INTEGER(1)	char	---	SBYTE
INTEGER(2)	short	Integer	SWORD
INTEGER(4)	int, long	Long	SDWORD
REAL(4)	float	Single	REAL4
REAL(8)	double	Double	REAL8
CHARACTER(1)	unsigned char	---	BYTE
CHARACTER*(*)	See Handling Character Strings		
COMPLEX(4)	struct complex4 { float real, imag; };	---	COMPLEX4 STRUCT 4 real REAL4 0 imag REAL4 0 COMPLEX4 ENDS
COMPLEX(8)	struct complex8 { double real, imag; };	---	COMPLEX8 STRUCT 8 real REAL8 0 imag REAL8 0 COMPLEX8 ENDS
All LOGICAL types	Use integer types for C, MASM, and Visual Basic		

The following sections describe how to reconcile data types between the different languages:

- [Handling Numeric, Complex, and Logical Data Types](#)
- [Handling Fortran 90 Array Pointers and Allocatable Arrays](#)
- [Handling DIGITAL Fortran Pointers](#)
- [Handling Arrays and Visual Fortran Array Descriptors](#)
- [Handling Character Strings](#)
- [Handling User-Defined Types](#)

Handling Numeric, Complex, and Logical Data Types

Normally, passing numeric data does not present a problem. If a C program passes an unsigned data type to a Fortran routine, the routine can accept the argument as the equivalent signed data type, but you should be careful that the range of the signed type is not exceeded.

The table of [Equivalent Data Types](#) (included in the section Handling Data Types in Mixed-Language Programming) summarizes equivalent numeric data types for Fortran, MASM, and Visual Visual C/C++.

C, Visual C++, and MASM do not directly implement the Fortran types COMPLEX(4) and

COMPLEX(8). However, you can write structures that are equivalent. The type COMPLEX(4) has two fields, both of which are 4-byte floating-point numbers; the first contains the real-number component, and the second contains the imaginary-number component. The type COMPLEX is equivalent to the type COMPLEX(4). The type COMPLEX(8) is similar except that each field contains an 8-byte floating-point number.

Note: Fortran functions of type COMPLEX place a hidden COMPLEX argument at the beginning of the argument list. C functions that implement such a call from Fortran must declare this hidden argument explicitly, and use it to return a value. The C return type should be **void**.

Following are the Visual C/C++ structure definitions for the Fortran COMPLEX types:

```
struct complex4 {
    float real, imag;
};
struct complex8 {
    double real, imag;
};
```

The following list contains the MASM structure definitions for the Fortran COMPLEX types:

```
COMPLEX4 STRUCT 4
    real REAL4 0
    imag REAL4 0
COMPLEX4 ENDS
COMPLEX8 STRUCT 8
    real REAL8 0
    imag REAL8 0
COMPLEX8 ENDS
```

A Fortran LOGICAL(2) is stored as a 2-byte indicator value (0=false, and the [/fpscomp:\[no\]logicals](#) compiler option determines how true values are handled). A Fortran LOGICAL(4) is stored as a 4-byte indicator value, and LOGICAL(1) is stored as a single byte. The type LOGICAL is the same as LOGICAL(4), which is equivalent to type int in C.

You can use a variable of type LOGICAL in an argument list, module, common block, or global variable in Fortran and type int in C for the same argument. Type LOGICAL(4) is recommended instead of the shorter variants for use in common blocks.

The Visual C++ class type has the same layout as the corresponding C struct type, unless the class defines virtual functions or has base classes. Classes that lack those features can be passed in the same way as C structures.

Handling Fortran 90 Array Pointers and Allocatable Arrays

How Fortran 90 array pointers and arrays are passed is affected by the **ATTRIBUTES** options in effect, and by the **INTERFACE**, if any, of the procedure they are passed to. If the **INTERFACE** declares the array pointer or array with deferred shape (for example, ARRAY(:)), its descriptor is passed. This is true for array pointers and all arrays, not just allocatable arrays. If the **INTERFACE** declares the array pointer or array with fixed shape, or if there is no interface, the array pointer or array is passed by base address, which is like passing the first element of an array.

When a Fortran 90 array pointer or array is passed to another language, either its descriptor or its

base address can be passed. The following table shows how Fortran 90 array pointers and allocatable arrays are passed with different attributes in effect. Note that the VALUE option cannot be used with descriptor-based arrays.

The effect of ATTRIBUTES options on Fortran 90 array pointers and allocatable arrays passed as arguments is as follows:

- If the property of the array pointer or array is **none**, it is passed by descriptor, regardless of the property of the passing procedure (None; C; STDCALL; C, REFERENCE; or STDCALL, REFERENCE).
- If the property of the array pointer or array is **VALUE**, an error is returned, regardless of the property of the passing procedure.
- If the property of the array pointer or array is **REFERENCE**, it is passed by descriptor, regardless of the property of the passing procedure.

When you pass a Fortran array pointer or an array by descriptor to a non-Fortran routine, that routine needs to know how to interpret the descriptor. Part of the descriptor is a pointer to address space, as a C pointer, and part of it is a description of the pointer or array properties, such as its rank, stride, and bounds. For information about the Visual Fortran array descriptor format, see [Handling Arrays and Visual Fortran Array Descriptors](#)

Fortran 90 pointers that point to scalar data contain the address of the data and are not passed by descriptor.

Handling DIGITAL Fortran Pointers

DIGITAL Fortran (integer) pointers are not the same as Fortran 90 pointers, but are instead like C pointers. DIGITAL Fortran pointers are 4-byte INTEGER quantities.

When passing a DIGITAL Fortran pointer to a routine written in another language:

- The argument should be declared in the non-Fortran routine as a pointer of the appropriate data type.
- The argument passed from the Fortran routine should be the DIGITAL Fortran pointer name, not the pointer-based variable name.

For example, on x86 systems:

```
! 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 int.
  p = LOC(A)

  CALL Ptr_Sub (p)
  WRITE(*,*) 'A(4) = ', A(4)
  END
!
```

```
//C subprogram
void Ptr_Sub (float *p)
{
    p[3] = 23.5;
}
```

On Alpha systems, the `!DEC$ ATTRIBUTES` line omits the leading underscore for `_Ptr_Sub`, as follows:

```
!DEC$ ATTRIBUTES C, ALIAS:'Ptr_Sub' :: Ptr_Sub
```

When the main Fortran program and C function are built and executed, the following output appears:

```
A(4) = 23.50000
```

When receiving a pointer from a routine written in another language:

- The argument should be declared in the non-Fortran routine as a pointer of the appropriate data type and passed as usual.
- The argument received by the Fortran routine should be declared as a DIGITAL Fortran pointer name, then the **POINTER** statement should associate it with a pointer-based variable of the appropriate data type (matching the data type of the passing routine). When inside the Fortran routine, use the pointer-based variable to set and access what the pointer points to.

For example, on x86 systems:

```
! Fortran subroutine.
SUBROUTINE Iptr_Sub (p)
!DEC$ ATTRIBUTES C, ALIAS:'_Iptr_Sub' :: Iptr_Sub
    integer VAR(10)
    POINTER (p, VAR)
    OPEN (8, FILE='STAT.DAT')
    READ (8, *) VAR(4) ! Read from file and store the
                        ! fourth element of VAR
END SUBROUTINE Iptr_Sub
!
//C main program
extern void Iptr_Sub(int *p);

main ( void )
{
    int a[10];
    Iptr_Sub (&a[0]);
    printf("a[3] = %i\n", a[3]);
}
```

On Alpha systems, the `!DEC$ ATTRIBUTES` line omits the leading underscore, as follows:

```
!DEC$ ATTRIBUTES C, ALIAS:'Iptr_Sub' :: Iptr_Sub
```

When the main C program and Fortran subroutine are built and executed, the following output appears:

```
a[3] = 4
```

Handling Arrays and Visual Fortran Array Descriptors

Fortran 90 allows arrays to be passed as array elements, as array subsections, or as whole arrays referenced by array name. Within Fortran 90, array elements are ordered in column-major order, meaning the subscripts of the lowest dimensions vary first.

When using arrays between Fortran and another language, differences in element indexing and ordering must be taken into account. You must reference the array elements individually and keep track of them. Fortran, Visual Basic, MASM and C vary in the way that array elements are indexed. Array indexing is a source-level consideration and involves no difference in the underlying data.

Visual Basic stores arrays and character strings as descriptors: data structures that contain array size and location. This storage difference is transparent to the user, however.

To pass an array from Visual Basic to Fortran, pass the first element of the array. By default, Visual Basic passes variables by reference, so passing the first element of the array will give Fortran the starting location of the array, just as Fortran expects. Visual Basic indexes the first array element as 0 by default, while Fortran by default indexes it as 1. Visual Basic indexing can be set to start with 1 using the statement:

```
Option Base 1
```

Alternatively, in the array declaration in either language you can set the array lower bound to any integer in the range -32,768 to 32,767. For example:

```
' In Basic
Declare Sub FORTARRAY Lib "fortarr.dll" (Barray as Single)
DIM barray (1 to 3, 1 to 7) As Single
Call FORTARRAY(barray (1,1))

! In Fortran
Subroutine FORTARRAY(arr)
  REAL arr(3,7)
```

In MASM, arrays are one-dimensional and array elements must be referenced byte-by-byte. The assembler stores elements of the array consecutively in memory, with the first address referenced by the array name. You then access each element relative to the first, skipping the total number of bytes of the previous elements. For example:

```
xarray      REAL4      1.1, 2.2, 3.3, 4.4 ; initializes
                                   ; a four element array with
                                   ; each element 4 bytes
```

Referencing xarray in MASM refers to the first element, the element containing 1.1. To refer to the second element, you must refer to the element 4 bytes beyond the first with xarray[4] or xarray+4. Similarly:

```
yarray      BYTE       256 DUP      ; establishes a
                                   ; 256 byte buffer, no initialization
zarray      SWORD 100  DUP(0) ; establishes 100
                                   ; two-byte elements, initialized to 0
```

Fortran and C arrays differ in two ways:

- The value of the lower array bound is different. By default, Fortran indexes the first element of an array as 1. C and Visual C++ index it as 0. Fortran subscripts should therefore be one higher. (Fortran also provides the option of specifying another integer lower bound.)
- In arrays of more than one dimension, Fortran varies the left-most index the fastest, while C varies the right-most index the fastest. These are sometimes called column-major order and row-major order, respectively.

In C, the first four elements of an array declared as `X[3][3]` are:

```
x[0][0] x[0][1] x[0][2] x[1][0]
```

In Fortran, the first four elements are:

```
x(1,1) x(2,1) x(3,1) x(1,2)
```

The order of indexing extends to any number of dimensions you declare. For example, the C declaration:

```
int arr1[2][10][15][20];
```

is equivalent to the Fortran declaration:

```
INTEGER arr1( 20, 15, 10, 2 )
```

The constants used in a C array declaration represent extents, not upper bounds as they do in other languages. Therefore, the last element in the C array declared as `int arr[5][5]` is `arr[4][4]`, not `arr[5][5]`.

The following table shows equivalencies for array declarations.

Equivalent Array Declarations for Different Languages

Language	Array declaration	Array reference from Fortran
Fortran	DIMENSION x(i, k) -or- <i>type</i> x(i, k)	x(i, k)
Visual Basic	DIM x(i, k) As <i>type</i>	x(i -1, k -1)
Visual C/C++	<i>type</i> x[k] [i]	x(i -1, k -1)
MASM	Declare and reference arrays as elements in consecutive storage	

Visual Fortran Array Descriptor Format

For cases where Fortran 90 needs to keep track of more than a pointer memory address, the DIGITAL Visual Fortran compiler uses an **array descriptor**, which stores the details of how an array is organized.

When using an explicit interface (by association or procedure interface block), Visual Fortran will generate a descriptor for the following types of array arguments:

- Pointers to arrays (array pointers)
- Assumed-shape arrays

Certain data structure arguments do not use a descriptor, even when an appropriate explicit interface

is provided. For example, explicit-shape and assumed-size arrays do not use a descriptor. In contrast, array pointers and allocatable arrays use descriptors regardless of whether they are used as arguments.

When calling between Visual Fortran and a non-Fortran language (such as C), using an *implicit* interface allows the array argument to be passed *without* a Visual Fortran descriptor.

However, for cases where the called routine needs the information in the Visual Fortran descriptor, declare the routine with an *explicit* interface and specify the dummy array as either an assumed-shape array or with the pointer attribute.

You can associate a Fortran 90 pointer with any piece of memory, organized in any way desired (so long as it is "rectangular" in terms of array bounds). You can also pass Fortran 90 pointers to other languages, such as C, and have the other language correctly interpret the descriptor to obtain the information it needs.

However, using array descriptors can increase the opportunity for errors and is not portable:

- If the descriptor is not defined correctly, the program might access the wrong memory address, possibly causing a General Protection Fault.
- Array descriptor formats are specific to each Fortran compiler. Code that uses array descriptors is **not** portable to other compilers or platforms. For example, the Visual Fortran array descriptor format (for Win32 systems) differs from the array descriptor format for DIGITAL Fortran on DIGITAL UNIX and OpenVMS systems. The Visual Fortran array descriptor format is the same format used by Microsoft Fortran Powerstation.
- The array descriptor format might change in the future.

The components of the Visual Fortran array descriptor follow:

- The first longword (bytes 0 to 3) contains the base address. The base address plus the offset defines the first memory location (start) of the array.
- The second longword (bytes 4 to 7) contains the size of a single element of the array.
- The third longword (bytes 8 to 11) contains the offset. The offset is added to the base address to define the start of the array.
- The fourth longword (bytes 12 to 15) contains the low-order bit set if the array has been defined (storage allocated).
- The fifth longword (bytes 16 to 19) contains the number of dimensions (rank) of the array.
- The remaining longwords (bytes 20 up to 103) contain information about each dimension (up to seven). Each dimension is described by three additional longwords:
 - The number of elements (extent)
 - The distance between the starting address of two successive elements, in bytes. This value is the stride of the array expression multiplied by the size of one array element.
 - The lower bound

An array of rank one would require three additional longwords for a total of in eight longwords ($5 + 3*1$) and end at byte 31. An array of rank seven would be described in a total of 26 longwords ($5 + 3*7$) and end at byte 103.

For example, consider the following declaration:

```
integer,target :: a(10,10)
integer,pointer :: p(:, :)
```

```
p => a(9:1:-2,1:9:3)
call f(p)
.
.
.
```

The descriptor for actual argument p would contain the following values:

- The first longword (bytes 0 to 3) contain the base address (assigned at run-time).
- The second longword (bytes 4 to 7) is set to 4 (size of a single element).
- The third longword (bytes 8 to 11) contain the offset (assigned at run-time).
- The fourth longword (bytes 12 to 15) contains 1 (low bit is set).
- The fifth longword (bytes 16 to 19) contains 2 (rank).
- The sixth, seventh, and eighth longwords (bytes 20 to 31) contain information for the first dimension, as follows:
 - 5 (extent)
 - -8 (distance between elements)
 - 1 (the lower bound)
- For the second dimension, the ninth, tenth, and eleventh longwords (bytes 32 to 43) contain
 - 3 (extent)
 - 120 (distance between elements)
 - 1 (the lower bound)
- Byte 43 is the last byte for this example.

Handling Character Strings

By default, Visual Fortran passes a hidden length argument for strings. The hidden length argument consists of an unsigned 4-byte integer, always passed by value, immediately following the address of the character string. You can alter the default way strings are passed by using attributes. The following table shows the effect of various attributes on passed strings.

Effect of ATTRIBUTES Options on Character Strings Passed as Arguments

Argument	Default	C	STDCALL	C, REFERENCE	STDCALL, REFERENCE
String	Passed by reference, along with length	First character converted to INTEGER(4) and passed by value	First character converted to INTEGER(4) and passed by value	Passed by reference, along with length	Passed by reference, no length
String with VALUE option	Error	First character converted to INTEGER(4) and passed by value	First character converted to INTEGER(4) and passed by value	First character converted to INTEGER(4) and passed by value	First character converted to INTEGER(4) and passed by value
String with REFERENCE option	Passed by reference, possibly along with	Passed by reference, no length	Passed by reference, no length	Passed by reference, no length	Passed by reference, no length

	length			
--	--------	--	--	--

The important things to note about the above table are:

- Character strings without the VALUE or REFERENCE attribute that are passed to C or STDCALL routines are not passed by reference. Instead, only the first character is passed and it is passed by value.
- Character strings with the VALUE option passed to C or STDCALL routines are not passed by pushing the entire string on the stack. Instead, only the value of the first character is passed.
- For string arguments with **ATTRIBUTES DEFAULT** or C, REFERENCE:
 - When `/iface:mixed str len arg` is set, the length of the string is pushed (by value) on the stack immediately after the address of the beginning of the string.
 - When `/iface:nomixed str len arg` is set, the length of the string is pushed (by value) on the stack after all of the other arguments.
- For string arguments passed by reference with default ATTRIBUTES:
 - When `/iface:mixed str len arg` is set, the length of the string is pushed (by value) on the stack immediately after the address of the beginning of the string.
 - When `/iface:nomixed str len arg` is set, the length of the string is not available to the called procedure.

Since all strings in C are pointers, C expects strings to be passed by reference, without a string length. In addition, C strings are null-terminated while Fortran strings are not. There are two basic ways to pass strings between Fortran and C: convert Fortran strings to C strings, or write C routines to accept Fortran strings.

To convert a Fortran string to C, choose a combination of attributes that passes the string by reference without length, and null terminate your strings. For example, on x86 systems:

```

INTERFACE
  SUBROUTINE Pass_Str (string)
    !DEC$ ATTRIBUTES C, ALIAS: '_Pass_Str' :: Pass_Str
    CHARACTER(*) string
    !DEC$ ATTRIBUTES REFERENCE, string
  END SUBROUTINE
END INTERFACE
CHARACTER(40) forstring
DATA forstring /'This is a null-terminated string.'C/

```

On Alpha systems, the first `!DEC$ ATTRIBUTES` line would omit the leading underscore and be as follows:

```
!DEC$ ATTRIBUTES C, ALIAS: '_Pass_Str' :: Pass_Str
```

This example shows the extension of using the null-terminator for the string in the Fortran DATA statement (see [C Strings](#)):

```
DATA forstring /'This is a null-terminated string.'C/
```

The C interface is:

```
void Pass_Str (char *string)
```

To get your C routines to accept Fortran strings, C must account for the length argument passed

along with the string address. For example:

```
Fortran code
INTERFACE
SUBROUTINE Pass_Str (string)
CHARACTER*(*) string
END INTERFACE
```

The C routine must expect two arguments:

```
void __stdcall PASS_STR (char *string, unsigned int length_arg )
```

This interface handles the hidden-length argument, but you must still reconcile C strings that are null-terminated and Fortran strings that are not. In addition, if the data assigned to the Fortran string is less than the declared length, the Fortran string will be blank padded.

Rather than trying to handle these string differences in your C routines, the best approach in Fortran/C mixed programming is to adopt C string behavior whenever possible. Another good reason for using C strings is that Win32 APIs and most C library functions expect null-terminated strings.

Fortran functions that return a character string using the syntax CHARACTER*(*) place a hidden string argument and the address of the string at the beginning of the argument list.

C functions that implement such a Fortran function call must declare this hidden string argument explicitly and use it to return a value. The C return type should be void. However, you are more likely to avoid errors by not using character-string return functions. Use subroutines or place the strings into modules or global variables whenever possible.

Visual Basic strings must be passed by value to Fortran. Visual Basic strings are actually stored as structures containing length and location information. Passing by value dereferences the structure and passes just the string location, as Fortran expects. For example:

```
! In Basic
Declare Sub forstr Lib "forstr.dll" (ByVal Bstring as String)
DIM bstring As String * 40 Fixed-length string
CALL forstr(bstring)

! In Fortran
SUBROUTINE forstr(s)
!DEC$ ATTRIBUTES STDCALL:: forstr
CHARACTER(40) s
s = 'Hello, Visual Basic!'
END
```

The Fortran directive `!DEC$ ATTRIBUTES STDCALL` informs Fortran not to expect the hidden length arguments to be passed from the Visual Basic calling program. The name in the Visual Basic program is specified as lowercase since `STDCALL` makes the Fortran name lowercase.

MASM does not add either a string length or a null character to strings by default. To append the string length, use the syntax:

```
lenstring BYTE "String with length", LENGTHOF lenstring
```

To add a null character, append it by hand to the string:

```
nullstring BYTE "Null-terminated string", 0
```

Handling User-Defined Types

Fortran 90 supports user-defined types (data structures similar to C structures). User-defined types can be passed in modules and common blocks just as other data types, but the other language must know the type's structure. For example:

```
! Fortran CODE
  TYPE LOTTA_DATA
    SEQUENCE
    REAL A
    INTEGER B
    CHARACTER(30) INFO
    COMPLEX CX
    CHARACTER(80) MOREINFO
  END TYPE LOTTA_DATA
  TYPE (LOTTA_DATA) D1, D2
  COMMON /T_BLOCK/ D1, D2

/* C code accessing D1 and D2 */
extern struct {
  struct {
    float a;
    int b;
    char info[30];
    struct {
      float real, imag;
    } cx;
    char moreinfo[80];
  } d1, d2;
} T_BLOCK;
```

Visual Fortran/Visual C++ Mixed-Language Programs

When you understand and reconcile the calling, naming and argument passing conventions between Fortran and C, you are ready to build an application.

If you are using Visual C/C++ you can edit, compile and debug your code within Microsoft Developer Studio. If you are using another C compiler, you can edit your code within Microsoft Developer Studio by selecting File/New and choosing Visual C/C++ source in the File tab or, after activating the editor, by selecting the View menu Properties item and selecting from the drop-down list.

However, if you are not using Visual C/C++, you must compile your code outside Microsoft Developer Studio and either build the Fortran/C program on the command line or add the compiled C .OBJ file to your Fortran project in Microsoft Developer Studio.

As an example of building from the command line, if you have a main C program CMAIN.C that calls Fortran subroutines contained in FORSUBS.F90, you can create the CMAIN application with the following commands:

```
cl /c cmain.c
DF cmain.obj forsubs.f90
```

The Fortran (DF) compiler accepts an object file for the main program written in C and compiled by the C compiler. The DF compiler compiles the .F90 file and then has the linker create an executable

file under the name CMAIN.EXE using the two object files.

Either compiler can do the linking, regardless of which language the main program is written in; however, if you use the DF compiler first, you must include DFOR.LIB with the C compiler, and you might experience some difficulty with the version of LIBC.LIB used by the C compiler. For these reasons, you may prefer to use the C compiler first or get your project settings for both Fortran and C to agree on the default C library to link against.

You need to link your application against one and only one copy of the C library.

When using Developer Studio to build your application, Fortran uses default libraries depending on the information specified in the Fortran tab in the Project menu, Settings item (Project Settings dialog box). You can also specify linker settings with the Linker tab in the Project Settings dialog box.

In the Fortran tab, within the Libraries category, the following options determine the default libraries selected:

- Use Fortran Run-time Libraries (Static or DLL) (Types of Projects) of Dynamic-Link Library or the DF command option `/dll`
- Use Multi-threaded Libraries (`/[no]threads`)
- Use C Debug Libraries (`/[no]dbglibs`)

The combinations of these options use the following libraries:

Static or DLL Project?	Use Multi-Theaded Libraries?	Use C Debug Libraries?	Fortran Link Library Used	C Link Library Used
Static	No	No	dfor.lib	libc.lib
Static	No	Yes	dfor.lib	libcd.lib
Static	Yes	No	dformat.lib	libcmt.lib
Static	Yes	Yes	dformat.lib	libcmt.d.lib
DLL	No	No	dfordll.lib (dforrt.dll)	msvcrt.lib (msvcrt.dll)
DLL	No	Yes	dfordll.lib (dforrt.dll)	msvcrt.d.lib (msvcrt.d.dll)
DLL	Yes	No	dformd.lib (dformd.dll)	msvcrt.lib (msvcrt.dll)
DLL	Yes	Yes	dformd.lib (dformd.dll)	msvcrt.d.lib (msvcrt.d.dll)

The way Visual C++ chooses libraries is also based upon the Project menu Settings item, but within the C/C++ tab. In the Code Generation category, the Use run-time library item lists the following C libraries:

Menu Item Selected	CL Option or Project Type Enabled	Default Library Specified in Object File
Single-threaded	/ML	libc.lib
Multithreaded	/MT	libcmt.lib
Multithreaded DLL	/MD	msvcrt.lib (msvcrt.dll)

Debug Single-threaded	/MLd	libcd.lib
Debug Multithreaded	/MTd	libcmt.lib
Debug Multithreaded DLL	/MDd	msvcrt.lib (msvcrt.dll)

If you are using Microsoft Visual C/C++, Microsoft Developer Studio can create mixed Fortran/C applications transparently, with no special directives or steps on your part. You can edit and browse your C and Fortran programs with appropriate syntax coloring for the language. You can add C source files to your Fortran project or Fortran source files to a C project, and they will be compiled and linked automatically.

When you debug a mixed Visual C/Fortran application, the debugger will adjust to the code type as it steps through: the C or Fortran expression evaluator will be selected automatically based on the code being debugged, and the stack window will show Fortran data types for Fortran procedures and C data types for C procedures.

When printing from Visual C++ programs while calling Fortran subprograms that also print, the output may not appear in the order you expect. In Visual C++, the output buffer contents are not written immediately, but written when the buffer is full, the I/O stream is closed or the program terminates normally. The buffer is said to be "flushed" when this occurs.

To make sure interleaving Visual C++ and Fortran program units print in the order expected, you can explicitly flush the Visual C++ buffers after an output command with the **flushall**, **fflush**, **fclose**, **setbuf**, or **setvbuf** Visual C++ library calls.

Multithreaded applications should have full multithread support, so if you use DFORMAT.LIB, be sure LIBCMT.LIB is specified as a default library.

Fortran/Visual Basic Mixed-Language Programs

Fortran/Visual Basic programs allow you to use the user-interface features of Visual Basic and do computation in Fortran. In Fortran/Visual Basic programs, the Visual Basic must be 32-bit (at least Version 4.0). To use a common Developer Studio environment (instead of the command line environment), use Visual Basic Version 5.0.

You can call Fortran subprograms from Visual Basic, but because Visual Basic subprograms are interpreted and not compiled, they cannot be called directly from compiled language programs like Fortran. Instead, Visual Basic creates OLE objects that export properties and routines.

Visual Basic calls to Fortran are discussed in the next sections:

- [Visual Basic User Interfaces for Fortran](#)
- [Examples of Fortran/Visual Basic Programs](#)

Visual Basic User Interfaces for Fortran

You can create user interfaces to your Fortran routines in Visual Basic by adding items (such as command buttons) to the Form and attaching to them the Visual Basic code that calls the Fortran subroutines. The Visual Basic call and the Fortran subroutine must match their calling and naming

conventions as discussed in the previous sections.

The following example steps through creating a simple Visual Basic interface to a Fortran subroutine that exchanges an array of integers between them.

► **To create the Visual Basic interface:**

1. Select File/New Project in Visual Basic. Add a Command Button and Text Box by dragging them from the Tool Bar to the Form. Change the caption of the Command Button by clicking on it, then selecting Caption in Windows/Properties.
2. Double-click on the Command Button. The code window for that control opens. Add the following Visual Basic code:

```
Static arr(1 To 3, 1 To 7) As Single
    Call ARRAYTEST(arr(1, 1))
    text1.Text = arr(3, 7)
```

3. Choose Project/ADD Module. Add the following code to the General Declarations window:

```
Type MyArray
    arr(1 To 3, 1 To 7) As Single
    Declare Sub ARRAYTEST Lib "d:\vb\f90vb.dll" (Myarray As Single)
```

4. Save the module as GLOBAL.BAS.

The following example defines a Visual Basic type, an array of strings. The Fortran DLL that contains the subroutine should be declared with its full path in the module, so Visual Basic doesn't have to search for it.

► **To create the Fortran subroutine for Visual Basic to call:**

1. In Visual Fortran:
 - From the File menu select New.
 - Click the Projects tab and select "Win32 Dynamic-Link Library."
 - Enter "F90VB" as the project name.
 - Click OK.
2. Perform the following steps:
 - From the File menu select New.
 - Click the Files tab and select "Fortran Free Format Source Files"
 - Enter "f90vb" as the file name.
 - Click OK.
3. Within the text editor, type the following code:

```
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
```

From the File menu, select Save.

4. Build the DLL and copy it to the Visual Basic directory you specified in your Visual Basic module, d:\vb in the example. To build the Fortran DLL from the command line, use the following command:

```
df /dll f90vb.f90
```

5. Run the Visual Basic project by selecting Run/Start. The Visual Basic interface you created appears. Start the Fortran subroutine by choosing the Command Button.

Examples of Fortran/Visual Basic Programs

The following brief code demonstrates the interface for a Fortran subroutine and function (free-form Fortran source):

1. In Developer Studio, create a new project of type *Win32 Dynamic-Link Library*. Name the project FCALL.
2. Create a new free-form source file (Project menu, Add to Project, New) for the project named FCALL.F90 with the following code:

```
! Fortran Code establishing subroutine
! Computes the MOD of R1 and 256.0 and stores the
! result in the argument NUM

SUBROUTINE FortranCall (r1, num)

! Specify that the routine name is to be made available to callers of the
! DLL and that the external name should not have any prefix or suffix

!DEC$ ATTRIBUTES DLLEXPORT :: FortranCall
!DEC$ ATTRIBUTES ALIAS:'FortranCall' :: FortranCall

REAL, INTENT(IN) :: r1          ! Input argument
REAL, INTENT(OUT) :: num

num = MOD (r1, 256.0)

END SUBROUTINE
```

3. Build the Fortran DLL as described in [Dynamic-Link Library Projects in Building Programs and Libraries](#).
4. Start Visual Basic and create a new *Standard EXE* project:
 - On the control toolbar, click on the CommandButton icon and then, with the cursor over the form, draw out a button.
 - In the button's Properties box, double-click on Caption and change the caption to "Do it!".
 - Click on the TextBox icon and, in the same fashion, draw a text box on the form. In its Properties box, find the Text property and change it to an empty string.
5. Double-click on the Command1 button on the form - a code window will appear. Fill in the code so that it looks like this:

```
Private Sub Command1_Click()
    r1 = 456.78
    Call FortranCall(r1, Num)
    Text1.Text = Str$(Num)
End Sub
```

6. Select Project..Add Module and click on Open to create a new module. Add the following code to the module (all of it goes on one line - broken into multiple lines here for clarity):

```
Declare Sub FortranCall Lib"c:\Program Files\DevStudio\MyProjects\Fcall\Debu
    (r1 As Single, Num As Single)
```

Replace the filename with the location of the Fortran DLL, if it is different.

7. Run the Basic program by pressing F5. Click on the *Do it!* button. The Fortran routine will be called to compute the modulus, returning the result to the Basic code. The Basic code will then convert the result to a string and display it in the text box.

Notes: Visual Basic, like Fortran, passes numeric values (such as integers and reals) by reference, so it is not necessary to change the passing mechanism on either side. The ALIAS attribute is required because Visual Basic, even though it uses the STDCALL calling mechanism, does not "decorate" routine names with the @n suffix. If the Fortran routine were also to be called by other Fortran code, it would be appropriate to use the Alias option on the Basic side to name it with the proper suffix.

Fortran/MASM Mixed-Language Programs

With Microsoft Macro Assembler (MASM), you can combine the unique strengths of assembly-language programming with Visual Fortran. If you structure your assembly-language procedures appropriately, you can call them from Visual Fortran programs and subprograms. MASM works with Visual Fortran, C, and Visual C++. These high-level languages can call MASM procedures, and each of the languages can be called from MASM programs. Details of the MASM interfaces with the other languages can be found in the *Microsoft MASM Programmer's Guide*.

Compile your Fortran source module with Visual Fortran, and assemble your assembly-language procedure with the MASM assembler. Then, link the two object files. The following example shows how to call a MASM assembler-language program from Fortran.

The Fortran code:

```
INTERFACE
    INTEGER (4) FUNCTION POWER2 (V,E)
    !DEC$ ATTRIBUTES STDCALL :: Power2
    INTEGER V, E
END INTERFACE
```

The MASM code:

```
POWER2 PROTO STDCALL, v, e
...
POWER2 PROC STDCALL, v, e
...
POWER2 ENDP
```


END

In the example, the Fortran call to MASM is `power2(v,e)`, which is identical to a Fortran function call.

There are two differences between this mixed-language call and a call between two Fortran modules:

- The subprogram `power2(v,e)` is implemented in MASM using standard MASM syntax. The **PROTO** declaration in MASM specifies that the procedure use the `STDCALL` calling convention.
- The **INTERFACE** statement in the Fortran module specifies the `STDCALL` calling convention, so the Fortran program uses same convention that the MASM procedure specifies.

This section covers the following topics:

- [Creating a MASM Procedure](#)
- [Fortran/MASM Alignment and Return Value Considerations](#)
- [Examples of Fortran/MASM Programming](#)

Creating a MASM Procedure

Normally you follow these steps in creating a MASM procedure:

1. Set up the procedure, defining compatible segments and declaring the procedure.
2. Enter the procedure and set up an appropriate stack frame.
3. Preserve register values by pushing any registers on the stack that you modify later.
4. Reserve space on the stack for any local data (optional).
5. Access arguments in the main body of your procedure.
6. Deallocate any local data by returning space from the stack.
7. Restore register values by popping any preserved registers from the stack.
8. If you called the procedure as a function, return a value (optional).
9. Set up the caller routine by restoring the caller stack frame.
10. Exit the procedure and return to the caller program.

Fortran/MASM Alignment and Return Value Considerations

Visual Fortran allows you to specify alignment for all data objects. Requesting alignment specifies that bytes may be added as padding, so that the object and its data start on a natural boundary (see [Data Alignment Considerations](#)). The MASM default is byte-alignment, so you should specify an alignment of 4 for MASM structures or use the Fortran compiler option `/alignment:keyword` (or `/Zpn`).

Your MASM procedure can return a value to your Fortran routine if you prototype it as a function. All return values of 4 bytes or less (except for floating-point values) are returned in the EAX register.

Procedures that return floating-point values return their results on the floating-point processor stack. This is possible because there is always a coprocessor or emulator available for 32-bit compilers.

To return REAL and COMPLEX floating-point values, records, arrays, and values larger than 4 bytes and return user-defined types larger than 8 bytes from assembly language to Fortran, you must use a special convention. Fortran creates space in the stack segment to hold the actual return value and

passes an extra parameter as the last parameter pushed onto the stack. This extra parameter contains the address of the stack space that contains the return value. For user-defined types, values of 4 bytes or less are returned in EAX and values of 5 to 8 bytes are returned in EAX:EDX.

In the assembly procedure, put the data for the return value at the location pointed to by the return value offset. Then copy the return-value offset (located at EBP+8 if you've created a stack frame in your assembly code) to EAX. This is necessary because the calling module expects EAX to point to the return value.

Summary of Ways to Return Values

Type of value to return	Method of returning value
Integer or logical variable of size 4 bytes or less	Return value in EAX register
Floating-point variable	Return value on the FPU stack
Structure of size more than 4 bytes (strings, complex values) or user-defined structures more than 8 bytes	Return value on stack, address of value in EAX register
User-defined structures between 5 and 8 bytes	Return value in EAX:EDX registers.

Examples of Fortran/MASM Programming

Several sample programs have been provided which illustrate Visual Fortran routines that call MASM procedures (see the \DF\SAMPLES\MIXLANG subdirectory).

Portability

This section presents topics to help you understand how language standards, operating system differences, and computing hardware influence your use of Visual Fortran and the portability of your programs.

Your program is portable if you can implement it on one hardware-software platform and then move it to additional systems with a minimum of changes to the source code. Correct results on the first system should be correct on the additional systems. The number of changes you might have to make when moving your program varies significantly. You might have no changes at all (strictly portable), or so many (non-portable customization) that it is more efficient to design or implement a new program. Most programs in their lifetime will need to be ported from one system to another, and this section can help you write code that makes this easy.

For information on special library routines to help port your program from one system to another, see [Portability Library](#).

For more information, see:

- [Standard Fortran Language](#)
- [Operating System](#)
- [Storage and Representation of Data](#)

Standard Fortran Language

A language standard specifies the form and establishes the interpretation of programs expressed in the language. Its primary purpose is to promote, among vendors and users, portability of programs across a variety of systems.

The vendor-user community has adopted three major Fortran language standards. ANSI (American National Standards Institute) and ISO (International Standards Organization) are the primary organizations that develop and publish the standards.

- FORTRAN IV

American National Standard Programming Language FORTRAN, ANSI X3.9-1966. This was the first attempt to standardize the languages called FORTRAN by many vendors.

- FORTRAN 77

American National Standard Programming Language FORTRAN, ANSI X3.9-1978. This standard added new features based on vendor extensions to FORTRAN IV and addressed problems associated with large-scale projects, such as improved control structures.

- Fortran 90

American National Standard Programming Language Fortran, ANSI X3.198-1992 and International Standards Organization, ISO/IEC 1539: 1991, Information technology -- Programming Languages -- Fortran. This recent standard emphasizes modernization of the language by introducing new developments. For information about differences between

Fortran 90 and FORTRAN 77, see [Features of Fortran 90](#) or the printed *DIGITAL Fortran Language Reference Manual*.

- Fortran 95 (proposed)

This proposed standard introduces certain language elements. Fortran 95 includes Fortran 90 and most features of FORTRAN 77. For information about differences between Fortran 95 and Fortran 90, see [Features of Fortran 95](#) or the printed *DIGITAL Fortran Language Reference Manual*.

Although a language standard seeks to define the form and the interpretation uniquely, a standard might not cover all areas of interpretation. It might also include some ambiguities. You need to carefully craft your program in these cases so that you get the answers that you want when producing a portable program.

For more information, see:

- [Standard vs. Extensions](#)
- [Compiler Optimizations](#)

Standard vs. Extensions

Use standard features to achieve the greatest degree of portability for your Visual Fortran programs. You can design a robust implementation to improve the portability of your program, or you can choose to use extensions to the standard to increase the readability, functionality, and efficiency of your programs. You can ensure your program enforces the Fortran standard by using the `/stand:f90` or `/stand:f95` compiler option to flag extensions.

Not all extensions will cause problems in porting to other platforms. Many extensions are supported on a wide range of platforms, and if a system you are porting a program to supports an extension, there is no reason to avoid using it. There is no guarantee, however, that the same feature on another system will be implemented in the same way as it is in Visual Fortran. Only the Fortran standard is guaranteed to coexist uniformly on all platforms.

DIGITAL Fortran supports many language extensions on multiple platforms, including DIGITAL Alpha systems. For information on compatibility with DIGITAL Fortran on Alpha systems, see [Compatibility with DIGITAL Fortran on Other Platforms](#). Also, the printed *DIGITAL Fortran Language Reference Manual* identifies whether each language element is supported on other DIGITAL Fortran platforms.

It is a good programming practice to declare any external procedures either in an **EXTERNAL** statement or in a procedure interface block, for the following reasons:

- The Fortran 90 standard added many new intrinsic procedures to the language. Programs that conformed to the FORTRAN 77 standard may include nonintrinsic functions or subroutines having the same name as new Fortran 90 procedures.
- Some processors include nonstandard intrinsic procedures that might conflict with procedure names in your program.

If you do not explicitly declare the external procedures and the name duplicates an intrinsic procedure, the processor calls the intrinsic procedure, not your external routine. For more

information on how the Fortran compiler resolves name definitions, see [Resolving Procedure References](#).

Compiler Optimizations

Many Fortran compilers perform code-generation optimizations to increase the speed of execution or to decrease the required amount of memory for the generated code. Although the behaviors of both the optimized and nonoptimized programs fall within the language standard specification, different behaviors can occur in areas not covered by the language standard. Compiler optimization especially can influence floating-point numeric results.

The DIGITAL Visual Fortran compiler can perform optimizations to increase execution speed and to improve floating-point numerical consistency. For a summary of optimization levels, see [Optimization Levels](#).

Floating-point consistency refers to obtaining results consistent with the IEEE binary floating-point standards (see the [/fltconsistency](#) option, x86 systems only).

Unless you properly design your code, you might encounter numerical difficulties when you optimize for fastest execution. The `/nofltconsistency` option uses the floating-point registers, which have a higher precision than stored variables, whenever possible. This tends to produce results that are inconsistent with the precision of stored variables. The [/fltconsistency](#) option (also set by `/Oxp`) can improve the consistency of generated code by rounding results of statement evaluations to the precision of the standard data types, but it does produce slower execution times.

Operating System

The operating system envelops your program and influences it both externally and internally. To achieve portability, you need to minimize the amount of operating-system-specific information required by your program. The Fortran language standards do not specify this information.

Operating-system-specific information consists of nonintrinsic extensions to the language, compiler and linker options, and possibly the graphical user interface of Windows. Input and output operations use devices that may be system-specific, and may involve a file system with system-specific record and file structures. You can find information on the file structure and its use in [Input/Output Editing](#).

The operating system also governs resource management and error handling. You can depend on default resource management and error handling mechanisms or provide mechanisms of your own. For information on special library routines to help port your program from one system to another, see [Portability Library](#).

The minimal interaction with the operating system is for input/output operations and usually consists of knowing the standard units preconnected for input and output. You can use default file units with the asterisk (*) unit specifier.

To increase the portability of your programs across operating systems, consider the following:

- Do not assume the use of a particular type of file system.
- Do not embed filenames or paths in the body of your program. Define them as constants at the

beginning of the program or read them from input data.

- Do not assume a particular type of standard I/O device or the "size" of that device (number of rows and columns).
- Do not assume display attributes for the standard I/O device. Some environments do not support attributes such as color, underlined text, blinking text, highlighted text, inverse text, protected text, or dim text.

Storage and Representation of Data

The Fortran language standard specifies little about the storage of data types. This loose specification of storage for data types results from a great diversity of computing hardware. This diversity poses problems in representing data and especially in transporting stored data among a multitude of systems. The size (as measured by the number of bits) of a storage unit (a word, usually several bytes) varies from machine to machine. In addition, the ordering of bits within bytes and bytes within words varies from one machine to another. Furthermore, binary representations of negative integers and floating-point representations of real and complex numbers take several different forms.

If you are careful, you can avoid most of the problems involving data storage. The simplest and most reliable means of transferring data between dissimilar systems is in *character* and not binary form. Simple programming practices ensure that your data as well as your program is portable.

For more information, see:

- [Size of Basic Types](#)
- [Bit, Byte, and Word Characteristics](#)
- [Transportability of Data](#)

Size of Basic Types

The intrinsic data types are INTEGER, REAL, LOGICAL, COMPLEX, and CHARACTER, whose sizes are shown in the following table.

Data Types and Storage Sizes

Types	Number of Bytes
INTEGER(1), LOGICAL(1), CHARACTER	1
INTEGER(2), LOGICAL(2)	2
INTEGER, LOGICAL, REAL	Depending on default integer size (set by the <code>/integer_size</code> compiler option or equivalent directive), INTEGER and LOGICAL can have 2, 4, or (on Alpha systems only) 8 bytes; default allocation is 4 bytes. Depending on default real size (set by the <code>/real_size</code> compiler option or equivalent directive), REAL can have 4 or 8 bytes; default allocation is 4 bytes.
INTEGER(4), REAL(4), LOGICAL(4)	4
INTEGER(8),	8

LOGICAL(8) (Alpha only)	
COMPLEX	Depending on default real, COMPLEX can have 8 or 16 bytes; default allocation is 8 bytes.
DOUBLE PRECISION, REAL(8), COMPLEX(8)	8
DOUBLE COMPLEX, COMPLEX(16)	16
CHARACTER(n)	n
Structures	Size of derived type (can be affected by PACK directive)
RECORD	Size of record structure (can be affected by PACK directive)

Bit, Byte, and Word Characteristics

In a 32-bit word environment such as that of Visual Fortran, it might seem as though there should be no problems with data storage, since all data types are consecutive subcomponents (bytes) of a word or are consecutive, multiple words. However, when transporting binary data among disparate systems -- either by intermediate storage medium (disk, tape) or by direct connection (serial port, network) -- problems arise from different definitions of serial bit and serial byte order. For simplicity, the following discussion considers only byte order within a word, since that is the usual case of difficulty. (For more information, refer to "On Holy Wars and a Plea for Peace" by Danny Cohen, *IEEE Computer*, vol. 14, pp. 48-54, 1981.)

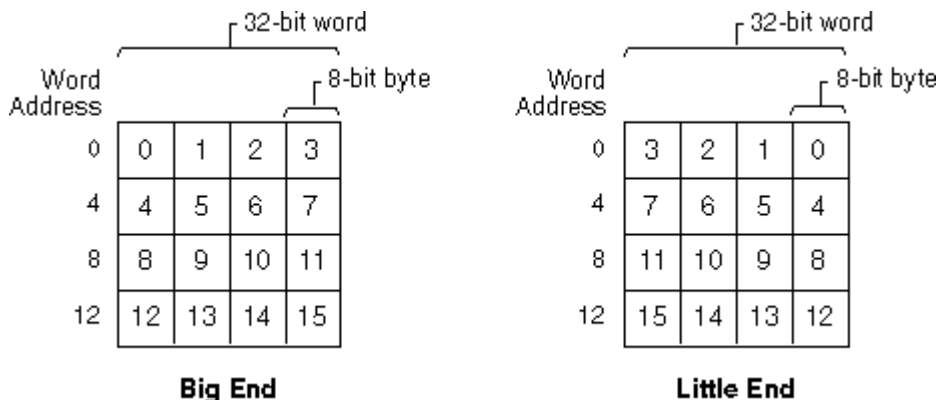
For more information, see:

- [Big End or Little End Ordering](#)
- [Binary Representations](#)
- [Declaring Data Types](#)

Big End or Little End Ordering

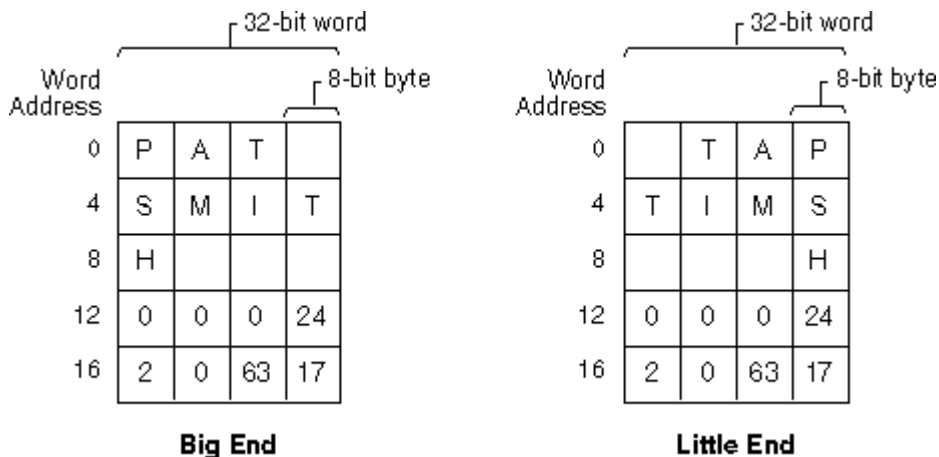
Computer memory is a linear sequence of bits organized into a hierarchical structure of bytes and words. One system is the "Big End," where bits and bytes are numbered starting at the most significant bit (MSB, "left," or high end). Another system is the "Little End," where bits and bytes start at the least significant bit (LSB, "right," or low end). The following figure illustrates the difference between the two conventions for the case of addressing bytes within words.

Byte Order Within Words: (a) Big End, (b) Little End



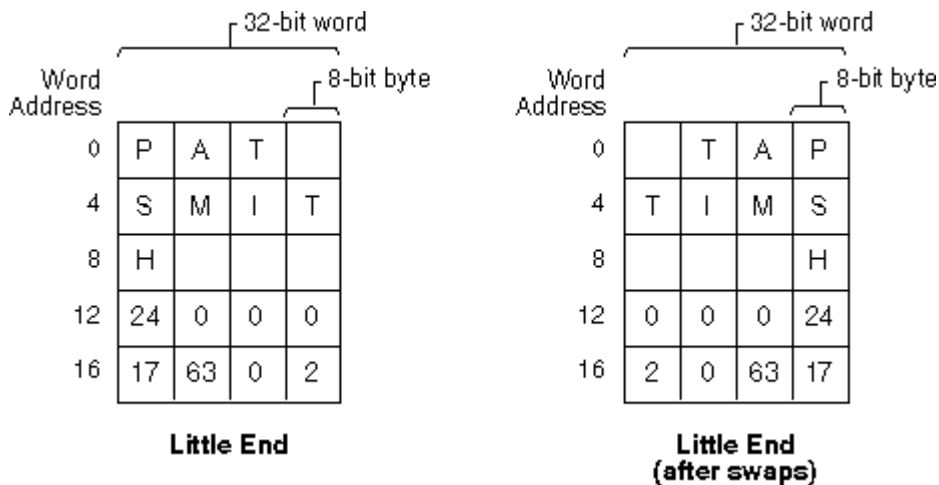
Data types stored as subcomponents (bytes stored in words) end up in different locations within corresponding words of the two conventions. The following figure illustrates the difference between the representation of several data types in the two conventions. Letters represent 8-bit character data, while numbers represent the 8-bit partial contribution to 32-bit integer data.

Character and Integer Data in Words: (a) Big End, (b) Little End



If you serially transfer bytes now from the Big End words to the Little End words (BE byte 0 to LE byte 0, BE byte 1 to LE byte 1, ...), the left half of the figure shows how the data ends up in the Little End words. Note that data of size one byte (characters in this case) is ordered correctly, but that integer data no longer correctly represents the original binary values. The right half of the figure shows that you need to swap bytes around the middle of the word to reconstitute the correct 32-bit integer values. After swapping bytes, the two preceding figures are identical.

Data Sent from Big to Little: (a) After Transfer, (b) After Byte Swaps



You can generalize the previous example to include floating-point data types and to include multiple-word data types. The following table summarizes the ordering nature of several common processors.

Ordering Nature of Processors

Processor	Byte Order	Bit Order
-----------	------------	-----------

Intel® 80486, Pentium®, Pentium Pro	Little	Little
DIGITAL Alpha and VAX™	Little	Little
Motorola® 680XX	Big	Little
IBM® Mainframes	Big	Big

The important result is that portable, serial transport of 8-bit character data between most systems is possible with little or no knowledge about the ordering nature of each system.

For more information on big and little endian data and Visual Fortran unformatted data conversion capabilities, see [Converting Unformatted Numeric Data](#).

Binary Representations

The discussion in [Big End or Little End Ordering](#) stresses 8-bit character data because you might encounter hardware that uses a different representation of binary data. The Visual Fortran system uses the two's-complement representation of negative binary integers. You might encounter a system that uses a signed magnitude representation, a one's complement representation, or a biased (excess) representation. Additionally, the bit representation of binary floating-point numbers is not unique.

If you transport binary data to or from a different system, you need to know the respective representations to convert the binary data appropriately.

Declaring Data Types

Use default data types unless you anticipate memory problems, or if your data is sensitive to overflow limits. If data precision errors or numeric overflow could affect your program, specify type and kind parameters for the intrinsic types as well as for declared data objects. Default data types are portable and are usually aligned by the compiler to achieve good memory access speed. Using some of the nondefault data types on certain machines may slow down memory access.

Transportability of Data

You can achieve the highest transportability of your data by formatting it as 8-bit character data. Use a standard character set such as the ASCII standard for encoding your character data. Although this practice is less efficient than using binary data, it will save you from shuffling and converting your data.

If you are transporting your data by means of a record-structured medium, it is best to use the Fortran sequential formatted (as character data) form. You can also use the direct formatted form, but you need to know the record length of your data. Remember also that some systems use a carriage return-linefeed pair as an end-of-record indicator, while other systems use linefeed only. If you use either the direct unformatted or the sequential unformatted form, there might be system-dependent values embedded within your data that complicate its transport.

Implementing a strictly portable solution requires a careful effort. Maximizing portability may also mean making compromises to the efficiency and functionality of your solution. If portability is not your highest priority, you can use some of the techniques that appear in later sections to ease your task of customizing a solution.

For more information on big and little endian data and unformatted data conversion, see [Converting Unformatted Numeric Data](#).

Using National Language Support Routines

Visual Fortran provides a complete National Language Support (NLS) library of language-localization routines and multibyte-character routines. You can use these routines to write applications in many different languages. In many languages, the standard ASCII character set is not enough because it lacks common symbols and punctuation (such as the British pound sign), or because the language uses a non-ASCII script (such as Cyrillic for Russian) or because the language consists of too many characters for each to be represented by a single byte (such as Chinese).

In the case of many non-ASCII languages, such as Arabic and Russian, an extended single-byte character set is sufficient. You need only change the language locale and codepage, which can be done at a system level or within your program. However, Eastern languages such as Japanese and Chinese use thousands of separate characters that cannot be encoded as single-byte characters. Multibyte characters are needed to represent them.

Character sets are stored in tables called code sets. There are three components of a code set: the locale, which is a language and country (since, for instance, the language Spanish may vary among countries), the codepage, which is a table of characters to make up the computer's alphabet, and the font used to represent the characters on the screen. These three components can be set independently. Each computer running Windows NT or Windows 95 comes with many code sets built into the system, such as English, Arabic, and Spanish. Multibyte code sets, such as Chinese and Japanese, are not standard but come with special versions of the operating system (for instance, Windows NT-J comes with the Japanese code set).

The default code set is obtained from the operating system when a program starts up. When you install your operating system, you should install the system supplied code sets. Thereafter, they are always available. You can switch among them by going into Control Panel in Windows NT or Windows 95, choosing International in Windows NT or Regional Settings in Windows 95, and choosing from the dropdown list of available locales (languages and countries).

When you select a new locale, it becomes the default system locale, and will remain the default locale until you change it. Each locale has a default codepage associated with it, and a default currency, number, and date format.

Note: The default codepage does not change when you select a new locale until you reboot your computer.

You can change the currency, number, and date format in the International dialog box or the Regional Setting dialog box independently of the locale.

The locale determines the character set available to the user. The locale you select becomes the default for the NLS routines described in this section, but the NLS routines allow you to change locales and their parameters from within your programs. These routines are useful for creating original foreign-language programs or different versions of the same program for various international markets. Changes you make to the locale from within a program affect only the program. They do not change the system default settings.

The codepage you select, which can be set independently, controls the multibyte (MB routines) character routines described in this section. Only users with special multibyte-character code sets

installed on their computers need to use MB routines. The standard code sets all use single-byte character code sets.

Note that in Visual Fortran source code, multibyte characters can be used only in character strings and source comments. They cannot be used within variable names or statements. Like program changes to the locale, program changes to codepages affect only the program, not the system defaults.

The NLS and MB routines are contained in the library DFNLS.LIB which consists of DFNLS.MOD and DFNLS.F90. To access the routines, the statement **USE DFNLS** should be present in any program unit that uses NLS or MB routines.

This section includes a discussion of character sets and the NLS library routines:

- [Single and Multibyte Character Sets](#)
- [National Language Support Library Routines](#)

Single and Multibyte Character Sets

The ASCII character set defines the characters from 0 to 127 and an extended set from 128 to 255. Several alternative single-byte character sets, primarily European, define the characters from 0 to 127 identically to ASCII, but define the characters from 128 to 255 differently. With this extension, 8-bit representation is sufficient for defining the needed characters in most European-derived languages. However, some languages, such as Japanese Kanji, include many more characters than can be represented with a single byte. These languages require multibyte coding.

A multibyte character set consists of both one-byte and two-byte characters. A multibyte-character string can contain a mix of single and double-byte characters. A two-byte character has a lead byte and a trail byte. In a particular multibyte character set, the lead and trail byte values can overlap, and it is then necessary to use the byte's context to determine whether it is a lead or trail byte.

National Language Support Library Routines

The library routines for handling extended and multibyte character sets are divided into three categories:

- [Locale Setting and Inquiry Routines to set locales \(local code sets\) and inquire about their current settings](#)
- [NLS Formatting Routines to format dates, currency, and numbers](#)
- [Multibyte character routines](#)

All of these routines are described in detail in the *Reference*.

In the descriptions that follow, function and parameter names are given with a mixture of upper- and lowercase letters. This is to make the names easier to understand. You can use any case for these names when writing your applications.

Locale Setting and Inquiry Routines

At program startup, the current language and country setting is retrieved from the operating system. The user can change this setting through the Windows NT Control Panel/International menu and Windows 95 Control Panel/Regional Settings menu. The current codepage is also retrieved from the system. There is a system default console codepage and a system default Windows codepage. Console programs retrieve the system console codepage, while Windows programs (including QuickWin applications) retrieve the system Windows codepage.

The NLS Library provides routines to determine the current locale (local code set), to return parameters of the current locale, to provide a list of all the system supported locales, and to set the locale to another language, country and/or codepage. These routines are summarized in the following table. Note that the locales and codepages set with these routines affect only the program or console that calls the routine. They do not change the system defaults or affect other programs or consoles.

Routines to Set and Inquire about the Locales

Name	Procedure Type	Description
NLSSetLocale	Function	Sets the language, country and codepage
NLSGetLocale	Subroutine	Retrieves the current language, country and codepage
NLSGetLocaleInfo	Function	Retrieves requested information about the current local code set
NLSEnumLocales	Function	Returns all the languages and country combinations supported by the system
NLSEnumCodepages	Function	Returns all the supported codepages on the system
NLSSetEnvironmentCodepage	Function	Changes the codepage for the current console
NLSGetEnvironmentCodepage	Function	Returns the codepage number for the system (Window) codepage or the console codepage

As an example:

```

USE DFNLS
INTEGER(4) strlen, status
CHARACTER(40) str

strlen = NLSGetLocaleInfo(NLS$LI_SDAYNAME1, str)
print *, str      ! prints Monday
strlen = NLSGetLocaleInfo(NLS$LI_SDAYNAME2, str)
print *, str      ! prints Tuesday
strlen = NLSGetLocaleInfo(NLS$LI_SDAYNAME3, str)
print *, str      ! prints Wednesday
! Change locale to Spanish, Mexico
status = NLSSetLocale("Spanish", "Mexico")
strlen = NLSGetLocaleInfo(NLS$LI_SDAYNAME1, str)
print *, str      ! prints lunes
strlen = NLSGetLocaleInfo(NLS$LI_SDAYNAME2, str)
print *, str      ! prints martes
strlen = NLSGetLocaleInfo(NLS$LI_SDAYNAME3, str)
print *, str      ! prints miércoles
END

```

NLS Formatting Routines

You can set time, date, currency and number formats from the Windows NT Control Panel/International and Windows 95 Control Panel/Regional Settings menu. The NLS Library also provides formatting routines for the current locale. These routines are summarized in the following table. These routines return strings in the current codepage, set by default at program start or by **NLSSetLocale**.

All the formatting routines return the number of bytes in the formatted string (not the number of characters, which can vary if multibyte characters are included). If the output string is longer than the formatted string, the output string is blank padded. If the output string is shorter than the formatted string, an error occurs, **NLS\$ErrorInsufficientBuffer** is returned, and nothing is written to the output string.

Formatting Routines

Name	Procedure Type	Description
NLSFormatCurrency	Function	Formats a number string and returns the correct currency string for the current locale
NLSFormatDate	Function	Returns a correctly formatted string containing the date for the current locale
NLSFormatNumber	Function	Formats a number string and returns the correct number string for the current locale
NLSFormatTime	Function	Returns a correctly formatted string containing the time for the current locale

As an example:

```

USE DFNLS
INTEGER(4) strlen, status
CHARACTER(40) str
strlen = NLSFormatTime(str)
print *, str           ! prints           11:42:24 AM
strlen = NLSFormatDate(str, flags= NLS$LongDate)
print *, str           ! prints           Friday, July 14, 1995
status = NLSSetLocale ("Spanish", "Mexico")
strlen = NLSFormatTime(str)
print *, str           ! prints           11:42:24
print *, str           ! prints viernes 14 de julio de 1995

```

Multibyte Character Routines

All of the routines in this section are intended for use with Multibyte Character Sets (MBCS). Examples of such characters sets are Japanese, Korean, and Chinese. The routines in this section work from the current codepage, set with **NLSSetLocale** and read back with **NLSGetLocale**. String comparison routines, such as **MBLLT**, are based on the current language and country settings.

Routines discussed in this section are:

- [MBCS Inquiry Routines](#)
- [MBCS Conversion Routines](#)
- [MBCS Fortran Equivalent Routines](#)
- [Standard Fortran 90 Routines That Handle MBCS Characters](#)

MBCS Inquiry Routines

The MBCS inquiry routines provide information on the maximum length of multibyte characters, the length, number and position of multibyte characters in strings, and whether a multibyte character is a leading or trailing byte. These routines are summarized in the following table. The NLS library provides a parameter, **MBLenMax**, defined in the NLS module to be the longest length (in bytes) of any character, in any codepage. This parameter can be useful in comparisons and tests. To determine the maximum character length of the current codepage, use the **MBCurMax** function.

MBCS Inquiry Routines

Name	Procedure Type	Description
MBCharLen	Function	Returns the length of the first multibyte character in a string
MBCurMax	Function	Returns the longest possible multibyte character for the current codepage
MBLead	Function	Determines whether a given character is the first byte of a multibyte character
MBLen	Function	Returns the number of multibyte characters in a string, including trailing spaces
MBLen_Trim	Function	Returns the number of multibyte characters in a string, not including trailing spaces
MBNext	Function	Returns the string position of the first byte of the multibyte character immediately after the given string position
MBPrev	Function	Returns the string position of the first byte of the multibyte character immediately before the given string position
MBStrLead	Function	Performs a context sensitive test to determine whether a given byte in a character string is a lead byte

As an example:

```

USE DFNLS
CHARACTER(4) str
INTEGER status
status = NLSSetLocale ("Japan")
str = " ., " ¿"
PRINT '(1X, ''String by char = '',\)'
DO i = 1, len(str)
  PRINT '(A2,\)',str(i:i)
END DO
PRINT '(/,1X, ''MBLead = '',\)'
DO i = 1, len(str)
  PRINT '(L2,\)',mblead(str(i:i))
END DO
PRINT '(/,1X, ''String as whole = '',A,\)',str
PRINT '(/,1X, ''MBStrLead = '',\)'
DO i = 1, len(str)
  PRINT '(L1,\)',MBStrLead(str,i)
END DO
END

```

This code produces the following output for *str = ., " ¿*

```
String by char = . . . .
MLead         = T T T F
String as whole = 高德
MBStrLead     = T T F F
```

MBCS Conversion Routines

There are four MBCS conversion routines: two convert Japan Industry Standard characters to Microsoft Kanji characters or vice versa, and the other two convert between a codepage multibyte character string and a Unicode string. These routines are summarized in the following table.

MBCS Conversion Routines

Name	Procedure Type	Description
MBConvertMBToUnicode	Function	Converts a character string from a multibyte codepage to a Unicode string
MBConvertUnicodeToMB	Function	Converts a Unicode string to a multibyte character string of the current codepage
MBJISToJMS	Function	Converts a Japan Industry Standard (JIS) character to a Microsoft Kanji (Shift JIS or JMS) character
MBJMSToJIS	Function	Converts a Microsoft Kanji (Shift JIS or JMS) character to a Japan Industry Standard (JIS) character

MBCS Fortran Equivalent Routines

The NLS Library provides several functions that are the exact equivalents of Fortran 90 functions except that the MBCS equivalents allow character strings to contain multibyte characters. These routines are summarized in the following table.

MBCS Fortran Equivalent Routines

Name	Procedure Type	Description
MBINCHARQQ	Function	Same as INCHARQQ but can read a single multibyte character at once and returns the number of bytes read
MBINDEX	Function	Same as INDEX except that multibyte characters can be included in its arguments
MBLGE, MBLGT, MBLLE, MBLLT, MBLEQ, MBLNE	Functions	Same as LGE, LGT, LLE, LLT and the operators .EQ. and .NE. except that multibyte characters can be included in their arguments
MBSCAN	Function	Same as SCAN except that multibyte characters can be included in its arguments
MBVERIFY	Function	Same as VERIFY except that multibyte characters can be included in its arguments

The following example is included in the \DF\SAMPLES\TUTORIAL subdirectory as MBCOMP.FOR:


```

USE DFNLIS

INTEGER(4) i, len(7), infotype(7)
CHARACTER(10) str(7)
LOGICAL(4) log4

data infotype / NLS$LI_SDAYNAME1, NLS$LI_SDAYNAME2, &
& NLS$LI_SDAYNAME3, NLS$LI_SDAYNAME4, &
& NLS$LI_SDAYNAME5, NLS$LI_SDAYNAME6, &
& NLS$LI_SDAYNAME7 /
WRITE(*,*) 'NLSGetLocaleInfo'
WRITE(*,*) '-----'
WRITE(*,*) ' '
WRITE(*,*) 'Getting the names of the days of the week...'

DO i = 1, 7
  len(i) = NLSGetLocaleInfo(infotype(i), str(i))
  WRITE(*, 11) 'len/str/hex = ', len(i), str(i), str(i)
END DO
11 FORMAT (1X, A, I2, 2X, A10, 2X, '[' , Z20, ']')

WRITE(*,*) ' '
WRITE(*,*) 'Lexically comparing the names of the days...'

DO i = 1, 6
  log4 = MBLGE(str(i), str(i+1), NLS$IgnoreCase)
  WRITE(*, 12) 'Is day ', i, ' GT day ', i+1, '? Answer = ', log4
END DO
12 FORMAT (1X, A, I1, A, I1, A, L1)

WRITE(*,*) ' '
WRITE(*,*) 'Done.'
END

```

This code produces the following output:

NLSGetLocaleInfo

```

Getting the names of the days of the week...
len/str/hex = 6 月曜日 [8C8E976A93FA20202020]
len/str/hex = 6 火曜日 [89CE976A93FA20202020]
len/str/hex = 6 水曜日 [9085976A93FA20202020]
len/str/hex = 6 木曜日 [96D8976A93FA20202020]
len/str/hex = 6 金曜日 [8BE0976A93FA20202020]
len/str/hex = 6 土曜日 [9379976A93FA20202020]
len/str/hex = 6 日曜日 [93FA976A93FA20202020]

```

Lexically comparing the names of the days...

```

Is day 1 GT day 2? Answer = T
Is day 2 GT day 3? Answer = F
Is day 3 GT day 4? Answer = F
Is day 4 GT day 5? Answer = T
Is day 5 GT day 6? Answer = F
Is day 6 GT day 7? Answer = F

```

Done.

Standard Fortran Routines That Handle MBCS Characters

This section describes Fortran routines that work as usual even if MBCS characters are included in strings.

Because a space can never be a lead or tail byte, many routines that deal with spaces work as expected on strings containing MBCS characters. Such functions include:

ADJUSTL(*string*), **ADJUSTR**(*string*), **TRIM** (*string*)

Some routines work with the computer collating sequence to return a character in a certain position in the sequence or the position in the sequence of a certain character. These functions are not dependent on a particular collating sequence. (You should note, however, that elsewhere in this manual the ASCII collating sequence is mentioned in reference to these functions.) Such functions include:

ACHAR(*position*), **CHAR**(*position* [, *kind*]), **IACHAR**(*c*), **ICHAR**(*c*)

Because Fortran uses character lengths instead of NULLs to indicate the length of a string, some functions work solely from the length of the string, and not with the contents of the string. These functions work as usual on strings containing MBCS characters, and include:

REPEAT (*string*, *ncopies*)

The Floating-Point Environment

This section describes the Visual Fortran numeric environment using IEEE® arithmetic for x86 and Alpha systems. The following topics are covered:

- [Representing Numbers](#)
- [Loss of Precision Errors: Rounding, Special Values, Underflow, and Overflow](#)
- [Setting and Retrieving Floating-Point Status and Control Words \(x86 systems only\)](#)
- [Handling Arithmetic Exceptions](#)
- [Intel Pentium Floating-Point Flaw \(x86 systems only\)](#)

When the term floating-point unit (FPU) appears, it refers to your math processor, which could be a math coprocessor for a 486 SX CPU, an integrated floating-point unit in an Intel® 486, Pentium®, or Pentium Pro processor, or software that emulates a coprocessor. The reference manual for your FPU describes its registers and features. The descriptions in this section minimize hardware-specific terminology.

Visual Fortran supplies a single library for floating-point operations. Earlier versions of Visual Fortran had several versions that you could specify, depending on your hardware configuration and your software development goals. You do not have to specify a library; the operating system selects the appropriate routines to execute at run-time. If the system on which your program executes contains an FPU, the hardware routines execute; if not, software routines emulate an FPU.

Representing Numbers

Fortran's numeric environment is flexible, which helps make Fortran a strong language for intensive numerical calculations. The Fortran standard purposely leaves the precision of numeric quantities and the method of rounding numeric results unspecified. This allows Fortran to operate efficiently for diverse applications on diverse systems.

The effect of math computations on integers is straightforward. Integers of `KIND=4` consist of a maximum positive integer (2,147,483,647), a minimum negative integer (-2,147,483,648), and all integers between them including zero. Operations on integers result in other integers within this range. The only arithmetic rule to remember is that integer division results in truncation (for example, $8/3$ evaluates to 2).

Computations on real numbers, however, may not yield what you expect. This happens because the hardware must represent numbers in a finite number of bits.

There are several effects of using finite floating-point numbers. The hardware is not able to represent every real number exactly, but must approximate exact representations by rounding or truncating to finite length. In addition, some numbers lie outside the range of representation of the maximum and minimum exponents and can result in calculations that underflow and overflow. As an example of one consequence, finite precision produces many numbers that, although non-zero, behave in addition as zero.

You can minimize the effects of finite representation with programming techniques; for example, by

not using floating-point numbers in LOGICAL comparisons or by giving them a tolerance (for example, IF ($x \leq 10.001$)), and by not attempting to combine or compare numbers that differ by more than the number of significant bits. (For more information on programming methods to reduce the effects of imprecision, see [Rounding Errors](#).)

For further discussion of how floating-point numbers are represented, see:

- [Floating-Point Numbers](#)
- [Retrieving Parameters of Numeric Representations](#)

Floating-Point Numbers

This version of Visual Fortran uses a close approximation to the IEEE floating-point standard (ANSI/IEEE Std 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic*, 1985). This standard is common to many microcomputer-based systems due to the availability of fast math coprocessors that implement the required characteristics. Software emulation of the standard is also possible if a hardware system does not include a coprocessor.

You should choose the appropriate setting of the `/fpe` compiler option to select the type of default floating-point exception handling provided by the Visual Fortran run-time system.

This section outlines the characteristics of the standard and its implementation for Visual Fortran. Except as noted, the description includes both the IEEE standard and the Visual Fortran implementation. The following topics are discussed:

- [Floating-Point Formats](#)
- [Floating-Point Representation](#)
- [Viewing Floating-Point Representations with BitViewer](#)
- [Special Values \(Signed Zero, NaN, Signed Infinity\)](#)

Floating-Point Formats

The IEEE® Standard 754 specifies values and requirements for floating-point representation (such as base 2). The standard outlines requirements for two formats: basic and extended, and for two word-lengths within each format: single and double.

Visual Fortran supports single-precision format (REAL(4)) and double-precision format (REAL(8)) floating-point numbers. Visual Fortran sets the process control word by default to use double-precision run-time intermediate calculations. At some levels of optimization, some single-precision numbers are stored on the floating-point stack (which defaults to double precision) rather than being stored back into memory where they would be truncated to single precision. The compiler option `/fltconsistency (x86 only)` can control floating-point consistency and request that results be stored in memory rather than on the floating-point stack.

Floating-Point Representation

Floating-point numbers approximate real numbers with a finite number of bits. You can see the bits representing a floating-point number with the BitViewer tool. The bits are calculated as shown in the following formula. The representation is binary, so the base is 2. The bits b_n represent binary digits (0 or 1). The precision P is the number of bits in the nonexponential part of the number (the

significand), and E is the exponent. With these parameters, binary floating-point numbers approximate real numbers with the values:

$$(-1)^s b_0 . b_1 b_2 \dots b_{p-1} \times 2^E$$

where s is 0 or 1 (+ or -), and $E_{\min} \leq E \leq E_{\max}$. The following table gives the standard values for these parameters for single, double, and extended-double formats and the resulting bit widths for the sign, the exponent, and the full number.

Parameters for IEEE Floating-Point Formats

Parameter	Single	Double	Extended double
Sign width in bits	1	1	1
P	24	53	64
E_{\max}	+127	+1023	+16383
E_{\min}	- 126	- 1022	- 16382
Exponent <i>bias</i>	+ 127	+1023	+16383
Exponent width in bits	8	11	15
Format width in bits	32	64	80

The standard requires that the single and double formats be normalized, so b_0 is always 1. The actual number of bits needed to represent the precisions 24 and 53 is therefore 23 and 52, respectively, because b_0 is chosen to be 1 implicitly.

Extended-double format need not be normalized, so it uses the full 64 bits for precision. A *bias* is added to all exponents so that only positive integer exponents occur. This expedites comparisons of exponent values. The stored exponent is actually:

$$e = E + \textit{bias}.$$

For more information on:

- Floating-point representation, see [Native IEEE Floating-Point Representations](#).
- Using the Bitviewer tool, see [Viewing Floating-Point Representations with BitViewer](#).
- Reading or writing floating-point data other than native IEEE little endian data, see [Converting Unformatted Numeric Data](#).

Viewing Floating-Point Representations with BitViewer

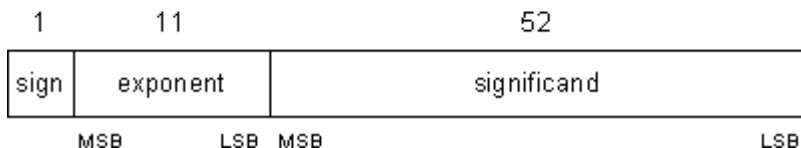
You can view the binary representation of real numbers in single and double format with the BitViewer utility. This tool is accessed from the command line with the command BITVIEW. By default Visual Fortran installs the BitViewer utility in the directory ...\\DF\\BIN.

The following figure shows the logical layout of the single and double formats. The figure shows the contents of each field, its width, and the location of the most significant bit (MSB) and the least significant bit (LSB).

Logical Structure of the IEEE Single and Double Formats

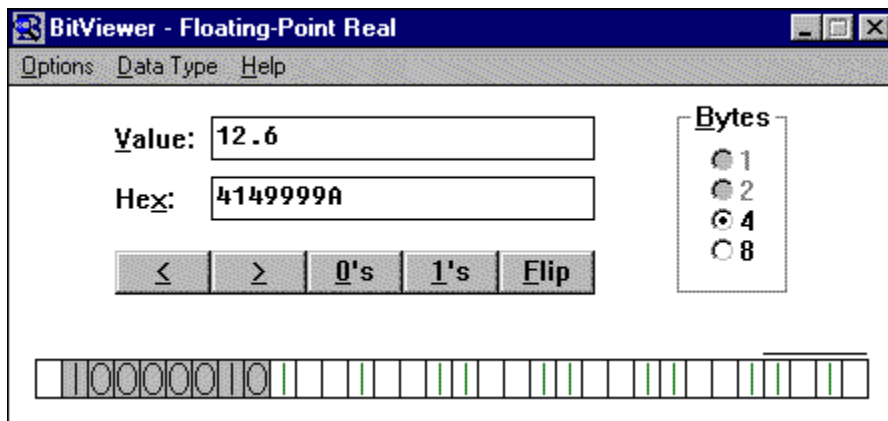


Double Format

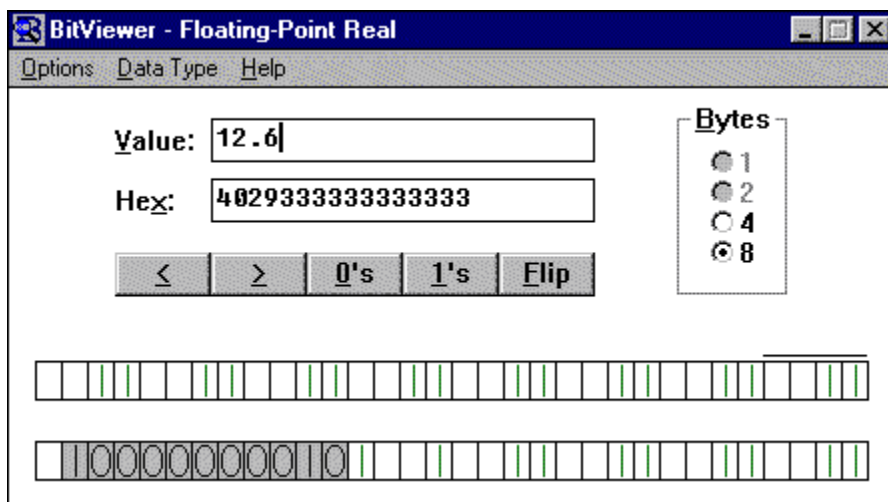


To view floating-point numbers in BitViewer, open the Data Type menu, then choose Floating-Point Real (or use the F9 shortcut key). Set the precision by selecting one of the choices in the Bytes box. Four bytes, REAL(4), displays the number in single format (23-bit precision). Eight bytes, REAL(8), displays the number in double format (52-bit precision). The following figures show the BitViewer display of the memory storage for a 4-byte real number and an 8-byte real number, both equal to 12.6. In the double format display, the most significant part is on the bottom and the least significant 32 bits above.

Single Format in BitViewer



Double Format in BitViewer



Note: BitViewer allows you to view and manipulate integer and character data as well as floating-point, and to translate between different data types. Refer to the BitViewer Help file for more information.

Special Values

Special cases of the exponent-significand combination represent four types of special values in addition to the normalized numbers. The following table shows all five types of values.

IEEE Floating-Point Values

Name	Quantity	Exponent	Significand
Signed zero	± 0	$E = E_{\min} - 1$	$sig = 0$
Denormalized number	$\pm 0 . sig \times 2^{E_{\min}}$	$E = E_{\min} - 1$	sig not equal 0
Normalized number	$\pm 1 . sig \times 2^E$	$E_{\min} \leq E \leq E_{\max}$	sig
Signed infinity	\pm infinity	$E = E_{\max} + 1$	$sig = 0$
Not a Number	NaN	$E = E_{\max} + 1$	sig not equal 0

These special values are interpreted as follows:

- Signed zero

Visual Fortran treats zero as signed by default. The sign of zero is the same as the sign of a nonzero number. If you use the intrinsic function **SIGN** with zero as the second argument, the sign of the zero will be transferred. Comparisons, however, consider +0 to be equal to -0. A signed zero is useful in certain numerical analysis algorithms, but in most applications the sign of zero is invisible.

- Denormalized numbers

Denormalized numbers (denormals) fill the gap between the smallest positive number and the smallest negative number. Otherwise only (\pm) 0 occurs in that interval. Denormalized numbers permit gradual underflow for intermediate results calculated internally in extended-double format. A status flag (on x86 systems, the precision bit in the FPU Status Word exception field) is set when a number loses precision due to denormalization.

- Signed infinity

Infinities are the result of arithmetic in the limiting case of operands with arbitrarily large magnitude. They provide a way to continue when an overflow occurs. The sign of an infinity is simply the sign you obtain for a finite number in the same operation as the finite number approaches an infinite value. By retrieving the status flags described in [Setting and Retrieving Floating-Point Status and Control Words](#) in this section, you can differentiate between an infinity that results from an overflow and one that results from division by zero. Visual Fortran treats infinity as signed by default. The output value of infinity is `Infinity` or `-Infinity`.

- Not a Number

Not a Number (NaN) results from an operation involving one or more invalid operands. For

instance $0/0$ and $\text{SQRT}(-1)$ result in NaN. In general, an operation involving a NaN produces another NaN. Because the fraction of a NaN is unspecified, there are many possible NaNs.

Visual Fortran treats all NaNs identically, but provide two different types:

- Signaling NaN, which has an initial fraction bit of 0 (zero).
- Quiet NaN, which has an initial fraction bit of 1.

The output value of NaN is NaN.

Retrieving Parameters of Numeric Representations

Visual Fortran includes several intrinsic functions that return details about the numeric representation. These are listed in the following table and described fully in the *Reference*.

Functions that return numeric parameters

Name	Description	Argument/Function type
<u>DIGITS</u>	DIGITS (x). Returns number of significant digits for data of the same type as x	x : Integer or Real result: INTEGER(4)
<u>EPSILON</u>	EPSILON (x). Returns the smallest positive number that when added to one produces a number greater than one for data of the same type as x	x : Real result: same type as x
<u>EXPONENT</u>	EXPONENT (x). Returns the exponent part of the representation of x	x : Real result: INTEGER(4)
<u>FRACTION</u>	FRACTION (x). Returns the fractional part (significand) of the representation of x	x : Real result: same type as x
<u>HUGE</u>	HUGE (x). Returns largest number that can be represented by data of type x	x : Integer or Real result: same type as x
<u>MAXEXPONENT</u>	MAXEXPONENT (x). Returns the largest positive decimal exponent for data of the same type as x	x : Real result: INTEGER(4)
<u>MINEXPONENT</u>	MINEXPONENT (x). Returns the largest negative decimal exponent for data of the same type as x	x : Real result: INTEGER(4)
<u>NEAREST</u>	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
<u>PRECISION</u>	PRECISION (x). Returns the number of significant digits for data of the same type as x	x : Real or Complex result: INTEGER(4)
<u>RADIX</u>	RADIX (x). Returns the base for data of the same type as x	x : Integer or Real result: INTEGER(4)
<u>RANGE</u>	RANGE (x). Returns the decimal exponent range for data of the same type as x	x : Integer, Real or Complex result: INTEGER(4)
<u>RRSPACING</u>	RRSPACING (x). Returns the reciprocal of the relative spacing of numbers near x	x : Real result: same type as x
<u>SCALE</u>	SCALE (x, i). Multiplies x by 2 raised to the power of i	x : Real i : Integer result: same type as x
<u>SET_EXPONENT</u>	SET_EXPONENT (x, i). Returns a number whose	x : Real

	fractional part is x and whose exponential part is i	i : Integer result: same type as x
<u>SPACING</u>	SPACING (x). Returns the absolute spacing of numbers near x	x : Real result: same type as x
<u>TINY</u>	TINY (x). Returns smallest positive number that can be represented by data of type x	x : Real result: same type as x

Loss of Precision Errors: Rounding, Special Values, Underflow, and Overflow

If a real number is not exactly one of the representable floating-point numbers, then the nearest floating-point number must represent it. The rounding error is the difference between the exact real number and its nearest floating-point representation. The floating-point number representing a rounded real number is called *inexact*.

Normally, calculations proceed when an inexact value results. Almost any floating-point operation can produce an inexact result. The rounding mode (round up, round down, round nearest, truncate) is determined by the floating-point control word.

If an arithmetic operation does not result in an exact, valid floating-point number, which includes numbers that have been rounded to an exactly representable floating-point number, it results in a special value: signed zero, signed infinity, NaN, or a denormal. Special-value results are a limiting case of the arithmetic operation involved. Special values can propagate through your arithmetic operations without causing your program to fail, and often providing usable results.

If an arithmetic operation results in an exact value, but the value is invalid, the operation causes underflow or overflow:

- Underflow occurs when an arithmetic result is too small for the math processor to handle. Depending on the setting of the `/fpe` compiler option, underflows are set to zero (they are usually harmless) or they are left as is (denormalized).
- Overflow occurs when an arithmetic result is too large for the math processor to handle. Overflows are more serious than underflows, and may indicate an error in the formulation of a problem (for example, unintended exponentiation of a large number by a large number). Overflows produce an appropriately signed infinity value.

Inexact numbers, special values, underflows, and overflows are floating-point exceptions. You can select how rounding is done and how exceptions are handled by setting the floating-point control word. Setting the control word is described in [Setting and Retrieving Floating-Point Status and Control Words \(x86 only\)](#) and exception handling in [Handling Floating-Point Exceptions \(x86 only\)](#).

For a further discussion of rounding errors see:

- [Rounding Errors](#)

Rounding Errors

Although the rounding error for one real number might be acceptably small in your calculations, at least two problems arise because of it. If you test for exact equality between what you consider to be two exact numbers, the rounding error of either or both floating-point representations of those

numbers may prevent a successful comparison and produce spurious results. Also, when you calculate with floating-point numbers the rounding errors may accumulate to a meaningful loss of numerical significance.

Carefully consider the numerics of your solution to minimize rounding errors or their effects. You might benefit from using double-precision arithmetic or restructuring your algorithm, or both. For instance, if your calculations involve arrays of linear data items, you might reduce the loss of numerical significance by subtracting the mean value of each array from each array element and by normalizing each element of such an array to the standard deviation of the array elements.

The following code segment can execute differently on different systems and produce different results for *n*, *x*, and *s*. It also produces different results if you use the `/fltconsistency` or `/nofltconsistency` compiler options on x86 systems. Rounding error accumulates in *x* because the floating-point representation of 0.2 is inexact, then accumulates in *s*, and affects the final value for *n*:

```

      INTEGER n
      REAL s, x
      n = 0
      s = 0.
      x = 0.
1     n = n + 1
      x = x + 0.2
      s = s + x
      IF ( x .LE. 10. ) GOTO 1 ! Will you get 51 cycles?
      WRITE(*,*) 'n = ', n, ' ; x = ', x, ' ; s = ', s

```

This example illustrates a common coding problem: carrying a floating-point variable through many successive cycles and then using it to perform an **IF** test. This process is common in numerical integration. There are several remedies. You can compute *x* and *s* as multiples of an integer index, for example, replacing the statement that increments *x* with $x = n * 0.2$ to avoid round-off accumulation. You might test for completion on the integer index, such as `IF (n <= 50) GOTO 1`, or use a **DO** loop, such as `DO n= 1,51`. If you must test on the real variable that is being cycled, use a realistic tolerance, such as `IF (x <= 10.001)`.

Floating-point arithmetic does not always obey the standard rules of algebra exactly. Addition is not precisely associative when round-off errors are considered. You can use parentheses to express the exact evaluation you require to compute a correct, accurate answer. This is recommended when you specify optimization for your generated code, since associativity may otherwise be unpredictable.

The expressions $(x + y) + z$ and $x + (y + z)$ can give unexpected results in some cases, as the example `ASSOCN.F90` in the `\DF\SAMPLES\TUTORIAL` subdirectory shows. This example demonstrates the danger of combining two numbers whose values differ by more than the number of significant digits.

The example `INTERVAL.F90` in the `\DF\SAMPLES\TUTORIAL` subdirectory shows how changing the rounding precision and rounding mode in the floating-point control word between calculations affects the calculated result of the following simple expression:

$$(q*r + s*t) / (u + v)$$

The example `EPSILON.F90` in the `\DF\SAMPLES\TUTORIAL` subdirectory illustrates difficulties that rounding errors can cause in expressions like $1.0 + \text{eps}$, where *eps* is just significant compared to

1.0.

The compiler uses the default rounding mode (round-to-nearest) during compilation. The compiler performs more compile-time operations that eliminate run-time operations as the optimization level increases. If you set rounding mode to a different setting (other than round-to-nearest), that rounding mode is used only if that computation is performed at run-time. For example, the Sample INTERVAL.F90 is compiled at /optimize:0, which disables certain compile-time optimizations, including constant propagation and inlining.

For more information, see:

- [ULPs, Relative Error, and Machine Epsilon](#)
- [/rounding_mode \(Alpha only\)](#)

ULPs, Relative Error, and Machine Epsilon

Several terms describe the magnitude of rounding error. A floating-point approximation to a real constant or to a computed result may err by as much as $1/2$ unit in the last place (the b_{P-1} bit). The abbreviation *ULP* represents the measure "unit in the last place." Another measure of the rounding error uses the relative error, which is the difference between the exact number and its approximation divided by the exact number. The relative error that corresponds to $1/2$ *ULP* is bounded by:

$$1/2 \cdot 2^{-P} \leq 1/2 \text{ ULP} \leq 2^{-P}.$$

The upper bound $EPS = 2^{-P}$, the machine epsilon, is commonly used in discussions of rounding errors because it expresses the smallest floating-point number that you can add to 1.0 with a result that does not round to 1.0.

Additional guard bits are included in floating-point hardware to allow rounding on the order of *EPS*. The result of any one floating-point operation is therefore tolerably imprecise, but the total error that results from many such operations on propagated numbers accumulates unavoidably.

Setting and Retrieving Floating-Point Status and Control Words (x86 only)

The FPU (floating-point unit) on x86 systems contains eight floating-point registers the system uses for numeric calculations, for status and control words, and for error pointers. You normally need to consider only the status and control words, and then only when customizing your floating-point environment.

The FPU status and control words correspond to 16-bit registers whose bits hold the value of a state of the FPU or control its operation. Visual Fortran defines a set of symbolic constants to set and reset the proper bits in the status and control words. For example:

```
USE DFLIB
CALL SETCONTROLFPQQ(FPCW$OVERFLOW .AND. FPCW$CHOP)
! set the floating-point control word to allow overflows
! and to round by truncation
```

The status and control symbolic constants (such as **FPCW\$OVERFLOW** and **FPCW\$CHOP** in the preceding example) are defined as INTEGER(2) parameters in the module DFLIB.F90 in the

\DF\INCLUDE subdirectory. The status and control words are made of logical combinations (such as with **.AND**.) of different parameters for different FPU options.

The name of a symbolic constant takes the general form *name\$option*. The prefix *name* is one of the following:

Prefixes for Parameter Flags

<i>name</i>	Meaning
FPSW	Floating-point status word
FPCW	Floating-point control word
SIG	Signal
FPE	Floating-point exception
MTH	Math function

The suffix *option* is one of the options available for that *name*. The parameter *name\$option* corresponds either to a status or control option (for example, **FPSW\$ZERODIVIDE**, a status word parameter that shows whether a zero-divide exception has occurred or not) or *name\$option* corresponds to a mask, which sets all symbolic constants to 1 for all the options of *name*. You can use the masks in logical functions (such as **IAND**, **IOR**, and **NOT**) to set or to clear all options for the specified *name*. The following sections define the *options* and illustrate their use with examples.

You can control the floating-point processor options (on x86 systems) and find out its status with the run-time library routines GETSTATUSFPQQ (x86 only), GETCONTROLFPQQ (x86 only), SETCONTROLFPQQ (x86 only), and MATHERRQQ (x86 only). Examples of using these routines also appear in the following sections.

For more information, see:

- Floating-Point Status Word (x86 only)
- Floating-Point Control Word (x86 only)

Floating-Point Status Word (x86 only)

On x86 systems, the FPU status word includes bits that show the floating-point exception state of the processor. The status word parameters describe six exceptions: invalid result, denormalized result, zero divide, overflow, underflow and inexact precision. These are described in the section, Loss of Precision Errors: Rounding, Special Values, Underflow, and Overflow. When one of the bits is set to 1, it means a past floating-point operation produced that exception type. (Visual Fortran initially clears all status bits. It does not reset the status bits before performing additional floating-point operations after an exception occurs. The status bits accumulate.) The following table shows the floating-point exception status parameters:

Parameter name	Value in hex	Description
FPSW\$MSW_EM	#003F	Status Mask (set all bits to 1)
FPSW\$INVALID	#0001	An invalid result occurred
FPSW\$DENORMAL	#0002	A denormal (very small number) occurred
FPSW\$ZERODIVIDE	#0004	A divide by zero occurred
FPSW\$OVERFLOW	#0008	An overflow occurred

FPSW\$UNDERFLOW	#0010	An underflow occurred
FPSW\$INEXACT	#0020	Inexact precision occurred

You can find out which exceptions have occurred by retrieving the status word and comparing it to the exception parameters. For example:

```
USE DFLIB
INTEGER(2) status
CALL GETSTATUSFPQQ(status)
IF ((status .AND. FPSW$INEXACT) > 0) THEN
    WRITE (*, *) "Inexact precision has occurred"
ELSE IF ((status .AND. FPSW$DENORMAL) > 0) THEN
    WRITE (*, *) "Denormal occurred"
END IF
```

Floating-Point Control Word (x86 only)

On x86 systems, the FPU control word includes bits that control the FPU's precision, roundingmode, and whether exceptions generate signals if they occur. You can read the control word value with GETCONTROLFPQQ (x86 only) to find out the current control settings, and you can change the control word with SETCONTROLFPQQ (x86 only).

Each bit in the floating-point control word corresponds to a mode of the floating-point math processor. The DFLIB.F90 module file in the \DF\INCLUDE subdirectory contains theINTEGER(2) parameters defined for the control word, as shown in the following table.

Parameter name	Value in hex	Description
FPCW\$MCW_IC	#1000	Infinity control mask
FPCW\$AFFINE	#1000	Affine infinity
FPCW\$PROJECTIVE	#0000	Projective infinity
FPCW\$MCW_PC	#0300	Precision control mask
FPCW\$64	#0300	64-bit precision
FPCW\$53	#0200	53-bit precision
FPCW\$24	#0000	24-bit precision
FPCW\$MCW_RC	#0C00	Rounding control mask
FPCW\$CHOP	#0C00	Truncate
FPCW\$UP	#0800	Round up
FPCW\$DOWN	#0400	Round down
FPCW\$NEAR	#0000	Round to nearest
FPCW\$MSW_EM	#003F	Exception mask
FPCW\$INVALID	#0001	Allow invalid numbers
FPCW\$DENORMAL	#0002	Allow denormals (very small numbers)
FPCW\$ZERODIVIDE	#0004	Allow divide by zero
FPCW\$OVERFLOW	#0008	Allow overflow
FPCW\$UNDERFLOW	#0010	Allow underflow
FPCW\$INEXACT	#0020	Allow inexact precision

The control word defaults are:

- 53-bit precision
- Round to nearest (rounding mode)
- The denormal, underflow, overflow, invalid, and inexact precision exceptions are disabled (do not generate an exception). To change exception handling, you can use the `/fpe` compiler option or the `FOR_SET_FPE` routine.

For more information (x86 only), see:

- [Exception Parameters](#)
- [Precision Parameters](#)
- [Rounding Parameters](#)

Exception Parameters

An exception is disabled if its bit is set to 1 and enabled if its bit is cleared to 0. If an exception is disabled (exceptions can be disabled by setting the flags to 1 with `SETCONTROLFPQQ (x86 only)`), it will not generate an interrupt signal if it occurs. The floating-point process will return an appropriate special value (for example, NaN or signed infinity), but the program continues. You can find out which exceptions (if any) occurred by calling `GETSTATUSFPQQ (x86 only)`. If errors on floating-point exceptions are enabled (by clearing the flags to 0 with `SETCONTROLFPQQ (x86 only)`), the operating system generates an interrupt when the exception occurs. By default these interrupts cause run-time errors, but you can capture the interrupts with `SIGNALQQ` and branch to your own error-handling routines.

You should remember not to clear all existing settings when changing one. The values you want to change should be combined with the existing control word in an inclusive-**OR** operation (**OR**, **IOR**, **.OR.**) if you don't want to reset all options. For example:

```
USE DFLIB
INTEGER(2) control, newcontrol
CALL GETCONTROLFPQQ(control)
newcontrol = (control .OR. FPCW$INVALID)
! Invalid exception set (disabled).
CALL SETCONTROLFPQQ(newcontrol)
```

Precision Parameters

On x86 systems, the precision bits control the precision to which the FPU rounds floating-point numbers. For example:

```
USE DFLIB
INTEGER(2) control, holdcontrol, newcontrol
CALL GETCONTROLFPQQ(control)
! Clear any existing precision flags.
holdcontrol = (control .AND. (.NOT. FPCW$MCW_PC))
newcontrol = holdcontrol .OR. FPCW$64
! Set precision to 64 bits.
CALL SETCONTROLFPQQ(newcontrol)
```

The precision options are mutually exclusive. If you set more than one, you may get an invalid mode or a mode other than the one you want. Therefore, you should clear the precision bits before setting a new precision mode.

Rounding Parameters

On x86 systems, the rounding flags control the method of rounding that the FPU uses. For example:

```
USE DFLIB
INTEGER(2) control, clearcontrol, newcontrol
CALL GETCONTROLFPQQ(control)
! Clear any existing rounding flags.
clearcontrol = (control .AND. (.NOT. FPCW$MCW_RC))
newcontrol = clearcontrol .OR. FPCW$SUP
! Set rounding mode to round up.
CALL SETCONTROLFPQQ(newcontrol)
```

The rounding options are mutually exclusive. If you set more than one, you may get an invalid mode or a mode other than the one you want. Therefore, you should clear the rounding bits before setting a new rounding mode.

On Alpha systems, you can use /rounding_mode (Alpha only) compiler option to control the rounding mode.

Handling Arithmetic Exceptions

Two levels of arithmetic exceptions occur in Visual Fortran. Low-level exceptions result from floating-point exceptions. High-level exceptions result from arithmetic errors that occur during execution of the mathematical functions. You have some flexibility in handling each type of exception.

The following sections describe:

- Floating-Point Exceptions
- Handling Run-Time Math Exceptions (x86 only)

Handling Floating-Point Exceptions

If a floating-point exception is disabled (set to 1), it will not generate an interrupt signal if it occurs. The floating-point process will return an appropriate special value (for example, NaN or signed infinity), and the program will continue. If a floating-point exception is enabled (set to 0), it will generate an interrupt signal (software interrupt) if it occurs. The following table lists the floating-point exception signals.

Parameter name	Value in hex	Description
FPE\$INVALID	#81	Invalid result
FPE\$DENORMAL	#82	Denormal operand
FPE\$ZERODIVIDE	#83	Divide by zero
FPE\$OVERFLOW	#84	Overflow
FPE\$UNDERFLOW	#85	Underflow
FPE\$INEXACT	#86	Inexact precision

If a floating-point exception interrupt occurs and you do not have an exception handling routine, the run-time system will respond to the interrupt according to the behavior selected by the compiler

option `/fpe`. Remember, interrupts only occur if an exception is enabled (set to 0).

If you do not want the default system exception handling, you need to write your own interrupt handling routine. This is a relatively simple two-step process:

- Write a function that performs whatever special behavior you require on the interrupt.
- Register that function as the procedure to be called on that interrupt with **SIGNALQQ**.

Note that your interrupt handling function must use the **ATTRIBUTES** C directive.

The drawback of writing your own routine is that your exception-handling routine cannot return to the process that caused the exception. This is because when your exception-handling routine is called, the floating-point processor is in an error condition, and if your routine returns, the processor is in the same state, which will cause a system termination. Your exception-handling routine can therefore either branch to another separate program unit or exit (after saving your program state and printing an appropriate message). You cannot return to a different statement in the program unit that caused the exception-handling routine, because a global **GOTO** does not exist, and you cannot reset the status word in the floating-point processor.

If you need to know when exceptions occur and also must continue if they do, you must disable exceptions so they do not cause an interrupt, then poll the floating-point status word at intervals with **GETSTATUSFPOQ (x86 only)** to see if any exceptions occurred. Obviously, this creates processing overhead for your program. In general, you will want to allow the program to terminate if there is an exception. An example of an exception-handling routine follows. The exception-handling routine `hand_fpe` and the program that invokes it are both contained in `SIGTEST.F90` in the `\DF\SAMPLES\TUTORIAL` subdirectory. The comments at the beginning of the `SIGTEST.F90` file describe how to compile this example.

```
! SIGTEST.F90
! Establish the name of the exception handler as the
! function to be invoked if an exception happens.
! The exception handler hand_fpe is attached below.
USE DFLIB
INTERFACE
  FUNCTION hand_fpe (sigid, except)
    !DEC$ATTRIBUTES C :: hand_fpe
    INTEGER(4) hand_fpe
    INTEGER(2) sigid, except
  END FUNCTION
END INTERFACE

INTEGER(4) iret
REAL(4) r1, r2
r1 = 0.0
iret = SIGNALQQ(SIG$FPE, hand_fpe)
WRITE(*,*) 'Set exception handler. Return = ', iret
! Cause divide by zero exception
r1 = 0.0
r2 = 3/r1
END

! Exception handler routine hand_fpe
FUNCTION hand_fpe (signum, excnum)
  !DEC$ ATTRIBUTES C :: hand_fpe
  USE DFLIB
  INTEGER(2) signum, excnum
  WRITE(*,*) 'In signal handler for SIG$FPE'
```



```

WRITE(*,*) 'signum = ', signum
WRITE(*,*) 'exception = ', excnum
SELECT CASE(excnum)
  CASE(FPE$INVALID )
    STOP ' Floating point exception: Invalid number'
  CASE( FPE$DENORMAL )
    STOP ' Floating point exception: Denormalized number'
  CASE( FPE$ZERODIVIDE )
    STOP ' Floating point exception: Zero divide'
  CASE( FPE$OVERFLOW )
    STOP ' Floating point exception: Overflow'
  CASE( FPE$UNDERFLOW )
    STOP ' Floating point exception: Underflow'
  CASE( FPE$INEXACT )
    STOP ' Floating point exception: Inexact precision'
  CASE DEFAULT
    STOP ' Floating point exception: Non-IEEE type'
END SELECT
hand_fpe = 1
END

```

Handling Run-Time Math Exceptions (x86 only)

On x86 systems, the run-time subroutine **MATHERRQQ (x86 only)** handles floating-point exceptions that occur in the math functions, such as **SIN** and **LOG10**. If you use the default version of **MATHERRQQ**, which the linker automatically includes in your executable program, then math exceptions result in a standard run-time error (such as `forrtl: severe (nnnn): sqrt: domain error`). If you want to alter the behavior of one or more math exceptions, you need to provide your own version of **MATHERRQQ**. You have more flexibility in the way you handle run-time math exceptions than floating-point exceptions, because your error handling routine can return to the program unit that caused the exception.

The module `DFLIB.F90` in the `\DF\INCLUDE` subdirectory contains the definitions of the run-time math exceptions. These are listed in the following table:

Parameter name	Value	Description
MTH\$E_DOMAIN	1	Argument domain error
MTH\$E_SINGULARITY	2	Argument singularity
MTH\$E_OVERFLOW	3	Overflow range error
MTH\$E_UNDERFLOW	4	Underflow range error
MTH\$E_TLOSS	5	Total loss of precision
MTH\$E_PLOSS	6	Partial loss of precision

A domain error means that an argument is outside the math function's domain, for example, `SQRT(-1)`. A singularity error means that an argument is a singularity value for the math function, and the result is not defined for that value, for example, `LOG10(0.0)`. Overflow and underflow errors are the same as floating-point counterparts, and precision loss the same as floating-point inexact results.

You can write a **MATHERRQQ** subroutine that resolves errors generated by math functions. Your **MATHERRQQ** can issue a warning, assign a default value if an error occurs, or take other action. If you do not provide your own **MATHERRQQ** subroutine, a default **MATHERRQQ** provided with the floating-point library will terminate the program. The following gives an example of an

alternative **MATHERRQQ** subroutine (in MATHERR.F90 in the /DF/SAMPLES/TUTORIAL subdirectory):

```

SUBROUTINE MATHERRQQ( name, length, info, retcode)
  USE DFLIB
  INTEGER(2) length, retcode
  CHARACTER(length) name
  RECORD /MTH$E_INFO/ info
  PRINT *, "Entered MATHERRQQ"
  PRINT *, "Failing function is: ", name
  PRINT *, "Error type is: ", info.errcode
  IF ((info.ftype == TY$REAL4 ).OR.(info.ftype == TY$REAL8)) THEN
    PRINT *, "Type: REAL"
    PRINT *, "Enter the desired function result: "
    READ(*,*) info.r8res
    retcode = 1
  ELSE IF ((info.ftype == TY$CMPLX8 ).OR.(info.ftype == TY$CMPLX16)) THEN
    PRINT *, "Type: COMPLEX"
    PRINT *, "Enter the desired function result: "
    READ(*,*) info.c16res
    retcode = 1
  END IF
END

```

The following is an example program (MATHTEST.F90 in the /DF/SAMPLES/TUTORIAL subdirectory) that causes **MATHERRQQ** to be called:

```

REAL(4) r1, r2 /-1.0/
REAL(8) r3, r4 /-1.0/
COMPLEX(4) c1, c2 /(0.0, 0.0)/
r1 = LOG(r2)
r3 = SQRT(r4)
c1 = CLOG(c2)

WRITE(*, *) r1
WRITE(*, *) r3
WRITE(*, *) c1
END

```

Intel Pentium Floating-Point Flaw (x86 only)

Certain versions of the Intel® Pentium® processor have a flaw in rare floating-point division operations, which can also manifest itself in floating-point **TAN**, **ATAN**, and **MOD** operations. Since the number of input cases that cause this problem is very small and the associated error in the results is also very small, it is unlikely that you will ever see a problem due to this flaw. It has been estimated that only 1 out of 9 billion operations will produce even the slightest inaccuracy.

To request a check for a flawed Pentium chip, you can use the default compiler option /check:flawed_pentium option (in the Run-Time Checking category of the Fortran tab in Developer Studio) on code you suspect demonstrates the Pentium flaw, such as the code shown below. This compiler option generates run-time calls to the run-time routine FOR_CHECK_FLAWED_PENTIUM. The default, /check:flawed_pentium, *does* issue a run-time error message for this condition and stops program execution. To allow program execution to continue when this condition occurs, set the environment variable FOR_RUN_FLAWED_PENTIUM to true and rerun the program (see Run-Time Environment Variables).

Visual Fortran does not include a software workaround for the flawed Pentium problems, and these operations could produce incorrect results on a flawed Pentium processor.

To determine if you have a flawed pentium, you can run the following program with the /check:flawed_pentium compiler option:

```
PROGRAM go
REAL(8) op1, op2
COMMON /divide_check/ op1, op2
DATA op1 /3145727.0/, op2 /4195835.0/
IF( op2/op1 > 1.3338 ) THEN
  PRINT *, 'This computer always divides correctly.'
ELSE
  PRINT *, 'This computer can have divide problems.'
ENDIF
END
```

If you compile and run this program without any compiler options (the default is /check:flawed_pentium), a run-time error occurs when it is run on a flawed Pentium system.

Both Windows NT and Windows 95 can also work around flawed Pentium processors by using software emulation for floating-point operations. Refer to your operating system documentation for more information. If the operating system has been configured for software emulation, then all floating-point operations in Visual Fortran will always operate correctly, including the above program. Note that the performance cost of an operating system fix can be very high, and if your program is run on another machine without the same operating system fix, it will execute incorrectly.

If you distribute software that is susceptible to the floating-point problems of a flawed Pentium, and want your program to halt if it is run on a system with such a processor, you can check the processor when your application starts. To do this, convert the program above into a simple subroutine, call the subroutine at the start of your application, and use the **STOP** statement to stop the application before it begins if a flawed Pentium processor is detected. If you distribute your software, you should compile it with the /check:flawed_pentium compiler option.

All the run-time libraries that come with Visual Fortran have been compiled to be safe with respect to the Pentium divide and **MOD** problems.

For more information on the Intel Pentium flaw, or to request a replacement Pentium processor, you can contact Intel in the US at 1-800-628-8686.

Handling Run-Time Errors

This section contains information on the following topics:

- [Default Run-Time Error Processing](#), including the message format and values returned at program termination.
- [Handling Run-Time Errors](#) within your program, including using the **END**, **EOR**, and **ERR** I/O statement branch specifiers and the **IOSTAT** specifier.
- [Locating Run-Time Errors](#), including suggested compiler options and information about debugging exceptions.
- [Run-Time Environment Variables](#) that allow program continuation under certain conditions, disable the display of certain dialog boxes under certain conditions, and allow "just-in-time" debugging.

During execution, your program may encounter errors or exception conditions. These conditions can result from any of the following:

- Errors that occur during I/O operations
- Invalid input data
- Argument errors in calls to the mathematical library
- Arithmetic errors
- Other system-detected errors

The DIGITAL Visual Fortran Run-Time Library (RTL) generates appropriate messages and takes action to recover from errors whenever possible.

For a description of each Visual Fortran run-time error message, see [Run-Time Errors](#)

Default Run-Time Error Processing

The Visual Fortran RTL processes a number of errors that can occur during program execution. A default action is defined for each error recognized by the Visual Fortran RTL. The default actions described throughout this chapter occur unless overridden by explicit error-processing methods.

The way in which the Visual Fortran RTL actually processes errors depends upon the following factors:

- The severity of the error. For instance, the program usually continues executing when an error message with a severity level of warning or info (informational) is detected.
- For certain errors associated with I/O statements, whether or not an I/O error-handling specifier was specified.
- For certain errors, whether or not the default action of an associated signal was changed.
- For certain errors related to arithmetic operations (including floating-point exceptions), compilation options can determine whether the error is reported and the severity of the reported error.

How arithmetic exception conditions are reported and handled depends on the cause of the exception and how the program was compiled. Unless the program was compiled to handle exceptions, the

exception might not be reported until *after* the instruction that caused the exception condition.

For More Information:

- On the Visual Fortran message format, see [Run-Time Message Format](#)
- On the Visual Fortran return values at program termination, see [Values Returned at Program Termination](#)
- On locating errors and the compiler options related to handling errors and exceptions, see [Locating Run-Time Errors](#).
- On DF command options and their categories in Developer Studio (Project Settings, Fortran tab), see [Categories of Compiler Options](#).
- On DIGITAL Fortran intrinsic data types and their ranges, see [Data Representation](#).
- On the floating-point environment, see [The Floating-Point Environment](#)

Run-Time Message Format

When errors occur during program execution (run time) of a scalar (nonparallel) program, the Visual Fortran RTL issues diagnostic messages. These run-time messages have the following format:

```
forrtl: severity (number): message-text
```

Run-time messages provide the following information:

Contents	Information Given
<i>forrtl</i>	Identifies the source as the Visual Fortran RTL.
<i>severity</i>	The severity levels are: <i>severe</i> , <i>error</i> , <i>warning</i> , or <i>info</i> (abbreviation of information) (see the table Severity Levels of Run-Time Messages).
<i>number</i>	This is the message number, also the IOSTAT value for I/O statements.
<i>message-text</i>	Explains the event that caused the message.

The following table explains the severity levels of run-time messages, in the order of greatest to least severity:

Severity Levels of Run-Time Messages

Severity	Description
<i>severe</i>	Must be corrected. The program's execution is terminated when the error is encountered unless the program's I/O statements use the END, EOR, or ERR branch specifiers to transfer control, perhaps to a routine that uses the IOSTAT specifier (see the section Using the END, EOR, and ERR Branch Specifiers and the section Using the IOSTAT Specifier and Fortran Exit Codes).
<i>error</i>	Should be corrected. The program might continue execution, but the output from this execution may be incorrect.
<i>warning</i>	Should be investigated. The program continues execution, but output from this execution may be incorrect.
<i>info</i>	For informational purposes only; the program continues.

For a description of each Visual Fortran run-time error message, see [Run-Time Errors](#)

Values Returned at Program Termination

A Visual Fortran program can terminate in one of several ways:

- The program runs to normal completion. A value of zero is returned to the shell.
- The program stops with a **PAUSE** statement. A value of zero is returned to the shell.
- The program stops because of a signal that is caught but does not allow the program to continue. A value of 1 is returned to the shell.
- The program stops because of a severe run-time error. The error number for that run-time error is returned to the shell. Error numbers are listed in [Run-Time Errors](#).
- The program stops with a **CALL EXIT** statement. The value passed to **EXIT** is returned to the shell.

Methods of Handling Errors

Whenever possible, the Visual Fortran RTL does certain error handling, such as generating appropriate messages and taking necessary action to recover from errors. You can explicitly supplement or override default actions by using the following methods:

- To transfer control to error-handling code within the program, use the **END**, **EOR**, and **ERR** branch specifiers in I/O statements, see [Using the END, EOR, and ERR Branch Specifiers](#).
- To identify Fortran-specific I/O errors based on the value of Visual Fortran RTL errorcodes, use the I/O status specifier (**IOSTAT**) in I/O statements (or call the **ERRSNS** subroutine), see [Using the IOSTAT Specifier and Fortran Exit Codes](#).

These error-processing methods are complementary; you can use any or all of them within the same program to obtain Visual Fortran run-time error codes.

On Alpha systems, if your program generates an exception, unless you are using the `/fpe:0` option, consider using the [/synchronous_exceptions](#) option and recompile and relink your application.

Using the END, EOR, and ERR Branch Specifiers

When a severe error occurs during Visual Fortran program execution, the default action is to display an error message and terminate the program. To override this default action, there are three branch specifiers you can use in I/O statements to transfer control to a specified point in the program:

- The **END** branch specifier handles an end-of-file condition.
- The **EOR** branch specifier handles an end-of-record condition for nonadvancing reads.
- The **ERR** branch specifier handles all error conditions.

If you use the **END**, **EOR**, or **ERR** branch specifiers, no error message is displayed and execution continues at the designated statement, usually an error-handling routine.

You might encounter an unexpected error that the error-handling routine cannot handle. In this case, do one of the following:

- Modify the error-handling routine to display the error message number
- Remove the **END**, **EOR**, or **ERR** branch specifiers from the I/O statement that causes the error

After you modify the source code, compile, link, and run the program to display the error message. For example:

```
READ ( 8,50,ERR=400 )
```

If any severe error occurs during execution of this statement, the Visual Fortran RTL transfers control to the statement at label 400. Similarly, you can use the **END** specifier to handle an end-of-file condition that might otherwise be treated as an error. For example:

```
READ ( 12,70,END=550 )
```

When using nonadvancing I/O, use the **EOR** specifier to handle the end-of-record condition. For example:

```
150 FORMAT (F10.2, F10.2, I6)  
READ (UNIT=20, FMT=150, SIZE=X, ADVANCE='NO', EOR=700) A, F, I
```

You can also use **ERR** as a specifier in an **OPEN**, **CLOSE**, or **INQUIRE** statement. For example:

```
OPEN (UNIT=10, FILE='FILNAM', STATUS='OLD', ERR=999)
```

If an error is detected during execution of this **OPEN** statement, control transfers to the statement at label 999.

Using the IOSTAT Specifier and Fortran Exit Codes

You can use the **IOSTAT** specifier to continue program execution after an I/O error and to return information about I/O operations (I/O is done in a scalar fashion). As described in [Run-Time Errors](#), certain errors are *not* returned in IOSTAT.

Although the **IOSTAT** specifier transfers control, it can only return information returned by the Visual Fortran RTL. For additional Win32 error handling capabilities, see .

The IOSTAT specifier can supplement or replace the **END**, **EOR**, and **ERR** branch transfers.

Execution of an I/O statement containing the **IOSTAT** specifier suppresses the display of an error message and defines the specified integer variable, array element, or scalar field reference as one of the following, which is returned as an exit code if the program terminates:

- A value of -2 if an end-of-record condition occurs with nonadvancing reads.
- A value of -1 if an end-of-file condition occurs.
- A value of 0 for normal completion (not an error condition, end-of-file, or end-of-record condition).
- A positive integer value if an error condition occurs. (This value is one of the Fortran-specific **IOSTAT** numbers listed in [Run-Time Errors](#).)

Following the execution of the I/O statement and assignment of an **IOSTAT** value, control transfers to the **END**, **EOR**, or **ERR** statement label, if any. If there is no control transfer, normal execution

continues. For more information on transfer of control, see [Errors, End-of-File, and End-of-Record Handling Specifiers](#).

You can include the `iosdef.for` file in your program to obtain symbolic definitions for the values of **IOSTAT**.

The following example uses the **IOSTAT** specifier and the `iosdef.for` file to handle an **OPEN** statement error (in the **FILE** specifier).

Example: Error Handling OPEN Statement File Name

```

CHARACTER(LEN=40) :: FILNM
INCLUDE 'iosdef.for'

DO I=1,4
  FILNM = ''
  WRITE (6,*) 'Type file name '
  READ (5,*) FILNM
  OPEN (UNIT=1, FILE=FILNM, STATUS='OLD', IOSTAT=IERR, ERR=100)
  WRITE (6,*) 'Opening file: ', FILNM
!   (process the input file)
  CLOSE (UNIT=1)
  STOP

100 IF (IERR .EQ. FOR$IOS_FILNOTFOU) THEN
  WRITE (6,*) 'File: ', FILNM, ' does not exist '
ELSE IF (IERR .EQ. FOR$IOS_FILNAMSP) THEN
  WRITE (6,*) 'File: ', FILNM, ' was bad, enter new file name'
ELSE
  PRINT *, 'Unrecoverable error, code =', IERR
  STOP
END IF
END DO
WRITE (6,*) 'File not found. Locate correct file with Explorer and run again'
END PROGRAM

```

Locating Run-Time Errors

This section provides some guidelines for locating the cause of exceptions and run-time errors. In Version 5.0, Visual Fortran error messages do not usually indicate the exact source location causing the error.

To locate the cause of errors use the various compiler options to isolate programming errors at compile-time and run-time or use the debugger to locate the cause of exceptions:

- The `/[no]warn` options control compile-time diagnostic messages, which in some circumstances can help determine the cause of a run-time error. In Developer Studio, specify the Warning Level in the General [Compiler Option Category](#) or specify individual Warning Options in the Miscellaneous [Compiler Option Category](#).
- The `/check:keyword` options generate extra code to catch certain conditions at run-time (see [/check](#) or in Developer Studio, specify the Extended Error Checking items in the Run time [Compiler Option Category](#)). For example:
 - The `/check:bounds` option generates extra code to catch access to data beyond the array

- or string boundaries.
- The `/check:overflow` option generates extra code to catch integer overflow conditions.
- The `/check:noformat`, `/check:nooutput_conversion`, and `/check:nopower` options reduce the severity level of the associated run-time error to allow program continuation (see [/check](#)).
- The `/check:underflow` option controls the reporting of floating-point underflow exceptions at run-time.
- The `/fpe` option controls the handling of floating-point arithmetic exceptions (IEEE arithmetic) at run-time. In Developer Studio, specify the Floating-Point Exception Handling in the Floating Point [Compiler Option Category](#).

For example, if you specified `/fpe:3`, exceptions related to exceptional IEEE values are not reported and your application may generate exceptional IEEE values, which later in your application may generate an exception or unexpected values. By recompiling the application at `/fpe:0`, any exceptional IEEE values generated will cause the program to terminate and report an error message earlier.

The [FOR_GET_FPE](#) and [FOR_SET_FPE](#) routines are also available to examine and set the run-time handling of certain arithmetic exceptions.

- On Alpha systems, the `/synchronous_exceptions` option (and certain `/fpe:n` option) influence the reporting of floating-point arithmetic exceptions at run-time. In Developer Studio, specify Enable Synchronous Floating-Point Exceptions in the Floating Point [Compiler Option Category](#).
- For many types of errors, using the Debugger can help you isolate the cause of errors.

In the Debug menu, select the Exceptions item to specify the types of exceptions that will stop program execution. When using this method, be aware that your program must use compiler options that allow the debugger to catch the appropriate exceptions.

For example, if you do not specify `/check:bounds`, the debugger will not catch and stop at array or character string bounds errors. Similarly, if you specify `/fpe:3`, certain floating-point exceptions will not be caught, since this setting allows IEEE exceptional values and program continuation.

For information on using the Debugger, see the Debugger section in the [Developer Studio Environment User's Guide](#).

Run-Time Environment Variables

The Visual Fortran run-time system recognizes the following environment variables:

Environment Variable	Description
FOR_DEFAULT_PRINT_DEVICE	Lets you specify the print device other than the default print device PRN (LPT1) for files closed (CLOSE statement) with the DISPOSE='PRINT' specifier. To specify a different print device for the file associated with the CLOSE statement DISPOSE='PRINT' specifier, set the environment variable

	FOR_DEFAULT_PRINT_DEVICE to any legal DOS print device before executing the program.
FOR_IGNORE_EXCEPTIONS	If set to true, disables the default run-time exception handling, for example, to allow just-in-time debugging. The run-time system exception handler returns EXCEPTION_CONTINUE_SEARCH to the operating system, which looks for other handlers to service the exception. For information on just-in-time debugging, see Running Fortran Applications and the Developer Studio Environment User's Guide .
FOR_RUN_FLAWED_PENTIUM	If set to true, allows the continuation of the executing program when <code>/check:flawed_pentium</code> (default) is in effect and a flawed Pentium chip is detected. For more information, see Intel Pentium Floating-Point Flaw .
FOR_NOERROR_DIALOGS	If set to true, disables the display of dialog boxes when certain exceptions or errors occur. This is useful when running many test programs in batch mode to prevent a failure from stopping execution of the entire test stream.
FORT n	Lets you specify the file name for a particular unit number (n), when a file name is not specified in the OPEN statement or an implicit OPEN is used, and the compiler option <code>/fpscomp:filesfromcmd</code> was <i>not</i> specified. Preconnected files attached to units 0, 5, and 6 are by default associated with system standard I/O files.
FORT_CONVERT n	Lets you specify the data format for an unformatted file associated with a particular unit number (n), as described in Methods of Specifying the Data Format .

Use the SET command to display (SET with no parameters) or set (SET *environment variable=value*) environment variables from the command line.

For a list of environment variables used with the DF command, see [Environment Variables Used with the DF Command](#).

Converting Unformatted Numeric Data

This section describes how you can use DIGITAL Visual Fortran to read and write nonnative unformatted numeric data, including DIGITAL Fortran for OpenVMS systems numeric data.

The following topics are available:

- [Supported Native and Nonnative Numeric Formats](#)
- [Limitations of Numeric Conversion](#)
- [Methods of Specifying the Data Format](#)
- [Environment Variable FORT_CONVERT_n Method](#)
- [OPEN Statement CONVERT='keyword' Method](#)
- [OPTIONS Statement Method](#)
- [Compiler Option /convert:keyword Method](#)
- [Additional Notes on Nonnative Data](#)

Supported Native and Nonnative Numeric Formats

DIGITAL Visual Fortran supports the following little endian floating-point formats in memory:

Floating-Point Size	Format in Memory
REAL(KIND=4), COMPLEX(KIND=4)	IEEE S_floating
REAL(KIND=8), COMPLEX(KIND=8)	IEEE T_floating

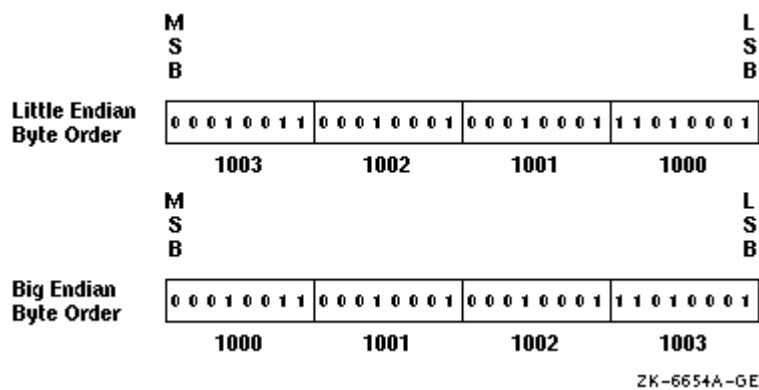
If your program needs to read or write unformatted data files containing a floating-point format that differs from the format in memory for that data size, you can request that the unformatted data be converted.

Data storage in different computers uses a convention of either little endian or big endian storage. The storage convention generally applies to numeric values that span multiple bytes, as follows:

- Little endian storage occurs when:
 - The least significant bit (LSB) value is in the byte with the lowest address.
 - The most significant bit (MSB) value is in the byte with the highest address.
 - The address of the numeric value is the byte containing the LSB. Subsequent bytes with higher addresses contain more significant bits.
- Big endian storage occurs when:
 - The least significant bit (LSB) value is in the byte with the highest address.
 - The most significant bit (MSB) value is in the byte with the lowest address.
 - The address of the numeric value is the byte containing the MSB. Subsequent bytes with higher addresses contain less significant bits.

The following figure shows the difference between the two byte-ordering schemes.

Little and Big Endian Storage of an INTEGER Value



Moving unformatted data files between big endian and little endian computers requires that the data be converted.

Visual Fortran provides the capability for programs to read and write unformatted data (originally written using unformatted I/O statements) in several nonnative floating-point formats and in big endian INTEGER or floating-point format. Supported nonnative floating-point formats include DIGITAL VAX™ little endian floating-point formats supported by VAX FORTRAN™, standard IEEE® big endian floating-point format found on most Sun Microsystems® systems and IBM® RISC System/6000 systems, IBM floating-point formats (associated with the IBM's System/370 and similar systems), and CRAY® floating-point formats.

Converting unformatted data instead of formatted data is generally faster and is less likely to lose precision of floating-point numbers.

The native memory format includes little endian integers and little endian IEEE floating-point formats, S_float for REAL(KIND=4) and COMPLEX(KIND=4) declarations and T_float for REAL(KIND=8) and COMPLEX(KIND=8) declarations.

The keywords for supported nonnative unformatted file formats and their data types are listed in the following table.

Nonnative Numeric Formats, Keywords, and Supported Data Types

Keyword	Description
BIG_ENDIAN	Big endian integer data of the appropriate size (one, two, or four bytes) and big endian IEEE floating-point (REAL(KIND=4), REAL(KIND=8), COMPLEX(KIND=4), COMPLEX(KIND=8)) formats of the appropriate size for either real or complex numbers. INTEGER(KIND=1) data is the same for little endian and big endian.
CRAY	Big endian integer data of the appropriate size (one, two, four, or on Alpha systems, eight bytes) and big endian CRAY proprietary floating-point format of size REAL(KIND=8) or COMPLEX(KIND=8).
FDX	Little endian integer data of the appropriate size (one, two, four, or on Alpha systems, eight bytes) and DIGITAL VAX floating-point data of format F_floating for REAL(KIND=4) or COMPLEX(KIND=4), and D_Floating for REAL(KIND=8) or COMPLEX(KIND=8).
FGX	Little endian integer data of the appropriate size (one, two, four, or on Alpha systems, eight bytes) and DIGITAL VAX floating-point data of format

	F_floating for REAL(KIND=4) or COMPLEX(KIND=4), and G_floating for REAL(KIND=8) or COMPLEX(KIND=8).
IBM	Big endian integer data of the appropriate size (one, two, or four bytes) and big endian IBM proprietary floating-point format of size REAL(KIND=4) or COMPLEX(KIND=4) or size REAL(KIND=8) or COMPLEX(KIND=8).
LITTLE_ENDIAN	Native little endian integers of the appropriate size (one, two, four, or on Alpha systems, eight bytes) and native little endian IEEE floating-point data of the appropriate size and type (REAL(KIND=4), REAL(KIND=8), COMPLEX(KIND=4), COMPLEX(KIND=8)). These are the same formats as stored in memory. For additional information on supported ranges for these data types, see <u>Native IEEE Floating-Point Representations</u> .
NATIVE	No conversion occurs between memory and disk. This is the default for unformatted files.
VAXD	Little endian integers of the appropriate size (one, two, four, or on Alpha systems, eight bytes) and DIGITAL VAX floating-point format F_floating for size REAL(KIND=4) or COMPLEX(KIND=4), and D_floating for size REAL(KIND=8) or COMPLEX(KIND=8).
VAXG	Little endian integers of the appropriate size (one, two, four, or on Alpha systems, eight bytes) and DIGITAL VAX floating-point format F_floating for size REAL(KIND=4) or COMPLEX(KIND=4), and G_floating for size REAL(KIND=8) or COMPLEX(KIND=8).

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 is displayed.

Limitations of Numeric Conversion

The DIGITAL Visual Fortran floating-point conversion solution is not expected to fulfill all floating-point conversion needs.

For instance, data in record structures variables (specified in a **STRUCTURE** statement) are not converted. When they are later examined as separate fields by the program, they will remain in the binary format they were stored in on disk, unless the program is modified. With **EQUIVALENCE** statements, the data type of the variable named in the I/O statement is used.

If a program reads an I/O record containing multiple format floating-point fields into a single variable (such as an array) instead of their respective variables, the fields will not be converted. When they are later examined as separate fields by the program, they will remain in the binary format they were stored in on disk, unless the program is modified.

The conversion of the following file structure types are *not* supported:

- Binary data (FORM='BINARY')
- Formatted data (FORM='FORMATTED')
- Unformatted data (FORM='UNFORMATTED') written by Microsoft Fortran Powerstation or by Visual Fortran with the /fpscomp:ioformat compiler option in effect.

Methods of Specifying the Data Format

There are four methods of specifying a nonnative numeric format for unformatted data. If none of these methods are specified, the native `LITTLE_ENDIAN` format is assumed (no conversion occurs between disk and memory).

Any keyword listed in [Supported Native and Nonnative Numeric Formats](#) can be used with any of these methods.

The four methods you can use to specify the type of nonnative (or native) format are as follows:

- Setting an environment variable for a specific unit number before the file is opened. The environment variable is named `FORT_CONVERT n` , where n is the unit number.
- Compiling the program with an **OPTIONS** statement that specifies the `/CONVERT= keyword` qualifier. This method affects all unit numbers using unformatted data specified by the program.
- Specifying the **CONVERT** keyword in the **OPEN** statement for a specific unit number.
- Compiling the program with the appropriate compiler option (DF command `/convert: keyword` or Developer Studio equivalent), which affects all unit numbers that use unformatted data specified by the program.

If you specify more than one method, the order of precedence when you open a file with unformatted data is to:

- Check for an environment variable first
- Check the **OPEN** statement **CONVERT** specifier
- Check whether an **OPTIONS** statement with a `/CONVERT=(keyword)` qualifier was present when the program was compiled
- Check whether the compiler option `/convert:keyword` was present when the program was compiled

The following sections describe each method:

- [Environment Variable FORT_CONVERT \$n\$ Method](#)
- [OPEN Statement CONVERT= Method](#)
- [OPTIONS Statement Method](#)
- [Compiler Option /convert Method](#)

Environment Variable FORT_CONVERT n Method

You can use this method to specify multiple formats in a single program, usually one format for each specified unit number. You specify the numeric format at run time by setting the appropriate environment variable before an implicit or explicit **OPEN** to that unit number.

When the appropriate environment variable is set when you open the file, the environment variable is always used because this method takes precedence over the other methods. For instance, you might use this method to specify that a unit number will use a particular format instead of the format specified in the program (perhaps for a one-time file conversion).

For example, assume you have a previously compiled program that reads numeric data from unit 28

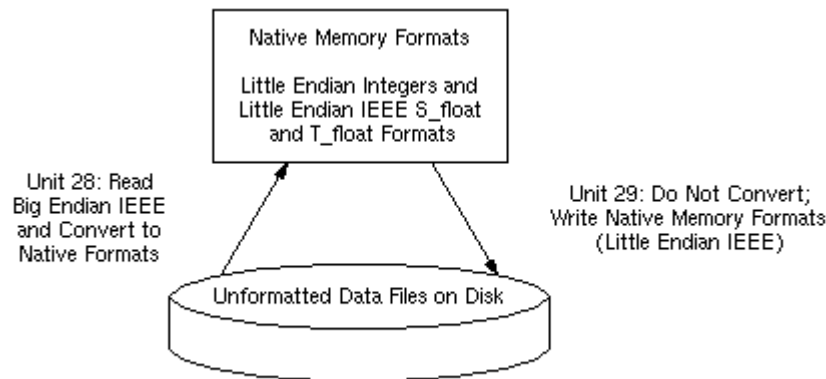
and writes it to unit 29 using unformatted I/O statements. You want the program to read nonnative big endian (IEEE floating-point) format from unit 28 and write that data in native little endian format to unit 29. In this case, the data is converted from big endian IEEE format to native little endian IEEE memory format (S_float and T_float) when read from unit 28, and then written without conversion in native little endian IEEE format to unit 29.

Without requiring source code modification or recompilation of this program, the following command sequence sets the appropriate environment variables before running the program (c:\users\leslie\conveeee.exe):

```
set FORT_CONVERT28=BIG_ENDIAN
set FORT_CONVERT29=NATIVE
c:\users\leslie\conveeee.exe
```

The following figure shows the data formats used on disk and in memory when the example file c:\users\leslie\conveeee.exe is run after the environment variables are set.

Sample Unformatted File Conversion



ZK-8326A-GE

OPEN Statement CONVERT Method

You can use this method to specify multiple formats in a single program, usually one format for each specified unit number. This method requires an explicit file **OPEN** statement to specify the numeric format of the file for that unit number.

This method takes precedence over the **OPTIONS** statement and the compiler option /convert:keyword method, but has a lower precedence than the environment variable method.

For example, the following source code shows how the **OPEN** statement would be coded to read unformatted VAXD numeric data from unit 15, which might be processed and possibly written in native little endian format to unit 20 (the absence of the **CONVERT** keyword or environment variable FORT_CONVERT20 indicates native little endian data for unit 20):

```
OPEN ( CONVERT='VAXD', FILE='graph3.dat', FORM='UNFORMATTED', UNIT=15)
.
.
.
OPEN ( FILE='graph3_t.dat', FORM='UNFORMATTED', UNIT=20)
```

A hard-coded **OPEN** statement **CONVERT** keyword value cannot be changed after compile time. However, to allow selection of a particular format at run time, equate the **CONVERT** keyword to a variable and provide the user with a menu that allows selection of the appropriate format (menu choice sets the variable) before the **OPEN** occurs. You can also select a particular format for a unit number at run time by using the environment variable method, which takes precedence over the **OPEN** statement **CONVERT** keyword method.

OPTIONS Statement Method

You can only specify one numeric file format for all unformatted file unit numbers using this method unless you also use the logical name or **OPEN** statement **CONVERT** specifier method.

You specify the numeric format at compile time and must compile all routines under the same **OPTIONS** statement `/CONVERT=keyword` qualifier. You could use one source program and compile it using different DF commands to create multiple executable programs that each read a certain format.

The environment variable or **OPEN** statement **CONVERT** specifier methods take precedence over this method. For instance, you might use the logical name or **OPEN CONVERT** specifier method to specify each unit number that will use a format other than that specified using the DF option method. This method takes precedence over the DF `/convert:keyword` compiler option method.

You can use **OPTIONS** statements to specify the appropriate floating-point formats (in memory and in unformatted files) instead of using the corresponding DF command qualifiers. For example, to use VAX F_floating and G_floating as the unformatted file format, specify the following **OPTIONS** statement:

```
OPTIONS /CONVERT=VAXG
```

Because this method affects all unit numbers, you cannot read data in one format and write it in another format unless you use it in combination with the logical name method or the **OPEN** statement **CONVERT** keyword method to specify a different format for a particular unit number.

For more information, see the **OPTIONS** statement.

Compiler Option /convert Method

You can only specify one numeric format for all unformatted file unit numbers using the compiler option `/convert` method unless you also use one (or both) of the previous methods. You specify the numeric format at compile time and must compile all routines under the same `/convert:keyword` compiler option, which is listed under the Compatibility category in the Developer Studio, Fortran tab. You could use the same source program and compile it using different DF commands (or the equivalent in Developer Studio) to create multiple executable programs that each read a certain format.

If you specify other methods, they take precedence over this method. For instance, you might use the environment variable or **OPEN** statement **CONVERT** keyword method to specify each unit number that will use a format different than that specified using the DF `/convert:keyword` compiler option

method for all other unit numbers.

For example, the following commands compile program file.for to use VAX D_floating (and F_floating) floating-point data for all unit numbers (unless superseded by one of the other methods). Data is converted between the file format and the little endian memory format (little endian integers, S_float and T_float little endian IEEE floating-point format). The created file, vconvert.exe, can then be run:

```
DF file.for /convert:vaxd /link /out:vconvert.exe
```

Because this method affects all unformatted file unit numbers, you cannot read data in one format and write it in another file format using the `/convert:keyword` compiler option method alone. You can if you use it in combination with the environment variable method or the **OPEN** statement **CONVERT** keyword method to specify a different format for a particular unit number.

Additional Notes on Nonnative Data

The following notes apply to porting nonnative data:

- When porting source code along with the unformatted data, vendors might use different units for specifying the record length (**RECL** specifier) of unformatted files. While formatted files are specified in units of characters (bytes), unformatted files are specified in longword units for DIGITAL Fortran (default) and some other vendors.

To allow you to specify the **RECL** units (bytes or longwords) for unformatted files without source file modification, use the `/assume:byterecl` compiler option (in Developer Studio, this is available in the Project menu Settings item, Fortran tab, Fortran Data category).

The Fortran 90 standard (American National Standard Fortran 90, ANSI X3.198-1991, and International Standards Organization standard ISO/IEC 1539:1991), in Section 9.3.4.5, states: "If the file is being connected for unformatted input/output, the length is measured in processor-dependent units."

- Certain vendors apply different **OPEN** statement defaults to determine the record type. The default record type (**RECORDTYPE**) with DIGITAL Fortran depends on the values for the **ACCESS** and **FORM** specifiers for the **OPEN** statement (also described in the *DIGITAL Fortran Language Reference Manual*).
- Certain vendors use a different identifier for the logical data types, such as hex FF instead of 01 to denote "true."
- Source code being ported might be coded specifically for big endian use.

Using Visual Fortran Tools

This section summarizes the available Visual Fortran tools and describes how to use tools from the Console command line:

- [Overview of Visual Fortran Tools](#)
- [Using Tools From the Command Line](#)
- [Setting Up the Command Console](#)
- [Fortran Compiler and Linker](#)
- [MS-DOS Editor](#)
- [Building Projects with NMAKE](#)
- [Resource Compiler Options](#)
- [Managing Libraries with LIB](#)
- [Editing Files with EDITBIN](#)
- [Examining Files with DUMPBIN](#)
- [Editing Format Descriptors with the Format Editor](#)
- [Profiling Code from the Command Line](#)
- [Fortran Tools: FSPLIT and FPR](#)

Overview of Visual Fortran Tools

The following tools are available in Visual Fortran:

Tool	Description
Integrated Tools in Developer Studio	
Editor	Provides general editing functionality. It recognizes Fortran syntax and can be customized. For more information, see "Text Editor" in the Developer Studio Environment User's Guide .
Debugger	Provides general debug functionality. For more information, see "Debugger" in the Developer Studio Environment User's Guide .
Format Editor (FRMTEDIT)	Presents format code with resulting data layout. For more information, see Editing Format Descriptors with the Format Editor .
Module Wizard (MODWIZ)	Simplifies the use of Component Object Model (COM) and Automation (OLE Automation) objects with Fortran. For more information, see Using COM and Automation Objects .
Profiler (PROFILE, PLIST, and PREP)	Determines unexecuted code or indicates where an application is spending most of its time. For more information, see Profiling Code from the Command Line .
Resource Editors	Develops user-interface components for projects; for example, to build a dialog box. For more information, see Using Dialogs and see "Resource Editors" in the Developer Studio Environment User's Guide .
Source Browser (BSCMAKE)	Creates an information file with details about the symbols in your program. The browse window displays this information and lets you move among instances of the symbols in your source code. For more information, see Source Browser and the /browser option.
Additional Tools¹	

Linker (LINK)	Lets you link object files and libraries, creating 32-bit executable images or DLLs. For more information, see Using the Compiler and Linker from the Command Line and Compiler and Linker Options .
Librarian (LIB)	Lets you manage object libraries, create import libraries to reference exported symbol definitions used when you build Dynamic Link Libraries (DLLs), and extract library members. For more information, see Managing Libraries with LIB .
Microsoft Binary File Dumper (DUMPBIN)	Displays various information from .obj, .exe, and .libs files. For more information, see Examining Files with DUMPBIN .
Microsoft Binary File Editor (EDITBIN)	Lets you modify execution characteristics of a program. For more information, see Editing Files with EDITBIN .
BitViewer (BITVIEW)	Lets you view the binary representation of real numbers in single and double format. For more information, see Viewing Floating-Point Representations with BitViewer .
CVTRES	Lets you convert binary resource files (.res) to linkable object (.obj) files. For more information, see CVTRES below.
DDESpy (DDESPY)	Lets you monitor Dynamic Data Exchange (DDE) activity between processes. For more information, see "Windows Utilities" in the Developer Studio Environment User's Guide
FPP	Lets you preprocess Fortran files; similar to the C preprocessor (CPP). For more information, see /fpp .
FPR	Lets you transform files formatted according to Fortran's carriagecontrol conventions into files formatted according to line printer conventions. For more information, see Fortran Tools: FSPLIT and FPR .
FSPLIT and FSPLIT90	Lets you split a multi-routine Fortran file into individual files. FSPLIT works on FORTRAN 77 files, while FSPLIT90 works on Fortran 90 files. For more information, see Fortran Tools: FSPLIT and FPR .
Microsoft Program Maintenance Utility (NMAKE)	Lets you build projects based on commands contained in a description (makefile) file. For more information, see Building Projects with NMAKE .
OLE Object Viewer (OLEVIEW)	Lets you browse, configure, test, and activate any COM class on your system; also called the OLEViewer. For more information, see OLE Object Viewer below.
PView (PVIEW)	Lets you examine and modify processes and threads running on your system. For more information, see PView and WinDiff and "Windows Utilities" in the Developer Studio Environment User's Guide
Resource Compiler (RC)	Compiles various resources so they can be included in an image. For more information, see Resource Compiler Command Line .
Running Object Table Viewer (IROTVIEW)	Lets you view the contents of the OLE Running Object Table. For more information, see Running Object Table Viewer below.
Spy++ (SPYXX)	Lets you monitor windows messages. For more information, see "Windows Utilities" in the Developer Studio Environment User's Guide
WinDiff (WINDIFF)	Lets you graphically compare the contents of two files or two directories. For more information, see PView and WinDiff and "Windows Utilities" in the Developer Studio Environment User's Guide
ZoomIn (ZOOMIN)	Lets you capture and enlarge an area of the Windows desktop. For more

information, see "Windows Utilities" in the Developer Studio Environment User's Guide

¹ To access these tools from a command window, the DIGITAL Visual Fortran environment must be initialized. During installation, the environment variables are initialized by default. If you chose not to install these variables during installation, see [Using the Command-Line Interface](#) for information on how to proceed.

Miscellaneous Tool Information

This section briefly describes tools that are not described in detail elsewhere in the documentation.

CVTRES

Binary resource files (.res) cannot be linked. CVTRES lets you convert a binary resource file into a linkable object file (.obj). For example:

```
cvtres /out:test.obj test.res
```

Running Object Table Viewer (IROTVIEW)

The Running Object Table Viewer lets you view the contents of the OLE Running Object Table (ROT). This table contains information about ActiveX™ and OLE objects currently existing in memory.

OLE Object Viewer (OLEVIEW)

The OLE/COM Object Viewer (OLEViewer) lets you do the following:

- Browse, in a structured way, all of the Component Object Model (COM) classes installed on your machine.
- See the registry entries for each class in an easy-to-read format.
- Configure any COM class (including Java-based classes) on your system. This includes Distributed COM activation and security settings.
- Configure system-wide COM settings, including enabling or disabling Distributed COM.
- Test any COM class by double-clicking its name. The list of interfaces that class supports will be displayed. Double-clicking an interface entry allows you to invoke a viewer that will "exercise" that interface.
- Activate COM classes locally or remotely. Use this to test Distributed COM setups.
- View type library contents. Use this to figure out what methods, properties, and events an ActiveX™ Control supports.
- Copy a properly formatted OBJECT tag to the clipboard for inserting into an HTML document.

The OLEViewer supports plug-in interface viewers. The code for the interface viewers is included in

OLEView (in IVIEWERS.DLL).

Using Tools from the Command Line

Although Visual Fortran comes with an integrated Windows-based development environment called Microsoft Developer Studio, you can still use many software tools directly from the command line. If you prefer to use a text-based environment, you can build your programs or libraries in the console (such as the F90 window in the Visual Fortran program folder), a command-line operating environment similar to MS-DOS provided by Windows 95 and Windows NT. However, to get the benefit of components that you cannot use from the command line, you may want to do some of your work from the console, and some of it in Developer Studio.

When you run an application for Windows (such as the Format Statement Editor) from the command line, Windows recognizes that the program does not execute within the command window and acts accordingly.

You can tell Windows to run a program with its own resources by using the START command. For example, to run the Library Manager as a separate task, the command is:

```
START LIB.EXE
```

Visual Fortran contains an extensive electronic reference called InfoViewer that includes the Visual Fortran online documentation and a search engine. You cannot access these books from outside Developer Studio. If you want to use Visual Fortran from the command line, you can still use Developer Studio to display InfoViewer, and task switch between it and the console.

The following related sections discuss command-line tools:

- [Setting Up the Command Console](#)
- [Fortran Compiler and Linker](#)
- [MS-DOS Editor](#)
- [Building Projects with NMAKE](#)
- [Resource Compiler Options](#)
- [Managing Libraries with LIB](#)
- [Editing Files with EDITBIN](#)
- [Examining Files with DUMPBIN](#)
- [Editing Format Descriptors with the Format Editor](#)
- [Profiling Code from the Command Line](#)
- [Fortran Tools: FSPLIT and FPR](#)

For a summary of all Visual Fortran tools, see [Overview of Visual Fortran Tools](#).

Setting Up the Command Console

To start the command console, open the Start menu and select MS-DOS® Prompt from the Programs submenu.

Similarly, Visual Fortran provides a command window with the appropriate environment variables already set. To start the Visual Fortran command window, open the Start menu and select Visual

Fortran from the Programs submenu. Select the F90 icon.

The console window provides a similar working environment to that provided by running a version of MS-DOS instead of Windows NT or Windows 95. You can use any command recognized by MS-DOS in the Windows NT console, plus some additional commands.

Because the command console runs within the context of Windows, you get the additional benefit that you can easily switch between the command console and other applications for Windows. If you want, you can even have multiple instances of the command console open at once. When you are finished working in a command console window, use the EXIT command to close the window and end the session.

► **To create a custom Start menu entry:**

1. Select Settings, then Taskbar from the Start menu. Choose the Start Menu Programs tab from the Taskbar dialog box and click Add.
2. The Create Shortcut dialog box opens. Enter the name and location of the executable you want to add to the Start menu (in this case the MS-DOS command prompt).

You can either type the path to your COMMAND.COM for Windows 95 systems or CMD.EXE for Windows NT systems (for example, C:\WINDOWS\COMMAND.COM) into the text box or use the browser by clicking on the Browse button, then selecting the COMMAND file from the WINDOWS folder. When you are done, click Next.

3. The Select Program Folder dialog box opens and you are shown the folders in your computer. Select the Visual Fortran Folder and choose Next.
4. The Select Title for the Program dialog box opens. Type the name you want to appear on the Fortran Start menu (for example, "Fortran Console") or leave it as the default MS-DOS Prompt. Click on Finish.
5. You are returned to the Start Menu tab. Choose Advanced.
6. The Explorer browser opens with a list of Start programs. Choose Visual Fortran. The Fortran menu expands. Select the MS-DOS Prompt by whatever name you have given it.
7. Choose Properties from the File menu and select the Program tab.
8. Type your working directory into the Working text box (for example, c:\work). Click Change Icon.
9. Choose a new icon by selecting one of the displayed choices. Click OK on the Properties dialog, close the Explorer browser, and choose OK on the Start Menu tab.

For more information, see:

- [Configuring the Command Console Window](#)
- [Setting Search Paths in the Console](#)

Configuring the Command Console Window

When you start a session in the command console, a window containing the command interpreter opens. The resources available, as well as the size and behavior of the window, are initially set by the operating system, but you can change these properties, including:

- Whether the command console takes over the entire screen or is presented in a window.

- The typeface and type size used to display text in the command console.
- The size of the command console text buffer and the position of the command console window if it is presented in a window.
- The colors used to display text in the the command console.
- The size of the command history buffer used to store commands that scroll out of view.
- The amount of each type of memory that is available to programs running in the command console.
- Special configuration files to be run when the console session begins.

The controls that you use to make these adjustments depend upon which version of Windows you are using. Both Windows NT and Windows 95 provide a way to specify configuration settings for all subsequent sessions with the command console.

In Windows NT, use the control panel.

In Windows 95 you use the Properties dialog box to set all of the initial and operating conditions for the command console. With the command window open, click the right mouse button at the top of the window. A drop-down list appears. Choose Properties. From the Properties dialog, set up the console display as you like.

Setting Search Paths in the Console

When the command console session begins, the search paths for libraries, module files, and so forth are those set for your user account on the PC. In Windows 95, these paths are initially specified in the AUTOEXEC.BAT file that is read when the computer is booted. By default, Windows NT uses a file called AUTOEXEC.NT to perform initialization of console sessions, but you can specify your own initialization file for the command console with the PIF Editor. (See your Windows NT manual for more about the PIF Editor.)

You can use the SET command to change these search paths manually within the console session, but your changes will only be in effect during that session. If you need to specify certain path changes each time you begin a console session, you can put the SET commands into a batch file and run it when you begin a session. The Setup program provides a batch file called DFVARS.BAT for this purpose. You can add your SET commands to this file and run it at the start of each session.

You can run DFVARS.BAT:

- Each time you begin a session on Windows 95 systems, by specifying it in the Program tab of the Properties dialog box for the console icon.
- On Windows NT systems, you can specify it as the initialization file with the PIF Editor.

The instructions specify the PATH, INCLUDE, and LIB environment variables. For example, the lines in the batch file that sets the INCLUDE environment variable include:

```
set LIB=%DFcdrom%\DF\LIB;%DFcdrom%\VC\LIB;%LIB%
```

The batch file inserts the directories used by Visual Fortran at the beginning of the existing paths. Because these directories appear first, they are searched before any directories in the path lists provided by Windows. This is especially important if the existing path includes directories with files

having the same names as those needed by Visual Fortran.

As described in [Using the Command-Line Interface](#) in *Getting Started*, the Visual Fortran F90 command window sets these variables for you automatically. To activate this command window, select the F90 icon in the Visual Fortran program folder.

Fortran Compiler and Linker

The DF (or FL32) command is the driver for running the compiler and linker. You can either compile and link your projects in one step with DF, or compile them with DF and then link them with LINK. You can also use LINK to build libraries of object modules. Each of these commands provides syntax instructions at the command line if you request it with the `/?` or `/help` option. For more information about the DF and LINK commands, see:

- [Using the Compiler and Linker from the Command Line](#)
- [Compiler and Linker Options](#)

MS-DOS Editor

You can use the MS-DOS Editor (EDIT.EXE) or any text editor to create your source programs, but you will not be able to perform the specialized functions built into Microsoft Developer Studio such as setting bookmarks or multi-file searches, and your source code will not be displayed with syntax coloring.

You invoke the MS-DOS Editor by typing EDIT followed by the name of the file you want to edit; for example, EDIT test.f90.

Building Projects with NMAKE

Some projects require an extensive set of build instructions to ensure that each component is built with the appropriate options. With Microsoft Developer Studio you can specify build instructions by routine, and you can have separate sets of instructions for the debug and release builds of a project. In Microsoft Developer Studio, you select these options in a set of dialog boxes. For information on creating (exporting) a makefile from Developer Studio, see [The Project Makefile](#).

When you build projects from the command line, you can put your build instructions into a special build file, and run the build process with NMAKE, the Microsoft Program Maintenance Utility. Other command-line building methods include using indirect command files (see [DF Indirect Command File Use](#)) and .BAT files.

The Microsoft Program Maintenance Utility (NMAKE.EXE) is a 32-bit tool that runs in Windows. This section discusses the following:

- [Running NMAKE](#)
- [Contents of a Makefile](#)
- [Description Blocks](#)
- [Commands in a Makefile](#)
- [Inline Files in a Makefile](#)

- [Macros and NMAKE](#)
- [NMAKE Inference Rules](#)
- [Dot Directives](#)
- [Makefile Preprocessing](#)

Running NMAKE

NMAKE builds only specified *targets* or, if none is specified, the first target in the makefile is used. The first makefile target can be a pseudotarget that builds other targets. NMAKE uses makefiles specified with the /F option. If the /F option is not specified, it uses the MAKEFILE file in the current directory. If no makefile is specified, it uses inference rules to build command-line *targets*.

The syntax for NMAKE is:

```
NMAKE [option...] [macros...] [targets...] [@commandfile...]
```

The *commandfile* text file contains command-line input. Other input can precede or follow *@commandfile*. A path is permitted. In *commandfile*, line breaks are treated as spaces. Enclose macro definitions in quotation marks if they contain spaces.

For more information:

- On targets, see [Description Blocks](#)
- On macros, see [Macros and NMAKE](#)
- On options, see [NMAKE Options](#)

NMAKE Options

NMAKE options are described in the following sections. Options are preceded by either a slash (/) or a dash (-) and are not case sensitive. Use **!CMDSWITCHES** (described in [Makefile Preprocessing Directives](#)) to change option settings in a makefile or in TOOLS.INI.

This section describes the following topics:

- [NMAKE Option /A](#)
- [NMAKE Option /B](#)
- [NMAKE Option /C](#)
- [NMAKE Option /D](#)
- [NMAKE Option /E](#)
- [NMAKE Option /F](#)
- [NMAKE Option /HELP](#)
- [NMAKE Option /I](#)
- [NMAKE Option /K](#)
- [NMAKE Option /N](#)
- [NMAKE Option /NOLOGO](#)
- [NMAKE Option /P](#)
- [NMAKE Option /Q](#)
- [NMAKE Option /R](#)
- [NMAKE Option /S](#)
- [NMAKE Option /T](#)
- [NMAKE Option /X](#)

- [TOOLS.INI and NMAKE](#)
- [Exit Codes from NMAKE](#)

NMAKE Option /A

Forces build of all evaluated targets, even if not out-of-date with respect to dependents. Does not force build of unrelated targets.

NMAKE Option /B

Forces build even if timestamps are equal. Recommended for very fast systems (resolution of two seconds or less).

NMAKE Option /C

Suppresses default output, including nonfatal NMAKE errors or warnings, timestamps, and NMAKE copyright message. Suppresses warnings issued by the [/K](#) option.

NMAKE Option /D

Displays timestamps of each evaluated target and dependent and a message when a target does not exist. Useful with the [/P](#) option for debugging a makefile. Use **!CMDSWITCHES** (described in [Makefile Preprocessing Directives](#)) to set or clear the [/D](#) option for part of a makefile.

NMAKE Option /E

Causes environment variables to override makefile macro definitions.

NMAKE Option /F

The option [/F filename](#) specifies *filename* as a makefile. Spaces or tabs can precede *filename*. Specify the [/F](#) option once for each makefile. To supply a makefile from standard input, specify a dash (-) for *filename*. End keyboard input with either F6 or CTRL+Z.

NMAKE Option /HELP

The option [/HELP](#) or [/?](#) displays a brief summary of NMAKE command-line syntax.

NMAKE Option /I

Ignores exit codes from all commands. To set or clear the [/I](#) option for part of a makefile, use **!CMDSWITCHES** (described in [Makefile Preprocessing Directives](#)). To ignore exit codes for part of a makefile, use a dash (-) command modifier or **.IGNORE**. Overrides the [/K](#) option if both are specified.

NMAKE Option /K

Continues building unrelated dependencies if a command returns an error; also issues a warning and returns an exit code of 1. By default, NMAKE halts if any command returns a nonzero exit code. Warnings from the /K option are suppressed by the /C option; the /I option overrides the /K option if both are specified.

NMAKE Option /N

Displays but does not execute commands; preprocessing commands are executed. Does not display commands in recursive NMAKE calls. Useful for debugging makefiles and checking timestamps. To set or clear the /N option for part of a makefile, use **!CMDSWITCHES** (described in [Makefile Preprocessing Directives](#)).

NMAKE Option /NOLOGO

Suppresses the NMAKE copyright message.

NMAKE Option /P

Displays information (macro definitions, inference rules, targets, **.SUFFIXES** list) to standard output, then runs the build. If no makefile or command-line target exists, it displays information only. Use with the /D option to debug a makefile.

NMAKE Option /Q

Checks timestamps of targets; does not run the build. Returns a zero exit code if all are up-to-date and a nonzero exit code if any target is not. Preprocessing commands are executed. Useful when running NMAKE from a batch file.

NMAKE Option /R

Clears the **.SUFFIXES** list and ignores inference rules and macros that are defined in the TOOLS.INI file or that are predefined.

NMAKE Option /S

Suppresses display of executed commands. To suppress display in part of a makefile, use the @ command modifier or **.SILENT**. To set or clear the /S option for part of a makefile, use **!CMDSWITCHES** (described in [Makefile Preprocessing Directives](#)).

NMAKE Option /T

Updates timestamps of command-line targets (or first makefile target) and executes preprocessing commands but does not run the build.

NMAKE Option /X

The option `/X filename` sends NMAKE error output to *filename* instead of standard error. Spaces or tabs can precede *filename*. To send error output to standard output, specify a dash (-) for *filename*. Does not affect output from commands to standard error.

TOOLS.INI and NMAKE

NMAKE reads TOOLS.INI before it reads makefiles, unless the `/R` option is used. It looks for TOOLS.INI first in the current directory and then in the directory specified by the `INITEnvironment` variable. The section for NMAKE settings in the initialization file begins with `[NMAKE]` and can contain any makefile information. Specify a comment on a separate line beginning with a semicolon (;) or a number sign (#).

Exit Codes from NMAKE

By default, NMAKE halts if any command returns a nonzero exit code. The `/I` option causes NMAKE to ignore exit codes. Warnings from the `/K` option are suppressed by the `/C` option; the `/I` option overrides the `/K` option if both are specified. The following table lists the exit codes.

Code	Meaning
0	No error (possibly a warning)
1	Incomplete build (issued only when the <code>/K</code> option is used)
2	Program error, possibly due to one of the following: <ul style="list-style-type: none"> • A syntax error in the makefile • An error or exit code from a command • An interruption by the user
4	System error--out of memory
255	Target is not up-to-date (issued only when the <code>/Q</code> option is used)

Contents of a Makefile

A makefile contains:

- Description blocks
- Commands
- Macros
- Inference Rules
- Dot Directives
- Preprocessing Directives

Other features of a makefile include wildcards, long filenames, comments, and special characters.

Wildcards and NMAKE

NMAKE expands filename wildcards (* and ?) in dependency lines. A wildcard specified in a

command is passed to the command; NMAKE does not expand it.

Long Filenames in a Makefile

Enclose long filenames in double quotation marks, as follows:

```
all : "VeryLongFileName.exe"
```

Comments in a Makefile

Precede a comment with a number sign (#). NMAKE ignores text from the number sign to the next newline character. The following are examples of comments:

```
# Comment on line by itself
OPTIONS = /MAP # Comment on macro definition line

all.exe : one.obj two.obj # Comment on dependency line
    link one.obj two.obj
# Comment in commands block
#   copy *.obj \objects # Command turned into comment
    copy one.exe \release

.obj.exe: # Comment on inference rule line
    link $<

my.exe : my.obj ; link my.obj # Error: cannot comment this
# Error: # must be the first character
.obj.exe: ; link $< # Error: cannot comment this
```

To specify a literal number sign, precede it with a caret (^), as follows:

```
DEF = ^#define #Macro representing a Fortran compiler directive
```

Special Characters in a Makefile

To use an NMAKE special character as a literal character, place a caret (^) in front of it. NMAKE ignores carets that precede other characters. The special characters are:

```
;;#()$^\{ }!@ -
```

A caret within a quoted string is treated as a literal caret character. A caret at the end of a line inserts a literal newline character in a string or macro.

In macros, a backslash followed by a newline character is replaced by a space.

In commands, a percent symbol (%) is a file specifier. To represent a percent symbol (%) literally in a command, specify a double percent sign (%%) in place of a single one. In other situations, NMAKE interprets a single % literally, but it always interprets a double %% as a single %. Therefore, to represent a literal %%, specify either three percent signs, %%%, or four percent signs, %%%%. .

To use the dollar sign (\$) as a literal character in a command, specify two dollar signs (\$\$); this method can also be used in other situations where ^\$ also works.

Description Blocks

A description block is a dependency line optionally followed by a commands block:

```
targets... : dependents...  
           commands...
```

A dependency line specifies one or more targets and zero or more dependents. A target must be at the start of the line. Separate targets from dependents by a colon (:); spaces or tabs are allowed. To split the line, use a backslash (\) after a target or dependent. If a target does not exist, has an earlier timestamp than a dependent, or is a pseudotarget, NMAKE executes the commands. If a dependent is a target elsewhere and does not exist or is out-of-date with respect to its own dependents, NMAKE updates the dependent before updating the current dependency.

For more information, see:

- [Targets](#)
- [Pseudotargets](#)
- [Multiple Targets](#)
- [Cumulative Dependencies](#)
- [Targets in Multiple Description Blocks](#)
- [Dependents](#)

Targets

In a dependency line, specify one or more targets, using any valid filename or pseudotarget. Separate multiple targets with one or more spaces or tabs. Targets are not case sensitive. Paths are permitted with filenames. A target cannot exceed 256 characters. If the target preceding the colon is a single character, use a separating space; otherwise, NMAKE interprets the letter-colon combination as a drive specifier.

Pseudotargets

A pseudotarget is a label used in place of a filename in a dependency line. It is interpreted as a file that does not exist and so is out-of-date. NMAKE assumes a pseudotarget's timestamp is the most recent of all its dependents; if it has no dependents, the current time is assumed. If a pseudotarget is used as a target, its commands are always executed.

A pseudotarget used as a dependent must also appear as a target in another dependency; however, that dependency does not need to have a commands block.

Pseudotarget names follow the filename syntax rules for targets. However, if the name does not have an extension (that is, does not contain a period), it can exceed the 8-character limit for filenames and can be up to 256 characters long.

Multiple Targets

NMAKE evaluates multiple targets in a single dependency as if each were specified in a separate description block as follows:

This...	...is evaluated as this
bounce.exe leap.exe : jump.obj echo Building...	bounce.exe : jump.obj echo Building... leap.exe : jump.obj echo Building...

Cumulative Dependencies

Dependencies are cumulative in a description block if a target is repeated as follows:

This...	...is evaluated as this
bounce.exe : jump.obj bounce.exe : up.obj echo Building bounce.exe...	bounce.exe : jump.obj up.obj echo Building bounce.exe...

Multiple targets in multiple dependency lines in a single description block are evaluated as if each were specified in a separate description block, but targets that are not in the last dependency line do not use the commands block as follows:

This...	...is evaluated as this
bounce.exe leap.exe : jump.obj bounce.exe climb.exe : up.obj echo Building...	bounce.exe : jump.obj up.obj echo Building bounce.exe... climb.exe : up.obj echo Building climb.exe... leap.exe : jump.obj # invokes an inference rule

Targets in Multiple Description Blocks

To update a target in more than one description block using different commands, specify two consecutive colons (::) between targets and dependents. For example:

```
target.lib :: one.f90 two.f90 three.f90
  df one.f90 two.f90 three.f90
  lib target one.obj two.obj three.obj
target.lib :: four.c five.c
  df /c four.for five.for
  lib target four.obj five.obj
```

If a target is specified with a colon (:) in two dependency lines in different locations, and if commands appear after only one of the lines, NMAKE interprets the dependencies as if adjacent or combined. It does not invoke an inference rule for the dependency that has no commands, but instead assumes that the dependencies belong to one description block and executes the commands specified with the other dependency as follows:

This...	...is evaluated as this
bounce.exe : jump.obj echo Building bounce.exe... bounce.exe : up.obj	bounce.exe : jump.obj up.obj echo Building bounce.exe...

This effect does not occur if :: is used as follows:

This...	...is evaluated as this
bounce.exe :: jump.obj	bounce.exe : jump.obj
echo Building bounce.exe...	echo Building bounce.exe...
bounce.exe :: up.obj	bounce.exe : up.obj
	# invokes an inference rule

Dependents

In a dependency line, specify zero or more dependents after the colon (:), or double colon (::), using any valid filename or pseudotarget. Separate multiple dependents with one or more spaces or tabs. Dependents are not case sensitive. Paths are permitted with filenames.

Inferred Dependents

An inferred dependent is derived from an inference rule and is evaluated before explicit dependents. If an inferred dependent is out-of-date with respect to its target, NMAKE invokes the commands block for the dependency. If an inferred dependent does not exist or is out-of-date with respect to its own dependents, NMAKE first updates the inferred dependent. For more information, see [Inference Rules](#).

Search Paths for Dependents

Each dependent has an optional search path, specified as follows:

```
{directory[:directory...]}dependent
```

NMAKE looks for a dependent first in the current directory, and then in directories in the order specified. A macro can specify part or all of a search path. Enclose directory names in braces ({ }); separate multiple directories with a semicolon (;). No spaces or tabs are allowed.

Commands in a Makefile

A description block or inference rule specifies a block of commands to run if the dependency is out-of-date. NMAKE displays each command before running it, unless the /S option, **.SILENT**, **!CMDSWITCHES**, or @ is used. NMAKE looks for a matching inference rule if a description block is not followed by a commands block.

A commands block contains one or more commands, each on its own line. No blank line can appear between the dependency or rule and the commands block. However, a line containing only spaces or tabs can appear; this line is interpreted as a null command and no error occurs. Blank lines are permitted between command lines.

A command line begins with one or more spaces or tabs. A backslash (\) followed by a newline character is interpreted as a space in the command; use a backslash at the end of a line to continue a command onto the next line. NMAKE interprets the backslash literally if any other character, including a space or tab, follows the backslash.

A command preceded by a semicolon (;) can appear on a dependency line or inference rule, whether or not a commands block follows:


```
project.obj : project.f90 ; df /c project.f90
```

For more information see:

- [Command Modifiers in NMAKE](#)
- [Filename-Parts Syntax in NMAKE](#)

Command Modifiers in NMAKE

You can specify one or more command modifiers preceding a command, optionally separated by spaces or tabs. As with commands, modifiers must be indented. The following table lists the command modifiers:

Modifier	Action
@ <i>command</i>	Prevents display of the command. Display by commands is not suppressed. By default, NMAKE echoes all executed commands. Use the /S option to suppress display for the entire makefile; use .SILENT to suppress display for part of the makefile.
-[<i>number</i>] <i>command</i>	Turns off error checking for <i>command</i> . By default, NMAKE halts when a command returns a nonzero exit code. If <i>-number</i> is used, NMAKE stops if the exit code exceeds <i>number</i> . Spaces or tabs cannot appear between the dash and <i>number</i> ; at least one space or tab must appear between <i>number</i> and <i>command</i> . Use the /I option to turn off error checking for the entire makefile; use .IGNORE to turn off error checking for part of the makefile.
! <i>command</i>	Executes <i>command</i> for each dependent file if <i>command</i> uses \$** (all dependent files in the dependency) or \$? (all dependent files in the dependency with a later timestamp than the target).

Filename-Parts Syntax in NMAKE

Filename-parts syntax in commands represents components of the first dependent filename (which may be an implied dependent). Filename components are the file's drive, path, base name, and extension as specified, not as it exists on disk.

Use %s to represent the complete filename. Use %|[*parts*]F to represent parts of the filename, where *parts* can be zero or more of the following letters, in any order:

Letter	Description
No letter	Complete name
d	Drive
p	Path
f	File base name
e	File extension

Inline Files in a Makefile

An inline file contains text you specify in the makefile. Its name can be used in commands as input (for example, a LINK command file), or it can pass commands to the operating system. The file is created on disk when a command that creates the file is run.

For more information, see:

- [Specifying an Inline File in Makefiles](#)
- [Creating Inline File Text](#)
- [Reusing Inline Files in Makefiles](#)
- [Multiple Inline Files](#)

Specifying an Inline File in Makefiles

The syntax for specifying an inline file in a command is:

```
<<[filename]
```

Specify two angle brackets (<<) in the command where the filename is to appear. The angle brackets cannot be a macro expansion. When the command is run, the angle brackets are replaced by *filename*, if specified, or by a unique NMAKE-generated name. If specified, *filename* must follow the angle brackets without a space or tab. A path is permitted. No extension is required or assumed.

If *filename* is specified, the file is created in the current or specified directory, overwriting any existing file by that name; otherwise, it is created in the TMP directory (or the current directory, if the TMP environment variable is not defined). If a previous *filename* is reused, NMAKE overwrites the previous file.

Creating Inline File Text in Makefiles

The syntax to create the content of an inline file is:

```
inlinetext
```

```
.  
. .  
.
```

```
<<[KEEP | NOKEEP]
```

Specify *inlinetext* on the first line after the command. Mark the end with double brackets at the beginning of a separate line. The file contains all *inlinetext* before the delimiting brackets. The *inlinetext* can have macro expansions and substitutions, but not directives or makefile comments. Spaces, tabs, and newline characters are treated literally.

Inline files are temporary or permanent. A temporary file exists for the duration of the session and can be reused by other commands. Specify **KEEP** after the closing angle brackets to retain the file after the NMAKE session; an unnamed file is preserved on disk with the generated filename. Specify **NOKEEP** or nothing for a temporary file. **KEEP** and **NOKEEP** are not case sensitive.

Reusing Inline Files in Makefiles

To reuse an inline file, specify <<*filename* where the file is defined and first used, then reuse *filename* without the angle brackets (<<) later in the same or another command. The command to create the inline file must run before all commands that use the file.

Multiple Inline Files

A command can create more than one inline file. The syntax to do this is:

```
command << <<  
inlinetext  
<<[KEEP | NOKEEP]  
inlinetext  
<<[KEEP | NOKEEP]
```

For each file, specify one or more lines of inline text followed by a closing line containing the delimiter. Begin the second file's text on the line following the delimiting line for the first file.

Macros and NMAKE

Macros replace a particular string in the makefile with another string. Using macros, you can create a makefile that can build different projects, specify options for commands, or set environment variables. You can define your own macros or use NMAKE's predefined macros.

For more information, see:

- [Defining an NMAKE Macro](#)
- [Special Characters in NMAKE Macros](#)
- [Null and Undefined NMAKE Macros](#)
- [Where to Define Macros](#)
- [Precedence in Macro Definitions](#)
- [Using an NMAKE Macro](#)
- [Macro Substitution](#)
- [Special NMAKE Macros](#)

Defining an NMAKE Macro

Use the following syntax to define a macro:

```
macroname=string
```

The *macroname* is a combination of letters, digits, and underscores (_) up to 1024 characters, and is case sensitive. The *macroname* can contain an invoked macro. If *macroname* consists entirely of an invoked macro, the macro being invoked cannot be null or undefined.

The *string* can be any sequence of zero or more characters. A null string contains zero characters or only spaces or tabs. The *string* can contain a macro invocation.

Special Characters in NMAKE Macros

A number sign (#) after a definition specifies a comment. To specify a literal number sign in a macro, use a caret (^), as in ^#.

A dollar sign (\$) specifies a macro invocation. To specify a literal \$, use \$\$.

To extend a definition to a new line, end the line with a backslash (\). When the macro is invoked, the backslash plus newline character is replaced with a space. To specify a literal backslash at the end of the line, precede it with a caret (^), or follow it with a comment specifier (#).

To specify a literal newline character, end the line with a caret (^), as in:

```
CMDS = cls^  
dir
```

Null and Undefined NMAKE Macros

Both null and undefined macros expand to null strings, but a macro defined as a null string is considered defined in preprocessing expressions. To define a macro as a null string, specify no characters except spaces or tabs after the equal sign (=) in a command line or command file, enclose the null string or definition in double quotation marks ("). To undefine a macro, use **!UNDEF**.

Where to Define Macros

You can define macros in a makefile command line, or command file.

In a makefile, each macro definition must appear on a separate line and cannot start with a space or tab. Spaces or tabs around the equal sign (=) are ignored. All *string* characters are literal, including surrounding quotation marks and embedded spaces.

In a command line or command file, spaces and tabs delimit arguments and cannot surround the equal sign. If *string* has embedded spaces or tabs, enclose either the string itself or the entire macro in double quotation marks (").

Precedence in Macro Definitions

If a macro is multiply defined, NMAKE uses the highest-precedence definition:

1. A macro defined on the command line
2. A macro defined in a makefile or include file
3. An inherited environment-variable macro
4. A predefined macro, such as **FOR** and **RC**

Use the /E option to cause macros inherited from environment variables to override makefile macros with the same name. Use **!UNDEF** to override a command line.

Using an NMAKE Macro

To use a macro, enclose its name in parentheses preceded by a dollar sign (\$):

```
$(macroname)
```

No spaces are allowed. The parentheses are optional if *macroname* is a single character. The definition string replaces **\$(*macroname*)**; an undefined macro is replaced by a null string.

Macro Substitution

To substitute text within a macro, use the following syntax:

```
$(macroname:string1=string2)
```

When *macroname* is invoked, each occurrence of *string1* in its definition string is replaced by *string2*. Macro substitution is case sensitive and is literal; *string1* and *string2* cannot invoke macros. Substitution does not modify the original definition. You can substitute text in any predefined macro except \$\$@.

No spaces or tabs precede the colon; any after the colon are interpreted as literal. If *string2* is null, all occurrences of *string1* are deleted from the macro's definition string.

Special NMAKE Macros

NMAKE provides several special macros to represent various filenames and commands. One use for some of these macros is in the predefined inference rules. Like all macros, the macros provided by NMAKE are case sensitive.

This section discusses:

- [Filename Macros](#)
- [Recursion Macros](#)
- [Command Macros and Options Macros](#)
- [Environment-Variable Macros](#)

Filename Macros

Filename macros are predefined as filenames specified in the dependency (not full filename specifications on disk). These macros do not need to be enclosed in parentheses when invoked; specify only a \$ as shown.

Macro	Meaning
\$@	Current target's full name (path, base name, extension), as currently specified.
\$\$@	Current target's full name (path, base name, extension), as currently specified. Valid only as a dependent in a dependency.
\$*	Current target's path and base name minus file extension.
\$**	All dependents of the current target.
\$?	All dependents with a later timestamp than the current target.
\$<	Dependent file with a later timestamp than the current target. Valid only in commands in inference rules.

To specify part of a predefined filename macro, append a macro modifier and enclose the modified macro in parentheses.

Modifier	Resulting filename part
D	Drive plus directory
B	Base name

F	Base name plus extension
R	Drive plus directory plus base name

Recursion Macros

Use recursion macros to call NMAKE recursively. Recursive sessions inherit command-line and environment-variable macros. They do not inherit makefile-defined inference rules or **.SUFFIXES** and **.PRECIOUS** specifications. To pass macros to a recursive NMAKE session, either set an environment variable with the SET command before the recursive call or define a macro in the command for the recursive call.

Macro	Definition
MAKE	Command used originally to invoke NMAKE.
MAKEDIR	Current directory when NMAKE was invoked.
MAKEFLAGS	Options currently in effect. Use as /\$(MAKEFLAGS).

Command Macros, Options Macros

Command macros are predefined for Microsoft products. Options macros represent options to these products and are undefined by default. Both are used in predefined inference rules and can be used in description blocks or user-defined inference rules. Command macros can be redefined to represent part or all of a command line, including options. Options macros generate a null string if left undefined.

Microsoft product	Command macro	Defined as	Options macro
Macro Assembler	AS	ml	AFLAGS
Basic Compiler	BC	bc	BFLAGS
C Compiler	CC	cl	CFLAGS
COBOL Compiler	COBOL	cobol	COBFLAGS
C++ Compiler	CPP	cl	CPPFLAGS
C++ Compiler	CXX	cl	CXXFLAGS
Visual Fortran Compiler	FOR	df	FFLAGS
Pascal Compiler	PASCAL	pl	PFLAGS
Resource Compiler	RC	rc	RFLAGS

Environment-Variable Macros

NMAKE inherits macro definitions for environment variables that exist before the start of the session. If a variable was set in the operating-system environment, it is available as an NMAKE macro. The inherited names are converted to uppercase. Inheritance occurs before preprocessing. Use the /E option to cause macros inherited from environment variables to override any macros with the same name in the makefile.

Environment-variable macros can be redefined in the session, but this does not change the corresponding environment variable; to change the variable, use the SET command. Using the SET command to change an environment variable in a session does not change the corresponding macro.

If an environment variable is defined as a string that would be syntactically incorrect in a makefile, no macro is created and no warning is generated. If a variable's value contains a dollar sign (\$), NMAKE interprets it as the beginning of a macro invocation; using the macro can cause unexpected

behavior.

NMAKE Inference Rules

Inference rules supply commands to update targets and to infer dependents for targets. Extensions in an inference rule match a single target and dependent that have the same base name. Inference rules are user-defined or predefined; predefined rules can be redefined.

If an out-of-date dependency has no commands and if **.SUFFIXES** contains the dependent's extension, NMAKE uses a rule whose extensions match the target and an existing file in the current or specified directory. If more than one rule matches existing files, the **.SUFFIXES** list determines which to use; list priority descends from left to right.

If a dependent file doesn't exist and is not listed as a target in another description block, an inference rule can create the missing dependent from another file with the same base name. If a description block's target has no dependents or commands, an inference rule can update the target. Inference rules can build a command-line target even if no description block exists. NMAKE may invoke a rule for an inferred dependent even if an explicit dependent is specified.

For more information, see:

- [Defining an Inference Rule in NMAKE](#)
- [Search Paths in Inference Rules](#)
- [Predefined Inference Rules](#)
- [Inferred Dependents and Rules](#)
- [Precedence in NMAKE Inference Rules](#)

Defining an Inference Rule in NMAKE

To define an inference rule, use the following syntax:

```
.fromext.toext :
commands
```

The *fromext* represents the extension of a dependent file, and *toext* represents the extension of a target file. Extensions are not case sensitive. Macros can be invoked to represent *fromext* and *toext*; the macros are expanded during preprocessing.

The period (.) preceding *fromext* must appear at the beginning of the line. The colon (:) is preceded by zero or more spaces or tabs; it can be followed only by spaces or tabs, a semicolon (;) to specify a command, a number sign (#) to specify a comment, or a newline character. No other spaces are allowed. Commands are specified as in description blocks.

Search Paths in Inference Rules

An inference rule that specifies paths has the following syntax:

```
{frompath},fromext{topath}.toext :
commands
```

An inference rule applies to a dependency only if paths specified in the dependency exactly match the inference-rule paths. Specify the dependent's directory *frompath* and the target's directory in *topath*; no spaces are allowed. Specify only one path for each extension. A path on one extension requires a path on the other. To specify the current directory, use either a period (.) or empty braces ({ }). Macros can represent *frompath* and *topath*; they are invoked during preprocessing.

Predefined Inference Rules

Predefined inference rules use NMAKE-supplied command and option macros:

Rule	Command	Default action
.asm.exe	\$(AS) \$(AFLAGS) \$*.asm	ml \$*.asm
.asm.obj	\$(AS) \$(AFLAGS) /c \$*.asm	ml /c \$*.asm
.c.exe	\$(CC) \$(CFLAGS) \$*.c	cl \$*.c
.c.obj	\$(CC) \$(CFLAGS) /c \$*.c	cl /c \$*.c
.cpp.exe	\$(CPP) \$(CPPFLAGS) \$*.cpp	cl \$*.cpp
.cpp.obj	\$(CPP) \$(CPPFLAGS) /c \$*.cpp	cl /c \$*.cpp
.cxx.exe	\$(CXX) \$(CXXFLAGS) \$*.cxx	cl \$*.cxx
.cxx.obj	\$(CXX) \$(CXXFLAGS) /c \$*.cxx	cl /c \$*.cxx
.bas.obj	\$(BC) \$(BFLAGS) \$*.bas;	bc \$*.bas;
.cbl.exe	\$(COBOL) \$(COBFLAGS) \$*.cbl, \$*.exe;	cobol \$*.cbl, \$*.exe;
.cbl.obj	\$(COBOL) \$(COBFLAGS) \$*.cbl;	cobol \$*.cbl;
.for.exe	\$(FOR) \$(FFLAGS) \$*.for	fl32 \$*.for
.f90.exe	\$(FOR) \$(FFLAGS) \$*.f90	fl32 \$*.f90
.f.exe	\$(FOR) \$(FFLAGS) \$*.f	fl32 \$*.f
.for.obj	\$(FOR) /c \$(FFLAGS) \$*.for	fl32 \$*.for /c
.f90.obj	\$(FOR) /c \$(FFLAGS) \$*.f90	fl32 \$*.f90 /c
.f.obj	\$(FOR) /c \$(FFLAGS) \$*.f	fl32 \$*.f /c
.pas.exe	\$(PASCAL) \$(PFLAGS) \$*.pas	pl \$*.pas
.pas.obj	\$(PASCAL) /c \$(PFLAGS) \$*.pas	pl /c \$*.pas
.rc.res	\$(RC) \$(RFLAGS) /r \$*	rc /r \$*

Inferred Dependents and Rules

NMAKE assumes an inferred dependent for a target if an applicable inference rule exists. A rule applies if:

- *toext* matches the target's extension.
- *fromext* matches the extension of a file that has the target's base name and that exists in the current or specified directory.
- *fromext* is in **.SUFFIXES**; no other *fromext* in a matching rule has a higher **.SUFFIXES** priority.
- No explicit dependent has a higher **.SUFFIXES** priority.

Inferred dependents can cause unexpected side effects. If the target's description block contains commands, NMAKE executes those commands and not the commands in the rule.

Precedence in NMAKE Inference Rules

If an inference rule is multiply defined, the highest-precedence rule applies:

1. An inference rule defined in a makefile; later definitions have precedence.
2. A predefined inference rule.

Dot Directives in Makefiles

Specify dot directives outside a description block, at the start of a line. Dot directives begin with a period (.) and are followed by a colon (:). Spaces and tabs are allowed. Dot directive names are case sensitive and are uppercase.

Directive	Action
.IGNORE :	Ignores nonzero exit codes returned by commands, from the place it is specified to the end of the makefile. By default, NMAKE halts if a command returns a nonzero exit code. To restore error checking, use !CMDSWITCHES (described in Makefile Preprocessing Directives). To ignore the exit code for a single command, use the dash modifier. To ignore exit codes for an entire file, use the /I option.
.PRECIOUS : <i>targets</i>	Preserves <i>targets</i> on disk if the commands to update them are halted; has no effect if a command handles an interrupt by deleting the file. Separate the target names with one or more spaces or tabs. By default, NMAKE deletes a target if a build is interrupted by CTRL+C or CTRL+BREAK. Each use of .PRECIOUS applies to the entire makefile; multiple specifications are cumulative.
.SILENT :	Suppresses display of executed commands, from the place it is specified to the end of the makefile. By default, NMAKE displays the commands it invokes. To restore echoing, use !CMDSWITCHES . To suppress echoing of a single command, use the @ modifier. To suppress echoing for an entire file, use the /S option.
.SUFFIXES : <i>list</i>	Lists extensions for inference-rule matching; predefined as: .exe .obj .asm .c .cpp .cxx .bas .cbl .for .pas .res .rc

To change the **.SUFFIXES** list order or to specify a new list, clear the list and specify a new setting. To clear the list, specify no extensions after the colon:

```
.SUFFIXES :
```

To add additional suffixes to the end of the list, specify:

```
.SUFFIXES : suffixlist
```

where *suffixlist* is a list of the additional suffixes, separated by one or more spaces or tabs. To see the current setting of **.SUFFIXES**, run NMAKE with the **/P** option.

Makefile Preprocessing

You can control the NMAKE session by using preprocessing directives and expressions. Preprocessing instructions can be placed in the makefile. Using directives, you can conditionally process your makefile, display error messages, include other makefiles, undefine a macro, and turn

certain options on or off.

For more information, see:

- [Makefile Preprocessing Directives](#)
- [Expressions in Makefile Preprocessing](#)
- [Makefile Preprocessing Operators](#)
- [Executing a Program in Preprocessing](#)

Makefile Preprocessing Directives

Preprocessing directives are not case sensitive. The initial exclamation point (!) must appear at the beginning of the line. Zero or more spaces or tabs can appear after the exclamation point, for indentation. The following are preprocessing directives:

- **!CMDSWITCHES** {+ | -}*option*...

Turns each *option* listed on or off. Spaces or tabs must appear before the + or - operator; none can appear between the operator and the option letters. Letters are not case sensitive and are specified without a slash (/). To turn some options on and others off, use separate specifications of **!CMDSWITCHES**.

Only D, I, N, and S can be used in a makefile. Changes specified in a description block do not take effect until the next description block. This directive updates the **MAKEFLAGS** recursion macro; changes are inherited during recursion if **MAKEFLAGS** is specified.

- **!ERROR** *text*

Displays *text* in error U1050, then halts NMAKE, even if /K, /I, .IGNORE, **!CMDSWITCHES**, or the dash (-) command modifier is used. Spaces or tabs before *text* are ignored.

- **!MESSAGE** *text*

Displays *text* to standard output. Spaces or tabs before *text* are ignored.

- **!INCLUDE** [<]*filename*[>]

Reads *filename* as a makefile, then continues with the current makefile. NMAKE searches for *filename* first in the specified or current directory, then recursively through directories of any parent makefiles, then, if *filename* is enclosed by angle brackets (<>), in directories specified by the **INCLUDE** macro, which is initially set to the **INCLUDE** environment variable. Useful to pass **.SUFFIXES** settings, **.PRECIOUS**, and inference rules to recursive makefiles.

- **!IF** *constantexpression*

Processes statements between **!IF** and the next **!ELSE** or **!ENDIF** if *constantexpression* evaluates to a nonzero value.

- **!IFDEF** *macroname*

Processes statements between **!IFDEF** and the next **!ELSE** or **!ENDIF** if *macroname* is

defined. A null macro is considered to be defined.

- **!IFDEF *macroname***

Processes statements between **!IFDEF** and the next **!ELSE** or **!ENDIF** if *macroname* is not defined.

- **!ELSE [IF *constantexpression* | IFDEF *macroname* | IFNDEF *macroname*]**

Processes statements between **!ELSE** and the next **!ENDIF** if the prior **!IF**, **!IFDEF**, or **!IFDEF** statement evaluated to zero. The optional keywords give further control of preprocessing.

- **!ELSEIF**

Synonym for **!ELSE IF**.

- **!ELSEIFDEF**

Synonym for **!ELSE IFDEF**.

- **!ELSEIFNDEF**

Synonym for **!ELSE IFNDEF**.

- **!ENDIF**

Marks the end of an **!IF**, **!IFDEF**, or **!IFDEF** block. Any text after **!ENDIF** on the same line is ignored.

- **!UNDEF *macroname***

Undefines *macroname*.

Expressions in Makefile Preprocessing

The **!IF** or **!ELSE IF** *constantexpression* consists of integer constants (in decimal or C-language notation), string constants, or commands. Use parentheses to group expressions. Expressions use C-style signed long integer arithmetic; numbers are in 32-bit two's-complement form in the range -2147483648 to 2147483647.

Expressions can use operators that act on constant values, exit codes from commands, strings, macros, and file-system paths.

Makefile Preprocessing Operators

The **DEFINED** operator is a logical operator that acts on a macro name. The expression **DEFINED** (*macroname*) is true if *macroname* is defined. **DEFINED** in combination with **!IF** or **!ELSE IF** is equivalent to **!IFDEF** or **!ELSE IFDEF**. However, unlike these directives, **DEFINED** can be used in complex expressions using binary logical operators.

The **EXIST** operator is a logical operator that acts on a file-system path. **EXIST** (*path*) is true if *path* exists. The result from **EXIST** can be used in binary expressions. If *path* contains spaces, enclose it in double quotation marks.

Integer constants can use the unary operators for numerical negation (-), one's complement (~), and logical negation (!).

Constant expressions can use the following binary operators:

Operator	Description	Operator	Description
+	Addition		Logical OR
-	Subtraction	<<	Left shift
*	Multiplication	>>	Right shift
/	Division	==	Equality
%	Modulus	!=	Inequality
&	Bitwise AND	<	Less than
	Bitwise OR	>	Greater than
^	Bitwise XOR	<=	Less than or equal to
&&	Logical AND	>=	Greater than or equal to

To compare two strings, use the equality (==) operator and the inequality (!=) operator. Enclose strings in double quotation marks.

Executing a Program in Preprocessing

To use a command's exit code during preprocessing, specify the command, with any arguments, within brackets ([]). Any macros are expanded before the command is executed. NMAKE replaces the command specification with the command's exit code, which can be used in an expression to control preprocessing.

Resource Compiler Options

With Visual Fortran, you can create dialog boxes for an interactive user interface at run-time. For example, you can provide selection lists and scroll bars and the user will not have to type in text strings or numerical control parameters. You can also create custom icons for your QuickWin applications.

Microsoft Developer Studio includes a special dialog editor for creating dialogs and placing the controls within them, and a graphic editor for drawing or importing icons. You must use the dialog editor and graphic editor in Microsoft Developer Studio to design dialogs and icons. Once you have created a dialog or icon, you can compile it from the command line using RC, the resource compiler.

This section describes the preprocessing directives and statements that make up a resource-definition (script) file and how to use RC.EXE to compile your application's resources and add them to an application's executable file.

The resource editor in Microsoft Developer Studio offers easy, time-saving alternatives to the

traditional hand-coded scripts used to create resources. These visual tools create and manage your project's script - you don't need to hand-code scripts. For more information on the working with resources in Microsoft Developer Studio, see the following sections in the [Developer Studio Environment User's Guide](#): Working with Resources, Using the Dialog Editor, and Using the Graphic Editor.

The following topics are covered in this section:

- [Including Resources in an Application](#)
- [Creating a Resource Definition File](#)
- [The RC Command Line](#)

Including Resources in an Application

► To include resources in your application:

1. Use the Microsoft Developer Studio dialog editor or graphic editor to create a resource or each dialog or icon in your application. (For more information, see Working with Resources, Using the Dialog Editor, and Using the Graphic Editor, in the [Developer Studio Environment User's Guide](#).)
2. Create a resource-definition file (also called a script) that describes all resources used by the application.
3. Compile the script into a resource (.RES) file with RC.EXE (RC).
4. Link the compiled resource files into the application's executable file.

You do not use RC to include compiled resources into the executable file or to mark the file as an application for Windows. The linker recognizes the compiled resource files and links them to the executable file.

Creating a Resource-Definition File

After creating individual resource files for your application's dialog box and icon resources, you create a resource-definition file, or script. A script is a text file with the extension .RC.

The script lists every resource in your application and describes some types of resources in great detail. For a resource that exists in a separate file, such as an icon or cursor, the script names the resource and the file that contains it. For some resources, such as a menu, the entire definition of the resource exists within the script.

A script file can contain the following information:

- [Preprocessing Directives](#), which instruct RC to perform actions on the script before compiling. Directives can also assign values to names.
- Statements, which name and describe resources. Statements can be [single-line](#) or [multiline](#) statements.

To view a script file that defines resources for an application, see [Sample Script File](#).

Preprocessing Directives

The directives listed in the following table can be used in the script to instruct RC to perform actions or assign values to names. The syntax and semantics for the RC preprocessor are the same as for the Microsoft C compiler.

Directive	Description
#define	Defines a specified name by assigning it a given value.
#elif	Marks an optional clause of a conditional-compilation block.
#else	Marks the last optional clause of a conditional-compilation block.
#endif	Marks the end of a conditional-compilation block.
#if	Conditionally compiles the script if a specified expression is true.
#ifdef	Conditionally compiles the script if a specified name is defined.
#ifndef	Conditionally compiles the script if a specified name is not defined.
#include	Copies the contents of a file into the resource-definition file.
#undef	Removes the definition of the specified name.

Single-Line Statements

A single-line statement can begin with any of the following keywords:

Keyword	Description
BITMAP	Defines a bitmap by naming it and specifying the name of the file that contains it. (To use a particular bitmap, the application requests it by name.)
CURSOR	Defines a cursor by naming it and specifying the name of the file that contains it. (To use a particular cursor, the application requests it by name.) Custom cursors are not available in Visual Fortran.
FONT	Specifies the name of a file that contains a font.
ICON	Defines an icon by naming it and specifying the name of the file that contains it. (To use a particular icon, the application requests it by name.)
LANGUAGE	Sets the language for all resources up to the next LANGUAGE statement or to the end of the file. When the LANGUAGE statement appears before the BEGIN in an ACCELERATORS , DIALOG , MENU , RCDATA , or STRINGTABLE resource definition, the specified language applies only to that resource.
MESSAGETABLE	Defines a message table by naming it and specifying the name of the file that contains it. The file is a binary resource file generated by the Message Compiler. (The Message Compiler is documented in InfoViewer.)

Multiline Statements

A multiline statement can begin with any of the following keywords:

Keyword	Description
ACCELERATORS	Defines menu accelerator (quick-access) keys.
DIALOG	Defines a template that an application can use to create dialog boxes.
MENU	Defines the appearance and function of a menu.
RCDATA	Defines data resources. Data resources let you include binary data in the executable file.

STRINGTABLE	Defines string resources. String resources are Unicode strings that can be loaded from the executable file.
--------------------	---

Each of these multiline statements allows you to specify zero or more optional statements before the **BEGIN ... END** block that defines the resource. You can specify the following statements:

Statement	Description
CHARACTERISTICS <i>dword</i>	User-defined information about the resource that can be used by tools that read and write resource files. The value appears in the compiled resource file; it is not stored in the executable file and is not used by Windows.
LANGUAGE <i>language, sublanguage</i>	Specifies the language for the resource. The arguments are constants from WINNLS.H.
VERSION <i>dword</i>	User-defined version number for the resource that can be used by tools that read and write resource files. The value appears in the compiled resource file; it is not stored in the executable file and is not used by Windows.

Sample Script File

The following example shows a script file that defines the resources for an application named Shapes:

```
#include "SHAPES.H"

ShapesCursor CURSOR SHAPES.CUR
ShapesIcon ICON SHAPES.ICO

    BEGIN
        POPUP "&Shape"
            BEGIN
                MENUITEM "&Clear", ID_CLEAR
                MENUITEM "&Rectangle", ID_RECT
                MENUITEM "&Triangle", ID_TRIANGLE
                MENUITEM "&Star", ID_STAR
                MENUITEM "&Ellipse", ID_ELLIPSE
            END
        END
    END
```

The **CURSOR** statement names the application's cursor resource ShapesCursor and specifies the cursor file SHAPES.CUR, which contains the image for that cursor. Custom cursors are not available in Visual Fortran.

The **ICON** statement names the application's icon resource ShapesIcon and specifies the icon file SHAPES.ICO, which contains the image for that icon.

The **MENU** statement defines an application menu named ShapesMenu, a pop-up menu with five menu items.

The menu definition, enclosed by the **BEGIN** and **END** keywords, specifies each menu item and the menu identifier that is returned when the user selects that item. For example, the first item on the menu, Clear, returns the menu identifier ID_CLEAR when the user selects it. The menu identifiers are defined in the application header file, SHAPES.H.

Resource Compiler Command Line

To start RC, use the following command-line syntax:

```
RC [options] script-file
```

The *options* argument can include one or more of the following options:

Option	Description
<code>/?</code>	Displays a list of RC command-line options.
<code>/d</code>	Defines a symbol for the preprocessor that you can test with the <code>#ifdef</code> directive.
<code>/fo <i>resname</i></code>	Uses <i>resname</i> for the name of the .RES file.
<code>/h</code>	Displays a list of RC command-line options.
<code>/i <i>directory</i></code>	Causes RC to search the specified <i>directory</i> before searching the directories specified by the INCLUDE environment variable.
<code>/r</code>	Ignored. Provided for compatibility with existing makefiles.
<code>/u</code>	Undefines a symbol.
<code>/v</code>	Causes a display of messages that report on the progress of the compiler.
<code>/x</code>	Prevents RC from checking the INCLUDE environment variable when searching for header files or resource files.

Options are not case sensitive and a dash (-) can be used in place of a forward slash (/). You can combine single-letter options if they do not require additional arguments. For example, the following commands are equivalent:

```
RC /V /X SAMPLE.RC
rc -vx sample.rc
```

The *script-file* argument specifies the name of the resource-definition script that contains the names, types, filenames, and descriptions of the resources to be compiled.

For more information, see:

- [Defining Names for the Resource Preprocessor](#)
- [Naming the Compiled Resource File](#)
- [Searching for Resource Files](#)
- [Adding a Directory to Search](#)
- [Suppressing the INCLUDE Environment Variable](#)
- [Displaying Resource-Compiler Progress Messages](#)
- [Common Resource Statement Arguments](#)

Defining Names for the Resource Preprocessor

You can use the `/d` option to specify conditional compilation in a script, based on whether a name is defined on the RC command line. You can also use the **DEFINE** directive to specify conditional compilation in the file or an include file.

For example, suppose your application has a pop-up menu, the Debug menu, that should appear only with debugging versions of the application. When you compile the application for normal use, the

Debug menu is not included.

The following example shows the statements that can be added to the resource-definition file to define the Debug menu:

```
MainMenu MENU
BEGIN
.
.
.
#ifdef DEBUG
    POPUP "&Debug"
    BEGIN
        MENUITEM "&Memory usage", ID_MEMORY
        MENUITEM "&Walk data heap", ID_WALK_HEAP
    END
#endif
END
```

When compiling resources for a debugging version of the application, you could include the Debug menu by using the following RC command:

```
rc -d DEBUG myapp.rc
```

To compile resources for a normal version of the application -- one that does not include the Debug menu -- you could use the following RC command:

```
rc myapp.rc
```

Naming the Compiled Resource File

By default, when compiling resources, RC names the compiled resource (.RES) file with the base name of the .RC file and places it in the same directory as the .RC file. The following example compiles MYAPP.RC and creates a compiled resource file named MYAPP.RES in the same directory as MYAPP.RC:

```
rc myapp.rc
```

The /fo option lets you give the resulting .RES file a name that differs from the name of the corresponding .RC file. For example, to name the resulting .RES file NEWFILE.RES, you would use the following command:

```
rc -fo newfile.res myapp.rc
```

The /fo option can also place the .RES file in a different directory. For example, the following command places the compiled resource file MYAPP.RES in the directory C:\SOURCE\RESOURCE:

```
rc -fo c:\source\resource\myapp.res myapp.rc
```

Searching for Resource Files

By default, RC searches for header files and resource files (such as icon and cursor files) first in the current directory and then in the directories specified by the INCLUDE environment variable. (The PATH environment variable has no effect on which directories RC searches.)

Adding a Directory to Search

You can use the /i option to add a directory to the list of directories RC searches. The compiler then searches the directories in the following order:

1. The current directory.
2. The directory or directories you specify by using the /i option, in the order in which they appear on the RC command line.
3. The list of directories specified by the INCLUDE environment variable, in the order in which the variable lists them, unless you specify the /x option.

The following example compiles the resource-definition file MYAPP.RC:

```
rc /i c:\source\stuff /i d:\resources myapp.rc
```

When compiling the script MYAPP.RC, RC searches for header files and resource files first in the current directory, then in C:\SOURCE\STUFF and D:\RESOURCES, and then in the directories specified by the INCLUDE environment variable.

Suppressing the INCLUDE Environment Variable

You can prevent RC from using the INCLUDE environment variable when determining the directories to search. To do so, use the /x option. The compiler then searches for files only in the current directory and in any directories you specify by using the /i option.

The following example compiles the script file MYAPP.RC:

```
rc /x /i c:\source\stuff myapp.rc
```

When compiling the script MYAPP.RC, RC searches for header files and resource files first in the current directory and then in C:\SOURCE\STUFF. It does not search the directories specified by the INCLUDE environment variable.

Displaying Resource-Compiler Progress Messages

You can use the /v option to specify that RC is to display progress messages. The following example causes RC to display progress messages as it compiles the resource-definition script SAMPLE.RC and creates the compiled resource file SAMPLE.RES:

```
rc /v sample.rc
```

Common Resource Statement Arguments

The following topics list arguments in common among the resource statements. Occasionally, a certain statement will use an argument differently, or may ignore an argument. The statement-specific variation is described with the statement in the alphabetical reference.

- [Common Control Arguments](#)
- [Common Resource Attributes](#)
- [Resource Memory Attributes](#)

Common Resource Control Arguments

The syntax for a resource control definition is as follows:

control [*text*,] *id*, *x*, *y*, *width*, *height* [, *style* [, *extended-style*]

Horizontal dialog units are 1/4 of the dialog base width unit. Vertical units are 1/8 of the dialog base height unit. The current dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** Win32 function returns the dialog base units in pixels. The coordinates are relative to the origin of the dialog box.

The arguments are as follows:

control

Specifies the keyword that indicates the type of control being defined, such as **PUSHBUTTON** or **CHECKBOX**.

text

Specifies text that is displayed with the control. The text is positioned within the control's specified dimensions, or adjacent to the control.

The *text* argument must contain zero or more characters enclosed in double quotation marks. Strings are automatically null-terminated and converted to Unicode in the resulting resource file, except for strings specified in *raw-data* statements (*raw-data* can be specified in **RCDATA** and user-defined resources). To specify a Unicode string in *raw-data*, explicitly qualify the string as a wide-character string by using the **L** prefix.

By default, the characters listed between the double quotation marks are ANSI characters, and escape sequences are interpreted as byte escape sequences. If the string is preceded by the **L** prefix, the string is a wide-character string and escape sequences are interpreted as two-byte escape sequences that specify Unicode characters. If a double quotation mark is required in the text, you must include the double quotation mark twice or use the `\` escape sequence.

An ampersand (&) character in the text indicates that the following character is used as a mnemonic character for the control. When the control is displayed, the ampersand is not shown, but the mnemonic character is underlined. The user can choose the control by pressing the key corresponding to the underlined mnemonic character. To use the ampersand as a character in a string, insert two ampersands (&&).

id

Specifies the control identifier. This value must be a 16-bit unsigned integer in the range 0 to

65,535 or a simple arithmetic expression that evaluates to a value in that range.

x

Specifies the x-coordinate of the left side of the control relative to the left side of the dialog box. This value must be a 16-bit unsigned integer in the range 0 to 65,535. The coordinate is in dialog units and is relative to the origin of the dialog box, window, or control containing the specified control.

y

Specifies the y-coordinate of the top side of the control relative to the top of the dialog box. This value must be a 16-bit unsigned integer in the range 0 to 65,535. The coordinate is in dialog units relative to the origin of the dialog box, window, or control containing the specified control.

width

Specifies the width of the control. This value must be a 16-bit unsigned integer in the range 1 to 65,535. The width is in 1/4-character units.

height

Specifies the height of the control. This value must be a 16-bit unsigned integer in the range 1 to 65,535. The height is in 1/8-character units.

style

Specifies the control styles. Use the bitwise OR (|) operator to combine styles.

extended-style

Specifies extended (WS_EX_*) styles. You must specify a *style* to specify an *extended-style*.

Common Resource Options

All resource-definition statements include a *load-mem* option that specifies the loading and memory characteristics of the resource. The only option used by Win32 is the DISCARDABLE option. The LOADONCALL and PRELOAD options are allowed in the script for compatibility with existing scripts, but are ignored.

Resource Memory Properties

Memory properties specify whether the resource is fixed or movable, whether it is discardable, and whether it is pure. The memory argument can be one or more of the following:

Argument	Description
FIXED	Ignored. In 16-bit Windows, the resource remains at a fixed memory location.
MOVEABLE	Ignored. In 16-bit Windows, the resource can be moved if necessary in order to compact memory.
DISCARDABLE	Resource can be discarded if no longer needed.

PURE	Ignored. Accepted for compatibility with existing resource scripts.
IMPURE	Ignored. Accepted for compatibility with existing resource scripts.

The default is **DISCARDABLE** for cursor, icon, and font resources.

Managing Libraries with LIB

You may find it useful to create libraries of Common Object File Format (COFF) object files to organize shared components of multiple projects. In Microsoft Developer Studio, you create and manage object libraries with a variety of dialogs. From the command line, you can use the Microsoft 32-Bit Library Manager (LIB.EXE) to manage COFF object libraries, create export files and import libraries to reference exported symbol definitions when you build Dynamic Link Libraries (DLLs), and extract library members.

You use the standard libraries, import libraries, and exports files LIB creates with LINK when building a 32-bit program. (LINK is described in [Using the Compiler and Linker from the Command Line](#) and [Compiler and Linker Options](#).) The three LIB modes -- creating standard (COFF) libraries, creating import libraries and export files, and extracting library members -- are mutually exclusive. You can use LIB in only one mode at a time.

You can use LIB to perform the following library-management tasks:

- Add objects to a library

Specify the filename for the existing library and the filenames for the new objects.

- Combine libraries

Specify the library filenames. You can add objects and combine libraries in a single LIB command.

- Replace a library member with a new object

Specify the library containing the member object to be replaced and the filename for the new object (or the library that contains it). When an object that has the same name exists in more than one input file, LIB puts the last object specified in the LIB command into the output library. When you replace a library member, be sure to specify the new object or library after the library that contains the old object.

- Delete a member from a library

Use the /REMOVE option. LIB processes any specifications of /REMOVE after combining all input objects, regardless of command-line order.

Note: You cannot both delete a member and extract it to a file in the same step. You must first extract the member object using /EXTRACT, then run LIB again using /REMOVE.

This section describes the Microsoft 32-Bit Library Manager (LIB.EXE). The following topics are covered:

- [LIB Input/Output](#)

- [Running LIB](#)
- [LIB Options](#)
- [Extracting a Library Member](#)
- [Import Libraries and Exports Files](#)

LIB Input/Output

LIB expects types of input files and generates types of output files depending on the mode in which it is used. You can also get information about the resulting library with the [/LIST](#) option, and you can examine the contents of the library by using [DUMPBIN](#) with the [/LINKERMEMBER](#) option.

For more information, see:

- [LIB Input Files](#)
- [LIB Output Files](#)
- [Other LIB Output](#)
- [Viewing Contents of a Library](#)

LIB Input Files

The input files expected by LIB depend on the mode in which it is used, as follows:

Mode	Input
Default (building or modifying a library)	COFF object (.OBJ) files, COFF libraries (.LIB), 32-bit OMF object (.OBJ) files
Extracting a member with /EXTRACT	COFF library (.LIB)
Building an exports file and import library with /DEF	Module-definition (.DEF) file, COFF object (.OBJ) files, COFF libraries (.LIB), 32-bit OMF object (.OBJ) files

Note: Object Model Format (OMF) libraries created by the 16-bit version of LIB cannot be used as input to the 32-bit LIB.

LIB Output Files

The output files produced by LIB depend on the usage mode as follows:

Mode	Output
Default (building or modifying a library)	COFF library (.LIB)
Extracting a member with /EXTRACT	Object (.OBJ) file
Building an exports file and import library with /DEF	Import library (.LIB) and exports (.EXP) file

Other LIB Output

In the default mode, you can use the [/LIST](#) option to display information about the resulting library. You can redirect this output to a file.

LIB displays a copyright and version message and echoes command files unless the [/NOLOGO](#) option is used.

When you type `lib` with no other input, LIB displays a usage statement that summarizes its options.

Error and warning messages issued by LIB have the form `LNKnnnn`. The LINK, DUMPBIN, and EDITBIN tools also use this range of errors.

Viewing Contents of a Library

A library contains COFF objects. Objects in a library contain functions and data that can be referenced externally by other objects in a program. An object in a library is sometimes referred to as a library member.

You can get additional information about the contents of a library by running the DUMPBIN tool with the `/LINKERMEMBER` option. For more information, see [Examining Files with DUMPBIN](#).

Running LIB

This section presents information on running LIB in any mode. It describes the LIB command line, discusses the use of command files, and gives general rules for using options.

For more information, see:

- [LIB Command Line](#)
- [LIB Command Files](#)
- [Using LIB Options](#)

LIB Command Line

To run LIB, type the command LIB followed by the options and filenames for the task you are using LIB to perform. LIB also accepts command-line input in command files. LIB does not use an environment variable.

Note: If you are accustomed to the LINK32.EXE and LIB32.EXE tools provided with the Microsoft Win32 Software Development Kit for Windows NT, you may have been using either the command `LINK32 -LIB` or the command `LIB32` for managing libraries and creating import libraries. Be sure to change your makefiles and batch files to use the LIB command instead.

LIB Command Files

You can pass command-line arguments to LIB in a command file by using the following syntax:

```
LIB @commandfile
```

The *commandfile* is the name of a text file. No space or tab is allowed between the at sign (@) and the filename. There is no default extension; you must specify the full filename, including any extension. Wildcards cannot be used. You can specify an absolute or relative path with the filename.

In the command file, arguments can be separated by spaces or tabs as they can on the commandline, and they can also be separated by newline characters. Use a semicolon (;) to mark a comment. LIB ignores all text from the semicolon to the end of the line.

You can specify either all or part of the command line in a command file, and you can use more than one command file in a LIB command. LIB accepts the command-file input as if it were specified in that location on the command line. Command files cannot be nested. LIB echoes the contents of command files unless the /NOLOGO option is used.

Using LIB Options

An option consists of an option specifier, which is either a dash (-) or a forward slash (/), followed by the name of the option. Option names cannot be abbreviated. Some options take an argument, specified after a colon (:). No spaces or tabs are allowed within an option specification. Use one or more spaces or tabs to separate option specifications on the command line.

Option names and their keyword or filename arguments are not case sensitive, but identifiers used as arguments are case sensitive. LIB processes options in the order specified on the command line and in command files. If an option is repeated with different arguments, the last one to be processed takes precedence.

The following LIB options apply to all modes of LIB:

- /MACHINE
Specifies the architecture of the library.
- /NOLOGO
Suppresses display of the LIB copyright message and version number and prevents echoing of command files.
- /VERBOSE
Displays details about the progress of the session. The information is sent to standard output and can be redirected to a file.

Other options apply only to specific modes of LIB. These options are discussed in the sections describing each mode.

LIB Options

The default mode for LIB is to build or modify a library of COFF objects. LIB runs in this mode when you do not specify /EXTRACT (to copy an object to a file) or /DEF (to build an import library).

To build a library from objects and/or libraries, use the following syntax:

```
LIB [options...] files...
```

This command creates a library from one or more input *files*. The *files* can be COFF object files, 32-bit OMF object files, and existing COFF libraries. LIB creates one library that contains all objects in the specified files. If an input file is a 32-bit OMF object file, LIB converts it to COFF before building the library. LIB cannot accept a 32-bit OMF object that is in a library created by the 16-bit

version of LIB. You must first use the 16-bit LIB to extract the object, then you can use the extracted object file as input to the 32-bit LIB. The 16-bit version of LIB is not provided with Visual Fortran.

By default, LIB names the output file using the base name of the first object or library file and the extension .LIB. If a file already exists with the same name, the output file overwrites the existing file. To preserve an existing library, use the /OUT option to specify a name for the output file.

The following options apply to building and modifying a library:

- /LIST
- /OUT
- /REMOVE

LIB Option /LIST

Displays information about the output library to standard output. The output can be redirected to a file. You can use /LIST to determine the contents of an existing library without modifying it.

LIB Option /OUT

The option LIB /OUT: *filename* overrides the default output filename. By default, the output library has the base name of the first library or object on the command line and the extension .LIB.

LIB Option /REMOVE

The option LIB /REMOVE: *object* omits the specified *object* from the output library. LIB creates an output library by first combining all objects (whether in object files or libraries), then deleting any objects specified with /REMOVE.

LIB Option /SUBSYSTEM

Tells the operating system how to run a program created by linking to the output library. For more information, see the description of the LINK /SUBSYSTEM option in Compiler and Linker Options.

You can use LIB to perform the following library-management tasks:

- Add objects to a library

Specify the filename for the existing library and the filenames for the new objects.

- Combine libraries

Specify the library filenames. You can add objects and combine libraries in a single LIB command.

- Replace a library member with a new object

Specify the library containing the member object to be replaced and the filename for the new object (or the library that contains it). When an object that has the same name exists in more than one input file, LIB puts the last object specified in the LIB command into the output

library. When you replace a library member, be sure to specify the new object or library after the library that contains the old object.

- Delete a member from a library

Use the /REMOVE option. LIB processes any specifications of /REMOVE after combining all input objects, regardless of command-line order.

Note: You cannot both delete a member and extract it to a file in the same step. You must first extract the member object using /EXTRACT, then run LIB again using /REMOVE. This behavior differs from that of the 16-bit LIB (for OMF libraries) provided in some Microsoft products.

Extracting a Library Member

You can use LIB to create an object (.OBJ) file that contains a copy of a member of an existing library. To extract a copy of a member, use the following syntax:

```
LIB library /EXTRACT:member /OUT:objectfile
```

This command creates an .OBJ file called *objectfile* that contains a copy of a *member* of a *library*. The *member* name is case sensitive. You can extract only one member in a single command. The /OUT option is required; there is no default output name. If a file called *objectfile* already exists in the specified directory (or current directory, if no directory is specified with *objectfile*), the extracted *objectfile* overwrites the existing file.

Import Libraries and Exports Files

You can use LIB with the /DEF option to create an import library and an exports file. LINK uses the exports file to build a program that contains exports (usually a DLL), and it uses the import library to resolve references to those exports in other programs.

In most situations, you do not need to use LIB to create your import library. When you link a program (either an executable file or a DLL) that contains exports, LINK automatically creates an import library that describes the exports. Later, when you link a program that references those exports, you specify the import library.

However, when a DLL exports to a program that it also imports from, whether directly or indirectly, you must use LIB to create one of the import libraries. When LIB creates an import library, it also creates an exports file. You must use the exports file when linking one of the DLLs.

For more information, see:

- [Building an Import Library and Exports File](#)
- [Using an Import Library and Exports File](#)

Building an Import Library and Exports File

To build an import library and exports file, use the following syntax:

```
LIB /DEF[:deffile] [options] [objfiles] [libraries]
```

When `/DEF` is specified, LIB creates the output files from export specifications that are passed in the LIB command. There are three methods for specifying exports, listed in recommended order of use:

- `cDEC$ ATTRIBUTES DLLEXPORT` in one of the *objfiles* or *libraries*
- A specification of `/EXPORT:name` on the LIB command line
- A definition in an EXPORTS statement in a *deffile*

These are the same methods you use to specify exports when linking an exporting program. A program can use more than one method. You can specify parts of the LIB command (such as multiple *objfiles* or `/EXPORT` specifications) in a command file in the LIB command, just as you can in a LINK command.

The following options apply to building an import library and exports file:

- `/DEBUGTYPE`
- `/OUT`
- `/EXPORT`
- `/INCLUDE`

LIB Import-Export Option `/DEBUGTYPE`

The option `/DEBUGTYPE:{CV|COFF|BOTH}` sets the format of debugging information. Specify CV for new-style Microsoft Symbolic Debugging Information, required by Visual C++ and Visual Fortran. Specify COFF for Common Object File Format (COFF) debugging information. Specify BOTH for both COFF debugging information and old-style Microsoft debugging information.

LIB Import-Export Option `/OUT`

The option `/OUT:import` overrides the default output filename for the *import* library being created. When `/OUT` is not specified, the default name is the base name of the first object file or library in the LIB command and the extension `.LIB`. The exports file is given the same base name as the import library and the extension `.EXP`.

LIB Import-Export Option `/EXPORT`

The option `/EXPORT:entryname[=internalname] [,@ordinal[,NONAME]][,DATA]` exports a function from your program to allow other programs to call the function. You can also export data. Exports are usually defined in a DLL.

The *entryname* is the name of the function or data item as it is to be used by the calling program. You can optionally specify the *internalname* as the function known in the defining program; by default, *internalname* is the same as *entryname*. The *ordinal* specifies an index into the exports table in the range 1 - 65535; if you do not specify *ordinal*, LIB assigns one. The **NONAME** keyword exports the function only as an ordinal, without an *entryname*.

LIB Import-Export Option `/INCLUDE`

The option `/INCLUDE:symbol` adds the specified symbol to the symbol table. This is useful for forcing the use of a library object that otherwise would not be included.

Using an Import Library and Exports File

When a program (either an executable file or a DLL) exports to another program that it also imports from, or if more than two programs both export to and import from each other, the commands to link these programs must accommodate the circular exports.

In a situation without circular exports, when you link a program that uses exports from another program, you must specify the import library for the exporting program. The import library for the exporting program is created when you link that exporting program. This requires that you link the exporting program before the importing program. For example, if TWO.DLL imports from ONE.DLL, you must first link ONE.DLL and get the import library ONE.LIB. You then specify ONE.LIB when you link TWO.DLL. When the linker creates TWO.DLL, it also creates its import library, TWO.LIB. You use TWO.LIB when linking programs that import from TWO.DLL.

However, in a circular export situation, it is not possible to link all of the interdependent programs using import libraries from the other programs. In the example discussed earlier, if TWO.DLL also exports to ONE.DLL, the import library for TWO.DLL won't exist yet when ONE.DLL is linked. When circular exports exist, you must use LIB to create an import library and exports file for one of the programs.

To begin, choose one of the programs on which to run LIB. In the LIB command, list all objects and libraries for the program and specify the /DEF option. If the program uses a .DEF file or /EXPORT specifications, specify these as well.

After you create the import library (.LIB) and the export file (.EXP) for the program, you then use the import library when linking the other program or programs. LINK creates an import library for each exporting program it builds. For example, if you ran LIB on the objects and exports for ONE.DLL, you created ONE.LIB and ONE.EXP. You can now use ONE.LIB when linking TWO.DLL; this step also creates the import library TWO.LIB.

Finally, link the program you began with. In the LINK command, specify the objects and libraries for the program, the .EXP file that LIB created for the program, and the import library or libraries for the exports used by the program. In the continuing example, the LINK command for ONE.DLL contains ONE.EXP and TWO.LIB, as well as the objects and libraries that go into ONE.DLL. Do not specify the .DEF file and /EXPORT specifications in the LINK command; these are not needed because the exports definitions are contained in the .EXP file. When you link using an .EXP file, LINK does not create an import library because it assumes that one was created when the .EXP file was created.

Editing files with EDITBIN

You can specify execution characteristics of a program or library by selecting options in Microsoft Developer Studio. For example, you might need to specify the base address at which a program is loaded by the operating system. If you work from the commandline, you can use the Microsoft Binary File Editor (EDITBIN) to set these types of controls.

This section describes the Microsoft COFF Binary File Editor (EDITBIN.EXE). EDITBIN modifies 32-bit Common Object File Format (COFF) binary files. You can use EDITBIN to modify object files, executable files, and dynamic-link libraries (DLLs).

EDITBIN converts the format of an Object Module Format (OMF) input file to COFF before making other changes to the file. You can use EDITBIN to convert the format of a file to COFF by running EDITBIN with no options.

The following topics are covered in this section:

- [EDITBIN Command Line](#)
- [EDITBIN Options](#)

EDITBIN Command Line

To run EDITBIN, use the following syntax:

```
EDITBIN [options] files...
```

Specify one or more files for the objects or images to be changed, and one or more *options* for changing the files.

When you type the command EDITBIN without any other command-line input, EDITBIN displays a usage statement that summarizes its options.

EDITBIN Options

An option consists of an option specifier, which is either a dash (-) or a forward slash (/), followed by the name of the option. Option names cannot be abbreviated. Some options take arguments, specified after a colon (:). No spaces or tabs are allowed within an option specification. Use one or more spaces or tabs to separate option specifications on the command line. Option names and their keyword or filename arguments are not case sensitive.

This section discusses the following EDITBIN options:

- [/BIND](#)
- [/HEAP](#)
- [/NOLOGO](#)
- [/REBASE](#)
- [/RELEASE](#)
- [/STACK](#)

EDITBIN Option /BIND

The option `/BIND[:PATH=path]` sets the addresses of the entry points in the import address table for an executable file or DLL. Use this option to reduce load time of a program.

Specify the program's executable file and DLLs in the *files* argument on the EDITBIN command line. The optional *path* argument to the `/BIND` option specifies the location of the DLLs used by the specified files. Separate multiple directories with semicolons (;). If *path* is not specified, EDITBIN searches the directories specified in the PATH environment variable. If *path* is specified, EDITBIN ignores the PATH variable.

By default, the Windows program loader sets the addresses of entry points when it loads a program. The amount of time this takes varies depending on the number of DLLs and the number of entry

points referenced in the program.

If a program has been modified with the `/BIND` option, and if the base addresses for the executable file and its DLLs do not conflict with DLLs that are already loaded, the operating system does not need to set these addresses. In a situation where the files are incorrectly based, the operating system will relocate the program's DLLs and recalculate the entry-point addresses; this adds to the program's load time.

EDITBIN Option /HEAP

The option `/HEAP:reserve[,commit]` sets the size of the heap in bytes. The *reserve* argument specifies the total heap allocation in virtual memory. The default heap size is 1MB. The linker rounds up the specified value to the nearest 4 bytes.

The optional *commit* argument is subject to interpretation by the operating system. In Windows 95 and Windows NT, it specifies the amount of physical memory to allocate at a time. Committed virtual memory causes space to be reserved in the paging file. A higher *commit* value saves time when the application needs more heap space but increases the memory requirements and possibly startup time.

Specify the *reserve* and *commit* values in decimal or C-language notation.

EDITBIN Option /NOLOGO

This option suppresses display of the EDITBIN copyright message and version number.

EDITBIN Option /REBASE

The option `/REBASE[:modifiers]` sets the base addresses for the specified files. EDITBIN assigns new base addresses in a contiguous address space according to the size of each file rounded up to the nearest 64K.

Specify the program's executable files and DLLs in the *files* argument on the EDITBIN command line in the order in which they are to be based. You can optionally specify one or more *modifiers*, each separated by a comma (,):

Modifier	Action
BASE= <i>address</i>	Provides a beginning address for reassigning base addresses to the files. Specify <i>address</i> in decimal or C-language notation. If BASE is not specified, the default starting base address is 0x400000. If DOWN is used, BASE must be specified, and <i>address</i> sets the end of the range of base addresses.
BASEFILE	Creates a file named COFFBASE.TXT, which is a text file in the format expected by LINK's <code>/BASE</code> option.
DOWN	Tells EDITBIN to reassign base addresses downward from an ending address. The files are reassigned in the order specified, with the first file located in the highest possible address below the end of the address range. BASE must be used with DOWN to ensure sufficient address space for basing the files. To determine the address space needed by the specified files, run EDITBIN with the <code>/REBASE</code> option

on the files and add 64K to the displayed total size.

EDITBIN Option /RELEASE

This option sets the checksum in the header of an executable file. The operating system requires the checksum for certain files such as device drivers. It is recommended that you set the checksum for release versions of your programs to ensure compatibility with future operating systems.

EDITBIN Option /STACK

The option `/STACK:reserve[,commit]` sets the size of the stack in bytes and takes arguments in decimal or C-language notation. The `/STACK` option applies only to an executable file.

The *reserve* argument specifies the total stack allocation in virtual memory. EDITBIN rounds up the specified value to the nearest 4 bytes. The optional *commit* argument is subject to interpretation by the operating system. In Windows NT, *commit* specifies the amount of physical memory to allocate at a time. Committed virtual memory causes space to be reserved in the paging file. A higher *commit* value saves time when the application needs more stack space but increases the memory requirements and possibly startup time.

Examining Files with DUMPBIN

There are times when you must examine or change OBJ, EXE, and DLL files. In Microsoft Developer Studio, you can open any file as a Binary rather than as an ASCII text file and work with both hexadecimal and ASCII versions of the contents. From the command line, you can use the Microsoft Binary File Dumper (DUMPBIN) to edit these types of files.

This section describes the Microsoft COFF Binary File Dumper (DUMPBIN.EXE). DUMPBIN displays information about 32-bit Common Object File Format (COFF) binary files. You can use DUMPBIN to examine COFF object files, standard libraries of COFF objects, executable files, and dynamic-link libraries (DLLs).

The following topics are covered in this section:

- [DUMPBIN Command Line](#)
- [DUMPBIN Options](#)

DUMPBIN Command Line

The syntax for DUMPBIN is:

```
DUMPBIN [options] files...
```

Specify one or more binary files, along with any options required to control the information. DUMPBIN displays the information to standard output. You can either redirect it to a file or use the `/OUT` option to specify a filename for the output.

When you run DUMPBIN on a file without specifying an option, DUMPBIN displays the `/SUMMARY` output.

When you type the command `dumpbin` without any other command-line input, `DUMPBIN` displays a usage statement that summarizes its options.

DUMPBIN Options

An option consists of an option specifier, which is either a dash (-) or a forward slash (/), followed by the name of the option. Option names cannot be abbreviated. Some options take arguments, specified after a colon (:). No spaces or tabs are allowed within an option specification. Use one or more spaces or tabs to separate option specifications on the command line. Option names and their keyword or filename arguments are not case sensitive. Most options apply to all binary files; a few apply only to certain types of files.

This section discusses the following `DUMPBIN` options:

- /ALL
- /ARCHIVEMEMBERS
- /DISASM
- /EXPORTS
- /FPO
- /HEADERS
- /IMPORTS
- /LINENUMBERS
- /LINKERMEMBER
- /OUT:filename
- /RAWDATA
- /RELOCATIONS
- /SUMMARY
- /SYMBOLS

DUMPBIN Option /ALL

Displays all available information except code disassembly. Use the /DISASM option to display disassembly. You can use /RAWDATA:NONE with the /ALL option to omit the raw binary details of the file.

DUMPBIN Option /ARCHIVEMEMBERS

Displays minimal information about member objects in a library.

DUMPBIN Option /DISASM

Displays disassembly of code sections, using symbols if present in the file.

DUMPBIN Option /EXPORTS

Displays all definitions exported from an executable file or DLL.

DUMPBIN Option /FPO

Displays Frame Pointer Optimization (FPO) records.

DUMPBIN Option /HEADERS

Displays coff header information.

DUMPBIN Option /IMPORTS

Displays all definitions imported to an executable file or DLL.

DUMPBIN Option /LINENUMBERS

Displays COFF line numbers. Line numbers exist in an object file if it was compiled with Program Database (/Zi) or Line Numbers Only (/Zd). An executable file or DLL contains COFF line numbers if it was linked with Generate Debug Info (/DEBUG) and COFF Format (/DEBUGTYPE:COFF).

DUMPBIN Option /LINKERMEMBER

The option /LINKERMEMBER[:{1|2}] displays public symbols defined in a library.

Specify the 1 argument to display symbols in object order, along with their offsets. Specify the 2 argument to display offsets and index numbers of objects, then list the symbols in alphabetical order along with the object index for each. To get both outputs, specify /LINKERMEMBER without the number argument.

DUMPBIN Option /OUT

The option /OUT:*filename* specifies a *filename* for the output. By default, DUMPBIN displays the information to standard output.

DUMPBIN Option /RAWDATA

The option /RAWDATA[:{ *BYTES* | *SHORTS* | *LONGS* | *NONE* }[, *number*]] displays the raw contents of each section in the file. The arguments control the format of the display, as follows:

Argument	Result
<i>BYTES</i>	The default. Contents are displayed in hexadecimal bytes, and also as ASCII if they have a printed representation.
<i>SHORTS</i>	Contents are displayed in hexadecimal words.
<i>LONGS</i>	Contents are displayed in hexadecimal longwords.
<i>NONE</i>	Raw data is suppressed. This is useful to control the output of the /ALL option.
<i>number</i>	Displayed lines are set to a width that holds <i>number</i> values per line.

DUMPBIN Option /RELOCATIONS

Displays any relocations in the object or image.

DUMPBIN Option /SUMMARY

Displays minimal information about sections, including total size. This option is the default if no other option is specified.

DUMPBIN Option /SYMBOLS

Displays the COFF symbol table. Symbol tables exist in all object files. A COFF symbol table appears in an image file only if it is linked with the Generate Debug Info and COFF Format options under Debug Info on the Debug category for the linker (or the /DEBUG and /DEBUGTYPE:COFF options on the command line).

Editing Format Descriptors with the Format Editor

The Format Editor is an application for Windows® that shows you what data formatted to match your edit descriptors will look like, and lets you edit either the descriptor list or the data. You can interactively create and edit Fortran 90 **FORMAT** statements and embedded formatting directives.

You can run the Format Editor either from the Edit menu in Microsoft Developer Studio or from the command line. The Format Editor program is located in the `...\DevStudio\SharedIDE\Bin` directory, and is called `FRMTEDIT.EXE`. To use it from the command line, you specify the source code file name, line number and column position in the argument list, and the Format Editor operates on the formatting at the indicated location. For example:

```
FRMTEDIT test.f90 5 18
```

If the line specified by the second parameter is empty, a new format statement is created with the words *label FORMAT*.

To use the Format Editor on a multi-line format statement, the argument list must specify the first line of the format statement. In-line comments in a multi-line Format statement are lost when the Format Editor writes the updated format statement and generates new continuation marks. Similarly, the part of a formatted I/O statement that follows the formatting directives is lost when the Format Editor writes the updated directive string back to the file.

When you are finished editing the format statement, the Format Editor rewrites the source file with code for the format you have developed. If the file has the extension `.F90`, the revised code is written with Fortran 90 free-form syntax rules; otherwise, it is written with Fortran 90 fixed-form syntax rules.

The Format Editor is installed on the Edit menu by the Visual Fortran Setup utility. If you have removed it for any reason, and need to reinstall it, you can do so by choosing Customize from the Tools menu. The argument list, which passes the current file name, line number and column position to the Format Editor, is `$File $Line $Column`. For more information about adding programs to the Tools menu, select the Help button in the Customize dialog.

Starting the Format Editor from Microsoft Developer Studio

The Format Editor presents the format code and a sample of the resulting data layout in a window that works like a dialog box. You can edit either the source code or the data layout, and the Format Editor changes the other to match.

▶ To open the Format Editor:

1. Load a Fortran source file that contains a **FORMAT** statement or an I/O edit descriptor.
2. Place the cursor on the first line of the **FORMAT** statement or on the line containing the edit descriptor.
3. From the Edit menu, choose Format Editor. The Format Editor dialog box opens.

The Format Editor dialog box consists of the following text boxes and buttons:

- The edit descriptors in the upper-left text box
- The sample data display in the lower-left text box
- The New Field, Remove Field, Change Value, OK, Cancel, and Help buttons along the bottom.

When you open the Format editor in a line with an edit descriptor, the editor attempts to parse the first opening quote that precedes the cursor position. When you open the editor in a line containing a **FORMAT** statement, the editor attempts to parse the edit descriptors in the statement. If the editor is successful, it displays the edit descriptors in the upper-left box and a sample data display in the lower text box. If the editor cannot parse the descriptor string or **FORMAT** statement, you will get a parse error message.

▶ To insert new I/O edit descriptors into existing formatted I/O statements or **FORMAT** statements in Microsoft Developer Studio:

1. Place the cursor in the existing descriptor or **FORMAT** statement.
2. From the Edit menu, choose Format Editor.
3. Choose New Field from the Format Editor dialog box.
4. Choose the descriptor type (Character, Integer, and so on) and choose whether to insert the descriptor before or after the current descriptor.
5. A default value is used for the descriptor you choose (for example, I5). To change the descriptor value, select the value and type in the new value. (For example, select 5 from I5 and type 8 to get an I8 format.)

▶ To insert a new Format statement:

1. Place the cursor on a blank line.
2. From the Edit menu, choose Format Editor.

The Format Editor inserts the words *label FORMAT* into the file at the cursor. You then must use the text editor in Microsoft Developer Studio or another text editor to add the edit descriptors.

For a discussion of the features of the Format Editor in Microsoft Developer Studio, choose Format Editor from the Edit menu, and click on Help in the Format Editor dialog box.

Profiling Code from the Command Line

The profiler is an analysis tool you can use to examine the run-time behavior of your programs. By profiling, you can find out which sections of your code are working efficiently and which need to be tuned. The profiler can also show areas of code that are not being executed.

Because profiling is a tuning process, you should use the profiler to make your programs run better, not to find bugs. Once your program is fairly stable, you should start profiling to find out where to optimize your code. Use the profiler to determine whether an algorithm is effective, a function is being called frequently (if at all), or if a piece of code is being covered by software testing procedures.

In Microsoft Developer Studio, you can use the Profiler to generate reports that characterize how your program executes. If you work from the command line, you can create a batch command file to run PREP, PROFILE and PLIST, the programs that generate execution profile reports.

For information on using the Profiler from Microsoft Developer Studio and timing your application, see [Analyze Program Performance](#).

This section describes how to use the components of the profiler from the command line. The following topics are covered:

- [Profiler Batch Processing](#)
- [Profiler Batch Files](#)
- [Profiler Command-Line Options](#)
- [Exporting Data From the Profiler](#)

Profiler Batch Processing

Profiling requires three separate programs: PREP, PROFILE, and PLIST. Microsoft Developer Studio executes all three of these programs for you automatically. To execute them efficiently from the command line, and to customize the output format or specify function and line count profiling, you must write batch files to invoke PREP, PROFILE, and PLIST. You can redirect the output of the batch file to a designated file by using the redirection character (>).

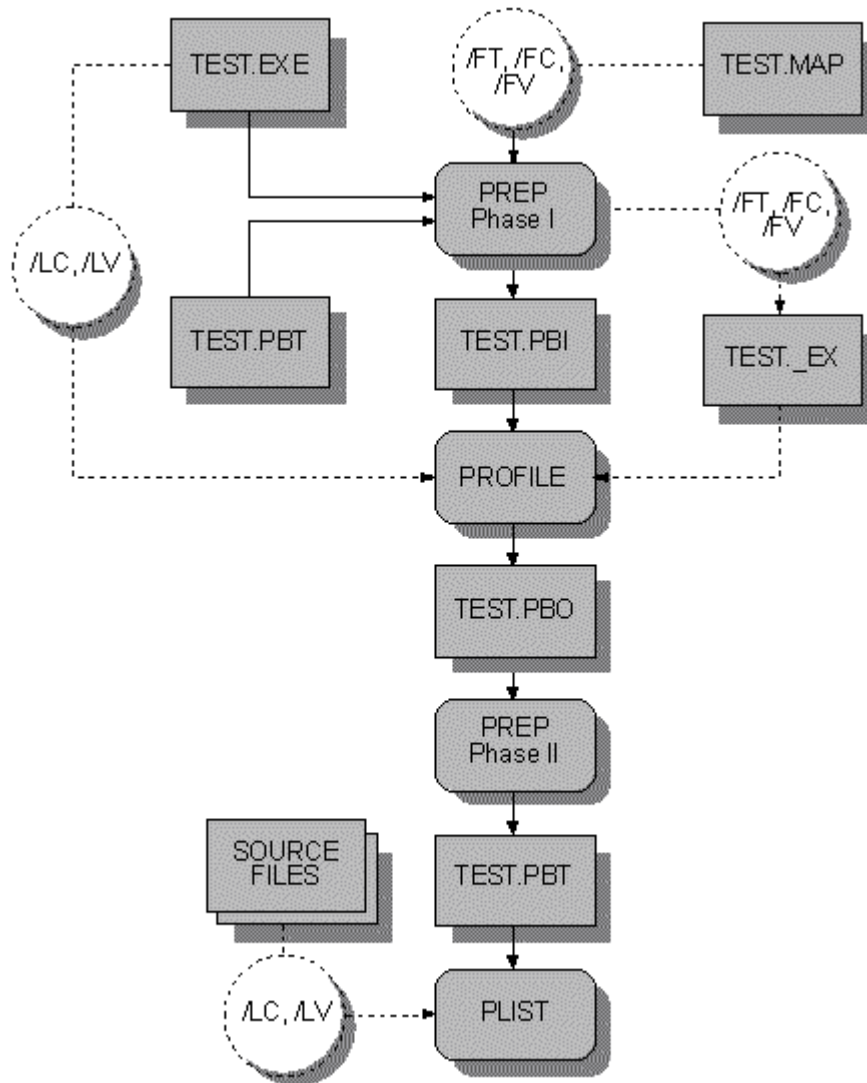
A typical profiler batch file might look like this:

```
PREP /OM /FT /EXC nafxcwd.lib %1
if errorlevel == 1 goto done
PROFILE %1 %2 %3 %4 %5 %6 %7 %8 %9
if errorlevel == 1 goto done
PREP /M %1
if errorlevel == 1 goto done
PLIST /SC %1 >%1.lst
:done
```

Note that the PREP program is called twice - once before the actual profiling and again afterward. The command-line arguments govern PREP's behavior. Intermediate files with extensions .PBI, .PBO, and .PBT are used to transfer information between profiling steps. The first call to PREP generates a .PBI file which is passed to PROFILER. PROFILER generates a .PBO file which is passed in the second call to PREP. The second call to PREP generates a .PBT file which is passed to

PLIST. The profiler data flow is shown in the following figure:

Profiler Data Flow



If the preceding batch file was named FTIME.BAT, and you wanted to profile the program TEST from the command prompt, you would type:

```
FTIME C:\Program Files\DF\MYDIR\TEST.EXE
```

Profiler Batch Files

Like the linker, all three profiler programs accept response files. The command line:

```
PREP /OM /FT /EXC nafxcwd.lib %1
```

can be replaced by the line:

```
PREP @opts.rsp %1
```

if you create a file OPTS.RSP that contains this text:

```
/OM /FT /EXC nafxcwd.lib # this is a comment
```

The # character in a response file defines a comment that runs through the end of the line.

Five standard batch files ship with the profiler:

Filename	Description
FTIME.BAT	Function timing
FCOUNT.BAT	Function counting
FCOVER.BAT	Function coverage
LCOUNT.BAT	Line counting
LCOVER.BAT	Line coverage

These batch files contain only the minimum parameters for the initial call to PREP. Use them as prototypes for your own batch files, which should contain selection parameters. If you ran an unmodified LCOVER batch file for a complex Fortran 90 application with a large number of functions, subroutines and modules, the output report could be thousands of lines long.

Profiler Command-Line Options

The next three sections describe the command-line options for the three components of the profiler:

- PREP
- PROFILE
- PLIST

PREP

The PREP program runs twice during a normal profiling operation. In Phase I, it reads an .EXE file and then creates .PBI and .PBT files. In Phase II, it reads .PBT and .PBO files and then writes a new .PBT file for PLIST. An 'X' in the following Options table indicates that a PREP command-line option applies to a particular phase.

The syntax for PREP is:

```
PREP [options] [programname1] [programname2...programname8]
```

PREP reads the command line from left to right, so the rightmost options override contradictory options to the left. None of the options are case sensitive. You must prefix options with a forward slash (/) or a dash (-), and options must be separated by spaces.

Parameter	Description
<i>options</i>	Control the kind of profiling, the inclusion and exclusion of code to be profiled, whether to merge profiles, and other profiling features. See the Options table .
<i>programname1</i>	Filename of primary program to profile (.DBG, .EXE, or .DLL). PROFILE adds the .EXE extension if no extension is given. This parameter must be specified in the first call to PREP and not the second call.

<i>programname2</i>	Additional programs to profile. These parameters can be specified for the first call to PREP only.
...	
<i>programname8</i>	

Options

Option	Phase		Description
	I	II	
/EXC	X		Excludes a specified module from the profile (See the <u>Remarks</u> section).
/EXCALL	X		Excludes all modules from the profile (See the <u>Remarks</u> section).
/FC	X		Selects function count profiling.
/FT	X		Selects function timing profiling. This option causes the profiler to generate count information as well.
/FV	X		Selects function coverage profiling.
/INC	X		Includes in profile (See the <u>Remarks</u> section).
/H[ELP]	X	X	Provides a short summary of PREP options.
/IO <i>filename</i>		X	Merges an existing .PBO file (the file generated by PROFILER to be passed in the second call to PREP). Up to eight .PBO files can be merged at a time. The default extension is .PBO.
/IT <i>filename</i>		X	Merges an existing .PBT file (the file generated by the second call to PREP to be passed to PLIST). Up to eight .PBT files can be merged at a time. You cannot merge .PBT files from different profiling methods. The default extension is .PBT.
/LC	X		Selects line count profiling.
/LV	X		Selects line coverage profiling.
/M <i>filename</i>		X	Substitutes for /IT, /IO, and /OT options.
/NOLOGO	X	X	Suppresses the PREP copyright message.
/OI <i>filename</i>	X		Creates a .PBI file (the file generated by the first call to PREP). The default extension is .PBI. If /OI is not specified, the output .PBI file is <i>programname1.PBI</i> .
/OM	X		Creates a self-profiling file with _XE or _LL extension for function timing, function counting, and function coverage. Without this option, the executable code is stored in the .PBI file. This option speeds up profiling.
/OT <i>filename</i>	X	X	Specifies the output .PBT file. The default extension is .PBT. If /OT is not specified, the output .PBT file is <i>programname1.PBT</i> .
/SF <i>function</i>	X		Starts profiling with <i>function</i> . The function name must correspond to an entry in the .MAP file.
/?	X	X	Provides a short summary of PREP options.

Environment Variable

The PREP environment variable specifies the default PREP command-line options. If a value for the PREP environment variable is not specified, the default options for PREP are:

```
/FT /OI filename /OT filename
```

where *filename* is set to the *programname1* parameter value.

Remarks

The `/INC` and `/EXC` options specify individual `.LIB`, `.OBJ`, `.FOR` and `.F90` files. For line counting and line coverage, you can specify line numbers with source files as in:

```
/EXCALL /INC TEST.F90(3-41,50-67)
```

In this example, the `/EXCALL` option excludes all modules from the profile, and the `/INC` option supercedes that to include only lines 3 - 41 and lines 50 - 67 from the source file `TEST.F90`. Note the absence of spaces in the source specification.

To specify all source lines in a particular module, specify the `.OBJ` file like this:

```
/EXCALL /INC TEST.OBJ
```

or by using the source filename with zero line numbers like this:

```
/EXCALL /INC TEST.F90(0-0)
```

The following statement profiles from line 50 to the end of the file:

```
/EXCALL /INC TEST.F90(50-0)
```

PROFILE

`PROFILE` profiles an application and generates a `.PBO` file of the results. Use `PROFILE` after creating a `.PBI` file with `PREP`.

The syntax for `PROFILE` is:

```
PROFILE [options] programname [programargs]
```

`PROFILE` reads the command line from left to right, so the rightmost options override contradictory options to the left. None of the options are case sensitive. You must prefix options with a forward slash (`/`) or a dash (`-`), and options must be separated by spaces.

If you do not specify a `.PBO` filename on the command line, `PROFILE` uses the base name of the `.PBI` file with a `.PBO` extension. If you do not specify a `.PBI` or a `.PBO` file, `PROFILE` uses the base name of *programname* with the `.PBI` and `.PBO` extensions.

Parameter	Description
<i>options</i>	Control <code>.PBI</code> input, <code>.PBO</code> output, error printing, and other profiler features. (See the Options table).
<i>programname</i>	Filename of program to profile. <code>PROFILE</code> adds the <code>.EXE</code> extension if no extension is given. (See the Remarks section).
<i>programargs</i>	Optional command-line arguments for <i>programname</i> . (See the Remarks section).

Options

Option	Description
/A	Appends any redirected error messages to an existing file. If the /E command-line option is used without the /A option, the file is overwritten. This option is valid only with the /E option.
/E <i>filename</i>	Sends profiler-generated error messages to <i>filename</i> .
/H[ELP]	Provides a short summary of PROFILE options.
/I <i>filename</i>	Specifies a .PBI file to be read. This file is generated by PREP.
/NOLOGO	Suppresses the PROFILE copyright message.
/O <i>filename</i>	Specifies a .PBO file to be generated. Use the PREP utility to merge with other .PBO files or to create a .PBT file for use with PLIST.
/X	Returns the exit code of the program being profiled.
/?	Provides a short summary of PROFILE options.

Remarks

You must specify the filename of the program to profile on the PROFILE command line. PROFILE assumes the .EXE extension, if no extension is given.

You can follow the program name with command-line arguments; these arguments are passed to the profiled program unchanged.

If you are profiling code in a .DLL file, give the name of an executable file that calls it. For example, if you want to profile SAMPLE.DLL, which is called by CALLER.EXE, you can type:

```
PROFILE CALLER.EXE
```

assuming that CALLER.PBI has SAMPLE.DLL selected for profiling.

Environment Variable

The PROFILE environment variable specifies the default command-line options for PROFILE. If the PROFILE environment variable is not specified, there are no defaults.

PLIST

PLIST converts results from the .PBT file generated by the second call to PREP into a formatted text file.

The syntax for PLIST is:

```
PLIST [options] inputfile
```

PLIST reads the command line from left to right, so the rightmost options override contradictory options to the left. None of the options are case sensitive. You must prefix options with a forward slash (/) or a dash (-), and options must be separated by spaces.

PLIST results are sent to STDOUT by default. Use the greater-than (>) redirection

PLIST must be run from the directory in which the profiled program was compiled.

Parameter	Description
<i>options</i>	Control the format and organization of profiler output data. (See the Options table .)
<i>inputfile</i>	The .PBT file to be converted by PLIST.

Options

Option	Description
<i>/C count</i>	Specifies the minimum hit count to appear in the listing.
<i>/D directory</i>	Specifies an additional directory for PLIST to search for source files. Use multiple <i>/D</i> command-line options to specify multiple directories. Use this option when PLIST cannot find a source file.
<i>/F</i>	Lists full paths in tab-delimited report.
<i>/H[ELP]</i>	Provides a short summary of PLIST options.
<i>/NOLOGO</i>	Suppresses the PLIST copyright message.
<i>/PL length</i>	Sets page length (in lines) of output. The length must be 0 or 15-255. A length of 0 suppresses page breaks. The default length is 0.
<i>/PW width</i>	Sets page width (in characters) of output. The width must be 1-511. The default width is 511.
<i>/SC</i>	Sorts output by counts, highest first.
<i>/SL</i>	Sorts output in the order that the lines appear in the file. This is the default. This option is available only when profiling by line.
<i>/SLS</i>	Forces line count profile output to be printed in coverage format.
<i>/SN</i>	Sorts output in alphabetical order by function name. This option is available only when profiling by function.
<i>/SNS</i>	Displays function timing or function counting information in function coverage format. Sorts output in alphabetical order by function name.
<i>/ST</i>	Sorts output by time, highest first.
<i>/T</i>	Tab-separated output. Generates a tab-delimited database from the .PBT file for export to other applications. All other options, including sort specifications, are ignored when using this option. For more information, see Exporting Data from the Profiler .
<i>/?</i>	Provides a summary of PLIST options.

Environment Variable

The PLIST environment variable specifies the default command-line options for PLIST. If the PLIST environment variable is not specified, the default options for PLIST depend on the profile type as shown:

Profile type	Sort option	Hit count option
Function timing	<i>/ST</i>	<i>/C 1</i>
Function counting	<i>/SC</i>	<i>/C 1</i>
Function coverage	<i>/SN</i>	<i>/C 0</i>
Line counting	<i>/SL</i>	<i>/C 0</i>
Line coverage	<i>/SL</i>	<i>/C 0</i>

Exporting Data from the Profiler

In addition to formatted reports, the PLIST report-generation utility can produce a tab-delimited report of profiler output. The following sections describe the data format of the report, steps for analyzing statistics in the report, and a Microsoft Excel macro that uses this report format:

- [Tab-Delimited Record Format](#)
- [Global Information Records](#)
- [Local Information Records](#)
- [Steps to Analyze Profiler Statistics](#)
- [Processing Profiler Output with Microsoft Excel](#)
- [Generating the Tab-Delimited Report](#)
- [Using the PROFILER.XLM Macro](#)
- [Changing the PROFILER.XLM Selection Criteria](#)

The PLIST /T command-line option causes PLIST to dump the contents of a .PBT file into a tab-delimited format suitable for import into a spreadsheet or database. This format can also be used by user-written programs.

For example, to create a tab-delimited file called MYPROG.TXT from MYPROG.PBT, enter:

```
PLIST /T MYPROG > MYPROG.TXT
```

Note: The ASCII tab-delimited format was designed to be read by other programs; it was not intended for general reporting.

Tab-Delimited Record Format

Every piece of data stored by the Profiler is available through the tab-delimited report. Because not all aspects of the database are recorded by every profiling method, unused fields within a record may be zero. For example, the total time of the program will be zero if the program was profiled for counts only. Also, all included functions will be listed for function counting and timing profiles, even if those functions were not executed.

The tab-delimited format is arranged with one record per line and two to eight fields per record. The following figure shows how a database looks when loaded into Microsoft Excel. The database was produced using the PLIST /T command-line option.

	A	B	C	D	E	F	G
1	0	51	Microsoft 32-bit PLIST Version 1.00				
2	1	522	Profile: Function timing, sorted by name				
3	2	4590.004	78.175	11			
4	3	266797	3251	38			
5	4	1993 May	C:\BSORT\BSORT				
6	5	?Sort@CSortStringList@@QAEXXZ					
7	6	bsort.exe	objcore.obj	0	0	0	AFX_CLAS
8	6	bsort.exe	except.obj	0	0	0	AFX_EXCE
9	6	bsort.exe	except.obj	0	0	0	AFX_EXCE
10	6	bsort.exe	except.obj	0	0	0	AFX_EXCE

Format tags: A, B, C, D, E, F
Data fields: G

The first item in each record is a format tag number. These tags range from 0 to 7 and indicate the kind of data given in the other fields of the record. The fields in each record are described in:

- Global Information Records
- Local Information Records

Tab-delimited reports are generated with global information records first, organized in numerical order by format tag. The local information records, containing information about specific lines or functions, are generated last. Local information records are organized by line number.

If the .PBT file contains information from more than one .EXE or .DLL file, the global information will cover them all. Local information records include the EXE field, which specifies the name of the executable file that each record pertains to.

Global Information Records

The global information records contain information about the entire executable file. The format tag numbers for global information records are 0 through 5. The record formats are as follows:

Profiler Banner		
0	Version	Banner

Field Explanation

0 Format tag number
Version PLIST version number
Banner PLIST banner

Profiling Method		
1	Method	Description

Field Explanation

1 Format tag number
Method Numeric value that indicates the profiling type (see the following table)
Description ASCII description of the profiling type given by the method field

The profiling types are listed in the following table:

Profiling Types	
Method	Description
321	Profile: Line counting, sorted by line
324	Profile: Line coverage, sorted by line
521	Profile: Function counting, sorted by function name
522	Profile: Function timing, sorted by function name
524	Profile: Function coverage, sorted by function name

Profiling Time and Depth			
2	Total Time	Outside Time	Call Depth

Field	Explanation
2	Format tag number
Total Time	Total amount of time used by the program being profiled. This field is zero for counting and coverage profiles
Outside Time	Amount of time spent before the first profiled function (with function profiling) or line (with line profiling) was executed. This field is zero for counting and coverage profiles
Call Depth	Maximum number of nested functions found while profiling. Only profiled functions are counted. This field is zero for line-level profiling

Hit Counts			
3	Total Hits	Lines/Funcs	Lines/Funcs Hit

Field	Explanation
3	Format tag number
Total Hits	Total number of times the profiler detected a profiled line or function being executed
Lines/Funcs	Total number of lines or functions marked for profiling
Lines/Funcs Hit	Number of marked lines or functions executed at least once while profiling

Date/Command Line		
4	Date	Command Line

Field	Explanation
4	Format tag number
Date	The date/time the profile was run (ASCII format)
Command Line	The PLIST command-line arguments

Starting Function Name	
5	Starting Function Name

Field	Explanation
5	Format tag number
Starting Function Name	The decorated name of the starting function identified by the PREP /SF parameter

Local Information Records

The local information records contain information about specific lines or functions that were profiled. The format tag numbers for local information records are 6 and 7. A report can have only one kind of local information record. The record formats are as follows:

Function Information						
6	Exe	Source	Count	Time	Child	Func

Field Explanation

6 Format tag number
 Exe ASCII name of the executable file that contains this function.
 Source ASCII name of the object module (including the .OBJ extension) that contains this function
 Count Number of times this function has been executed
 Time Amount of time spent executing this function in milliseconds. This field is zero with profiling by counting or coverage
 Child Amount of time spent executing the function and any child functions it calls. This field is zero with profiling by counting or coverage
 Func ASCII name of the function

Line Information				
7	Exe	Source	Line	Count

Field Explanation

7 Format tag number
 Exe ASCII name of the executable file that contains this function
 Source ASCII name of the source that contains the first line of this function
 Line Line number of this line
 Count Number of times this line has been executed. With coverage, this field is 1 if the line has been executed and 0 otherwise

Steps to Analyze Profiler Statistics

The profiler tab-delimited report format can contain a great deal of information. You can process this data in a spreadsheet, database, or user-written program.

► **To process the data in the tab-delimited report:**

1. Collect the cumulative data from the global information records. These lines begin with the numbers 0 through 5. Each of these lines appears only once, and always in ascending order.
2. Determine the type of database by finding the value of the "Method" field. This field is the second field of record type 1.
3. If the value in the "Method" field is greater than 400, the report comes from function profiling. If it is less than 400, the report comes from line profiling. The type of information in the local information records given later is directly related to this value.
4. In any one report, the local information records are always of the same type, either line information or function information.
5. Process data from the local information records. For example, to calculate the percentage of hits on a given function, divide the value of the "Count" field in record type 6 by the total number of hits from the "Total Hits" field of record type 3.
6. Remember that there can be only one type of local information record (either line or function information) in a report.

7. Send the results to a file or STDOUT.

Processing Profiler Output with Microsoft Excel

PROFILER.XLM is an example Microsoft Exceltm macro that processes a tab-delimited profiler report (generated by PLIST) and creates a graph based on the results. You will find this macro in the `..\vc\bin` directory.

The PROFILER.XLM macro is composed of four sub-macros. The first two macros, in columns A and B, are helper macros that copy and preprocess the data for use by the second pair of macros in columns C and D. The macro in column C, labeled `CreateColumnChart`, creates a graph showing the number of times that each function or line was executed. The final macro, in column D, is `CreateColumnTimeChart`; it works like `CreateColumnChart`, but operates on timing information.

Generating the Tab-Delimited Report

To generate the tab-delimited report, use the `PLIST /T` option. This can be done after the normal profile run has been completed; PLIST will read the profile data from the last profiler execution. The output of `PLIST /T` should be redirected to a file, preferably with the `.XLS` extension for easy loading into Excel (Excel will interpret the tab-delimited file correctly as a text file even with the `.XLS` extension).

Using the PROFILER.XLM Macro

To run the macro, follow these steps from within Microsoft Excel:

1. Open PROFILER.XLM by choosing Open from the File menu.
2. Open the tab-delimited report that was created by PLIST by choosing Open from the File menu.
3. If you have several open worksheets, activate the one containing the profiler data by selecting it with the mouse or by choosing its title from the Windows menu.
4. Run the macro:
 - Press CTRL+C for a chart based on hit counts.
 - Press CTRL+T for a chart based on timing.

You cannot get a timing chart if the report contains only counting or coverage information.

The macro typically takes only a few seconds to execute. When it is complete, Microsoft Excel displays a 3-D bar chart based on the results in the report. You can change the chart type by using the Gallery menu.

This macro copies the data in the report to another worksheet before processing it. The original tab-delimited report is left untouched.

Changing the PROFILER.XLM Selection Criteria

The standard PROFILER.XLM macro displays hit counts greater than 0 (for CTRL+C) and times greater than .01 millisecond (for CTRL+T). If you need to narrow the selections without analyzing the

macro, edit the formulas in cells C10 and D10.

Fortran Tools: FSPLIT and FPR

This section describes the following Fortran command-line tools:

- [FSPLIT](#) (includes F90SPLIT)
- [FPR](#)

FSPLIT and F90SPLIT

The FSPLIT and F90SPLIT tools split a multi-routine Fortran file into individual files. These tools are useful if you have a large Fortran program.

Use F90SPLIT when your program uses free-form source or Fortran 90/95 constructs. Use FSPLIT for FORTRAN 77 code. The FSPLIT and F90SPLIT commands have the same form:

```
FSPLIT [ -e:name] [-extend_source] [-silent] [input-file...]
```

```
F90SPLIT [ -e:name] [-extend_source] [-silent] [input-file...]
```

The command options are:

<code>-e:name</code>	Processes only the program unit name. You can specify more than one <code>-e name</code> on a command line.
<code>-extend_source</code>	Treats the statement field of each source line as ending in column 132, instead of column 72.
<code>-help, ?</code>	Displays information about the FSPLIT command.
<code>-nologo</code>	Suppresses the copyright notice that is displayed when FSPLIT or F90SPLIT is run.
<code>-silent</code>	Suppresses display of the name of each file opened (input and output files).
<code>input-file</code>	A Fortran source file to be split. You can specify more than one file by using a list of files. If <code>input-file</code> is omitted the FSPLIT or F90SPLIT Utility reads from standard input.

FSPLIT or F90SPLIT splits multi-routine Fortran files into separate routine files of the form `filename.for`, where `filename` is the name of the program unit (for example: a function, subroutine, block data, or program). The name for unnamed block data subprograms has the form `blkdtannn.for`, where `nnn` is a 3-digit code. For unnamed main programs, the name has the form `mainnnn.for`.

If there is an error in classifying a program unit, or if `filename.for` already exists, the program unit is put in a file named `zzznnn.for`, where `nnn` is a 3-digit code.

Normally each subprogram unit is split into a separate file.

Avoid using the `-e` option for unnamed main programs and block data subprograms since you must predict the created file name.

If FSPLIT or F90SPLIT cannot find the names specified by the `-e` option, an error message is written

to standard error device.

The following command example splits the subprogram units `readit` and `doit` into separate files:

```
FSPLIT -e readit -e doit prog.for
```

FPR

The FPR tool transforms files formatted according to Fortran's carriage control conventions into files formatted according to line printer conventions. The FPR command has the following form:

```
FPR [-f record-size] [filename]
```

The FPR command options are:

<code>-f record-size</code>	Specifies a fixed-length record as input. The <i>record-size</i> must be a decimal integer.
<i>filename</i>	Specifies the data file to be transformed.

FPR copies the input *filename* onto itself, replacing the carriage control characters with characters that will produce the intended effects when printed using the PRINT command. The first character of each line determines the vertical spacing as follows:

Character	Vertical Space Before Printing
Blank	One line
0	Two lines
1	To first line of next page
+	No advance
\$ or ASCII NUL	One line; no return after printing

FPR interprets the first character of every line of input, even if that character is not a recognizable control character. Control characters that are not recognized are treated as blanks and result in a single line advance.

FPR handles stream and fixed-length files. Input to FPR is assumed to be a stream (Stream_LF) file, unless you specify the `-f` option.

No diagnostic message is issued when FPR encounters an unrecognized control character.

PView and WinDiff

The PView process viewer (PVIEW.EXE) and the WinDiff utility (WINDIFF.EXE) are tools included with Microsoft Developer Studio. PView lets you examine and modify processes and threads running on your system. WinDiff graphically compares the contents of two files or two directories.

If these components have been installed, you can launch PView and WinDiff by double-clicking their icons in the Visual Fortran program folder. This section describes how to use these tools.

For more information, see:

- [Using WinDiff](#)
- [Using PView](#)

Using WinDiff

WinDiff graphically compares the contents of files and directories. If WinDiff has been installed, you can launch WinDiff by double-clicking its icon in the Visual Fortran program folder.

The full WinDiff syntax is:

```
WINDIFF path1 [path2] [-s [options] savefile]
```

The parameters for WinDiff are:

path1

Compares files in *path1* with files in current directory.

path1 path2

Compares files in *path1* with files in *path2*.

options

Can be any combination of the following options:

- *s*: Compares files that are the same in both paths.
- *l*: Compares only files in the first (left-hand) path.
- *r*: Compares only files in the second (right-hand) path.
- *d*: Compares two different files in both paths.

savefile

Name of text file to which comparison results are written.

These sections describe WinDiff features:

- [Expand/Outline Button](#)
- [WinDiff Colors](#)
- [WinDiff Menus](#)

Expand/Outline Button

When chosen, the Expand button displays the contents of the selected file. The button label then changes to Outline. When the Outline button is chosen, it displays only the filename(s). Files with the same name but different contents are displayed in red text. Identical files are displayed in black text.

WinDiff Colors

File contents are displayed in two background colors:

- Red--Indicates text from first (left-hand) file.
- Yellow--Indicates text from second (right-hand) file.

WinDiff Menus

WinDiff has File, Edit, View, Expand, Options, and Help menus. The items in these menus and their purposes are listed in the following tables.

File Menu

Menu item	Function
Compare Files	Displays the File Open dialog box in which you can enter the names of two files to compare.
Compare Directories	Displays the Select Directories dialog box in which you can enter the names of two directories to compare.
Close	Closes the current file(s).
Abort	Terminates a file-scanning operation. This menu selection is unavailable until a scanning operation is initiated.
Save File List	Displays a dialog box in which you can specify the name of the output file to which the comparison results are written.
Copy Files	Displays a dialog box in which you can specify files to be moved from one directory to another.
Print	Sends the comparison results to a printer.

Edit Menu

Menu item	Function
Edit Left File	Displays the contents of the first (left-hand) file using the default Notepad editor.
Edit Right File	Displays the contents of the second (right-hand) file using the default Notepad editor.
Edit Composite File	Displays both files using the default Notepad editor.
Set Editor	Displays a dialog box in which you can specify the editor to be used for the above operations. By default, Notepad is used.

View Menu

Menu item	Function
Outline	Shows only the list of filenames (equivalent to the Outline button).

Expand	Shows comparison of the contents of selected files (equivalent to the Expand button).
Picture	Shows a graphical representation of the contents of the two files.
Previous Change	Goes directly to previous area of the file that was changed (if any).
Next Change	Goes directly to next area of the file that was changed (if any).

Expand Menu

Menu item	Function
Left File Only	Expands only the left-hand file (with changed lines colored appropriately).
Right File Only	Expands only the right-hand file (with changed lines colored appropriately).
Both Files	Expands both files (with changed lines colored appropriately).
Left Line Numbers	Displays line numbers for the first (left-hand) file.
Right Line Numbers	Displays line numbers for the second (right-hand) file.
No Line Numbers	Turns off line number display.

Options Menu

Menu item	Function
Ignore Blanks	Blank spaces are ignored in the expanded view so that lines differing only in the amount of white space are shown as identical.
Show Identical Files	In outline view, displays files that are identical.
Show Left-Only Files	In outline view, displays files that appear only in the first (left-hand) path.
Show Right-Only Files	In outline view, displays files that appear only in the second (right-hand) path.
Show Different Files	In outline view, displays files that are in both paths, but are different.

Help Menu

Menu item	Purpose
About	Displays copyright and version information about WinDiff.

Using PView

With the PView process viewer (PVIEW.EXE) you can examine and modify many characteristics of processes and threads running on your system. This can help you optimize the operation of your system and your programs. PView can answer questions such as:

- How much memory does the program allocate at various points in its execution, and how much memory is being paged out?
- Which processes and threads are using the most CPU time?
- How does the program run at different system priorities?
- What happens if a thread or process stops responding to DDE (system dynamic data exchange), OLE, or pipe I/O?

- What percentage of time is spent in calls to Windows APIs (Application Programming Interface)?

Warning: PView lets you modify the status of processes running on your system. As a result, by using PView, you can stop processes and potentially halt the entire system. Make sure you save edited files before running PView.

The following topics are covered in this section:

- [Opening PView](#)
- [Selecting System Processes](#)
- [Process Memory Usage](#)
- [Selecting Base Process Priority](#)
- [Selecting Threads](#)
- [Thread Execution Information](#)
- [Thread Priority](#)
- [Process Memory Details](#)

Opening PView

To start PView, double-click its icon in the Visual Fortran program folder. PView opens by displaying the main Process Viewer dialog box. The Process Viewer dialog box consists of several boxes containing information on active processes and threads, and controls to change their behavior.

The following buttons control PView actions:

Button	Function
Exit	Closes PView.
Memory Details	Opens the Memory Details dialog box.
Kill Process	Removes the highlighted process from the system. This is different from choosing Close from the system menu, because the process is not informed of the shutdown (with WM_DESTROY) before it is stopped.
Refresh	Updates information in the Process Viewer dialog box and the Memory Details dialog box.
Connect	Views information about the computer specified in the Computer text box. The Computer text box should contain the network name of the computer you wish to view. Your ability to connect to a remote system may be affected by security on the target machine.

See the topics below for information on the Process Viewer boxes that relate to active processes and threads:

- [Process Selection](#)
- [Process Memory Used](#)
- [Process Priority](#)
- [Thread Selection](#)
- [Thread Information](#)
- [Thread Priority](#)

- Memory Details Dialog Box

Process Selection

The Process Selection list box in the Process Viewer dialog box displays information on the accessible processes running on the system. From this list, you can select a process to use for future actions. All other information and control areas in PView reflect the process chosen in this box.

Note: Because Windows NT is a secure operating system, you may not be able to view or alter attributes of some programs running on the system. See your *Windows NT User's Guide* for more information on security.

The fields in the Process Selection box are shown in the following table:

Field	Contents
Process	Name of the process on this line. Usually an .EXE filename.
CPU Time	Amount of CPU time this process has used.
Privileged	Percentage of the CPU time that was spent executing privileged code (code in the Windows NT Executive).
User	Percentage of the CPU time that was spent executing user code. This time includes time running protected subsystem code.

Process Memory Used

The Process Memory Used box in the Process Viewer dialog box displays information on the memory usage of the process selected in the Process Selection box.

Field	Contents
Working Set	The average amount of physical memory used by the process. The longer a process has been running, the more accurate this value is.
Heap Usage	The current total heap being used by the process. Heap space is taken by dynamically allocated data, including memory reserved by calls to the malloc , new , LocalAlloc , HeapAlloc , VirtualAlloc , and GlobalAlloc Window APIs.

Process Priority

The Priority buttons in the Process Viewer dialog box let you change the base priority of the process highlighted in the Process Selection box. This priority determines the activity of all threads of the selected process.

Button	Purpose
Very High	Maximum priority. CPU time is split between this and other Very High priority processes. Lower priority processes execute only when all Very High priority processes are blocked.
Normal	The standard priority group, also known as foreground. Most applications run with normal priority.
Idle	The lowest priority group, also known as background. Processes with this priority execute only when the system has no higher-priority processes that need CPU time. Screen savers run at this priority.

Thread Selection

The Thread Selection list box in the Process Viewer dialog box displays statistics for threads of the process selected in the Process Selection box and lets you select a thread for further operations.

Field	Contents
Threads	The thread ID number. This is the handle returned by CreateThread .
CPU Time	The amount of time that this instance of the thread has been running.
% Privileged	The percentage of the CPU time that was spent executing privileged code (code in the Windows NT Executive).
% User	The percentage of the CPU time that was spent executing user code. This time includes time running protected subsystem code.

Thread Information

The Thread Information box in the Process Viewer dialog box displays execution information about the thread selected in the Thread Selection box.

Field	Contents
User PC Value	The value of the instruction pointer for this thread.
Start Address	The address of the entry point of this thread. This information is useful for debugging.
Context Switches	Number of times that this thread has received CPU attention.
Dynamic Priority	The current dynamic thread priority. This number is determined by many factors, including user activity.

Thread Priority

The Thread Priority box in the Process Viewer dialog box shows you the base priority of the thread selected in the Thread Selection box. This is not an absolute priority, but is a range of priorities that can be selected by the operating system for the selected thread.

Field	Contents
Highest	The highest priority level allowed by the process priority.
Above Normal	Slightly elevated priority.
Normal	The standard priority level for the given process priority.
Below Normal	Reduced priority.
Idle	No CPU time will be spent on this thread unless all other threads are blocked.

Memory Details Dialog Box

The Memory Details dialog box gives information on the process selected by the Process Selection box in the Process Viewer dialog box.

To update the information in this dialog box, return to the Process Viewer dialog box and click the Refresh button.

This dialog box consists of the following buttons:

- OK

Returns to the Process Viewer dialog box.

- Process

The name and process ID of the process selected in the Process Selection box of the Process Viewer dialog box.

- User Address Space for

Displays the statistics for specific .EXE or .DLL files or Total Image Commit, which displays statistics for all components of the currently selected process. These statistics are shown in the following table.

Field	Contents
Inaccessible	Address space that cannot be accessed. This includes memory reserved by VirtualAlloc .
Read Only	Read-only data and code.
Writeable	Total data address space that can be written to.
Writeable (Not Written)	Data address space that can be written to, but has not been.
Executable	Code in selected EXEs and DLLs.

- Virtual Memory Counts

Displays the following statistics on Virtual Memory usage.

Field	Contents
Working Set	Average amount of virtual memory used by the process. The longer a process has been running, the more accurate this value is.
Peak Working Set	Maximum value attained by the Working Set described above.
Private Pages	Number of pages marked as private.
Virtual Size	Current size of virtual memory for this process.
Peak Virtual Size	Maximum size of virtual memory for this process.
Fault Count	Number of page faults. Each page fault represents an attempt to access memory at an address that was not in physical memory.
Paged	Number of pages currently in the swap file.
Peak Paged	Maximum number of pages currently in the swap file.
Non-Paged	Number of pages that have not been moved to the swap file.

Using the IMSL Mathematical and Statistical Libraries

The Professional Edition of Visual Fortran includes the IMSL[™] libraries, a collection of nearly 1000 mathematical and statistical functions easily accessible from Microsoft Developer Studio.

The IMSL libraries are installed with the Visual Fortran Professional Edition, as described in [Using Setup to Install Visual Fortran and Related Software](#) (in *Getting Started*).

When you install IMSL, you should also install the IMSL online documentation, which allow you to quickly find details on the purpose and use of any IMSL Library routine. You should view the IMSL `readme` file and online help provided in the Visual Fortran program folder (Professional Edition only). You can access the following topics through the IMSL Help file, available in the Visual Fortran program folder:

- IMSL MATH/LIBRARY Subroutines
- IMSL MATH/LIBRARY Special Functions
- IMSL STAT/LIBRARY Subroutines
- IMSL Fortran 90 MP Subroutines

Click on any of the libraries to see a submenu of items grouped by subject. Within each category (for example, Linear Systems, Eigensystem Analysis, and so on), click on a routine for more information.

IMSL libraries are included with the Professional Edition of Visual Fortran (*not* the Standard Edition).

This section provides information on the following topics:

- [Using the Libraries from Visual Fortran](#)
- [Library Naming Conventions](#)
- [Using IMSL Libraries in a Mixed-Language Environment](#)

Using the Libraries from Visual Fortran

To use the IMSL libraries, you need to:

1. Set the necessary IMSL environment variables for your development environment by executing the `DFVARS.BAT` file (installed by default in the directory `...\DF\BIN`). This sets the `INCLUDE` path and library (linker) search paths.

Within the F90 command-line window in the Visual Fortran program folder, the `DFVARS.BAT` file is already executed. Within Developer Studio, the equivalent of `DFVARS.BAT` file (as installed by Visual Fortran) is executed. You can view these directory paths within Developer Studio by:

- In the Tools menu, click Options.
- Click the Directory tab.
- In the drop-down list for Show Directories, select Library files and view the library paths.
- In the drop-down list for Show Directories, select Include files and view the include file

paths.

- Click OK if you have changed any information.

2. You may need to explicitly pass IMSL libraries to the Linker. In most cases, these are passed automatically by using `cDEC$ OBJCOMMENT LIB` source directives. To view the list of library names to be passed to the linker in Developer Studio:

- If not already open, open your Project Workspace (File menu, Open Workspace).
- In the Project menu, click on Settings.
- Click on the Link tab to view the list of Object/Library modules (General category). The IMSL libraries are listed in [Library Naming Conventions](#) and include the following library names:

```
sstatd.lib sstats.lib smathd.lib smaths.lib sf90mp.lib
```

- Click OK if you have changed any information.

3. Make IMSL routines and their interfaces available to your program:

- When calling MATH and STAT library routines from a Fortran 90 program, you should use the `numerical_libraries` module to provide interface blocks and parameter definitions for the routines. Including the following **USE** statement in your calling program will verify the correct usage of the IMSL routines at compile time:

```
USE numerical_libraries
```

When calling MATH and STAT library routines from a FORTRAN 77 style program, you can use the corresponding **INCLUDE** statement to perform the equivalent of the prior USE statement:

```
INCLUDE IMSLF90.FI
```

For more details, see the IMSL `readme` file in the Visual Fortran program folder.

When calling MATH and STAT library routines, you do not need to declare the functions or subroutines separately.

- When also calling Fortran 90 MP library routines, you should instead use the `imslf90` module to provide interface blocks and parameter definitions for all the Fortran 90 MP routines and the MATH and STAT library routines. Including the following **USE** statement in your calling program will verify the correct usage of the IMSL routines at compile time:

```
USE IMSLF90
```

For more information about calling the Fortran 90 MP routines, see the IMSL Libraries online help file.

The free-form Fortran 90 example program below invokes the function `AMACH` and the subroutine `UMACH` from the IMSL Libraries. The `AMACH` function retrieves real machine constants that

define the computer's real arithmetic. A value for positive machine infinity is returned (`Infinity`). The subprogram `UMACH` retrieves the output unit number.

```
! This free-form example demonstrates how to call
! IMSL routines from Visual Fortran.
!
! The module numerical_libraries includes the Math and
! Stat libraries; these contain the type declarations
! and interface statements for the library routines.

PROGRAM SHOWIMSL

USE NUMERICAL_LIBRARIES
INTEGER NOUT
REAL RINFP

! The AMACH function and UMACH subroutine are
! declared in the numerical_libraries module

CALL UMACH(2,NOUT)
RINFP = AMACH(7)
WRITE(NOUT,*) 'REAL POSITIVE MACHINE INFINITY = ',RINFP
END PROGRAM
```

For information on compiling and linking with Microsoft Developer Studio, see [Building Programs and Libraries in InfoViewer](#).

Note: IMSL routines are in general not multithread safe. In a multithread environment, you should take care that no two IMSL routines are active at the same time. To insure this, use multithread control techniques. For further information, see [Creating Multithread Applications](#).

Library Naming Conventions

The IMSL FORTRAN 77 MATH and STAT Numerical Libraries are provided in separate single- and double-precision versions. The IMSL libraries use the following library names:

File Name	Library Description
SMATHS	Single-precision MATH library, one of the IMSL FORTRAN 77 Numerical Libraries.
SMATHD	Double-precision MATH library, one of the IMSL FORTRAN 77 Numerical Libraries.
SSTATS	Single-precision STAT library, one of the IMSL FORTRAN 77 Numerical Libraries.
SSTATD	Double-precision STAT library, one of the IMSL FORTRAN 77 Numerical Libraries.
SF90MP	Fortran 90 MP library, a new generation of Fortran 90-based algorithms, optimized for multiprocessor and other high-performance systems.

The IMSL FORTRAN 77 Numerical Libraries are for applications in general applied mathematics and for analyzing and presenting statistical data in scientific and business applications.

For command-line window development, executing the `DFVARS.BAT` file (see [Using the Compiler and Linker from the Command Line](#)) sets Visual Fortran environment variables as well as IMSL (Professional Edition) environment variables (see [Environment Variables Used with the DF Command](#)).

For more information on the IMSL libraries, see:

- The IMSL `readme` file provided in the Visual Fortran program folder.
- The IMSL online help provided in the Visual Fortran program folder.
- Product information about IMSL at the following URL: <http://www.vni.com>.

Using IMSL Libraries in a Mixed-Language Environment

This section explains how to use the IMSL Libraries in a mixed-language development environment with Visual Fortran and Microsoft Visual C++®.

Messages that IMSL routines write to standard output or to error output in a mixed-language application or an application for Windows can be awkward if they are written to the screen. You can avoid this by calling `UMACH` from a Fortran routine to remap the output and error units to a file instead of to the screen. For example, the following free-form program writes the standard output from `VHSTP` to the file `STD.TXT`, and the error message from `AMACH` to the file `ERR.TXT`:

```

PROGRAM fileout
!   This program demonstrates how to use the UMACH routine to
!   redirect the standard output and error output from IMSL
!   routines to files instead of to the screen. The routines
!   AMACH and UMACH are declared in the numerical_libraries module
!
USE numerical_libraries
INTEGER STDU, ERRU
REAL x, frq(10)/3.0,1.0,4.0,1.0,5.0,9.0,2.0,6.0,5.0,3.0/
!
!   Redirect IMSL standard output to STD.TXT at unit 8
!
CALL umach(-2, STDU)
OPEN (unit=STDU, file='std.txt')
CALL vhistp(10,frq,1,'Histogram Plot')
CLOSE(8)
!
!   Redirect IMSL error output to ERR.TXT at unit 9
!
CALL umach(-3, ERRU)
OPEN (unit=ERRU, file='err.txt')
x = amach(0)    ! Illegal parameter error
CLOSE(9)
END

```

The standard output from IMSL routine `VHSTP` written to `STD.TXT` is:

```

1
          Histogram Plot
Frequency-----
  9 *           I           *
  8 *           I           *
  7 *           I           *
  6 *           I   I       *
  5 *           I I   I I   *
  4 *           I  I I   I I *
  3 *  I   I   I I   I I I   *
  2 *  I   I   I I I I I I   *
  1 *  I I I I I I I I I I   *
-----
Class           5           10

```

The error output from IMSL routine AMACH written to ERR.TXT is:

```
*** TERMINAL ERROR 5 from AMACH. The argument must be between 1 and 8
***          inclusive. N = 0
```

Consider the following simple Fortran example that uses the IMSL library:

```
USE numerical_libraries
real rinfp
rinfp = AMACH(7)
write(*,*) 'Real positive machine infinity = ',rinfp
end
```

The output is:

```
Real positive machine infinity = Infinity
```

The corresponding C example is:

```
/* FILE CSAMP0.C */
#include <stdio.h>
#include <stdlib.h>

extern float _stdcall AMACH(long *);

main()
{
    long n;
    float rinfp;

    n = 7;
    rinfp = AMACH(&n);
    printf("Real positive machine infinity = %16E\n", rinfp);

    fflush(stdout);
    _exit(0);
}
```

This C language example demonstrates the use of:

- The `_stdcall` modifier in the function prototype needed when calling the IMSL libraries.
- The `&` address operator passes the address of the variable to the subprogram (IMSL libraries expect arguments passed by reference).

The C example can be compiled by the `c1` command to create an object file that can be linked using the `DF` command.

For more information on mixed-language programming, see [Programming with Mixed Languages](#).

Appendix A: Compatibility Information

Visual Fortran uses the same DIGITAL Fortran compiler available on DIGITAL UNIX® and OpenVMS™ Alpha systems. DIGITAL Visual Fortran supports extensions to the ISO and ANSI standards, including a number of extensions defined by:

- Microsoft® Fortran PowerStation 4.0
- DIGITAL Fortran for the various DIGITAL Fortran platforms

Many language extensions associated with Microsoft Fortran Powerstation Version 4 have been added to Visual Fortran; most of these extensions will be added to different releases of DIGITAL Fortran on Alpha platforms.

The following sections describe Visual Fortran compatibility information:

- [Compatibility with Microsoft Fortran Powerstation](#)
- [Compatibility with DIGITAL Fortran on Other Platforms](#)

Compatibility with Microsoft Fortran Powerstation

Visual Fortran recognizes the FL32 command and many of the command-line options provided by the Microsoft Fortran Powerstation Version 4 compiler. For more information on command-line compatibility, see [Microsoft Fortran PowerStation Command-Line Compatibility](#).

Visual Fortran supports many of the language extensions to the Fortran 90 Standard supported by Microsoft Fortran Powerstation Version 4. Certain extensions may require the `/fpscomp` compiler option (also see [Categories of Compiler Options](#)). These extensions include the following:

- `.f`, `.for`, `.f90` source file types
- `# Constants` - constants using other than base 10
- `C strings` - NULL terminated strings
- `MBCS characters in comments`
- `MBCS characters in string literals`
- `Conditional compilation and metacommand (directive) expressions` (`$DEFINE`, `$UNDEFINE`, `$IF`, `$ELSEIF`, `$ELSE`, `$ENDIF`)
- `!MS$ directive form` (see [Compiler Directives: table](#))
- `$FREEFORM`, `$NOFREEFORM`, `$FIXEDFORM` - source file format
- `$OBJCOMMENT` - place library-search record in object file
- `$INTEGER`, `$REAL` - selects size
- `$FIXEDFORMLINESIZE` - line length for fixed form source
- `$STRICT`, `$NOSTRICT` - F90 conformance
- `$ATTRIBUTES`, identifier attributes (`C`, `STDCALL`, `REFERENCE`, `VALUE`, `DLLIMPORT`, `DLLEXPORT`, `EXTERN`, `ALIAS`, `VARYING`)
- `$PACK` - structure packing
- `Kind numbers match bytes` - kind parameters
- `AUTOMATIC` attribute - automatic storage class
- `Integer Pointers (Cray pointers)`
- `VAX Structures == F90 sequence derived types`

- Mixing logicals and numerics - logicals used with arithmetic operators and variables
- Argument matching for procedure calls
- Mixing integer kinds to intrinsics
- Byte data type == INTEGER*1
- **\$ATTRIBUTES []** Form
- **\$ATTRIBUTES ALIAS** - external name for a subprogram
- **\$ATTRIBUTES C, STDCALL** - calling and naming conventions
- **\$ATTRIBUTES VALUE, REFERENCE** - argument passing calling conventions
- **\$ATTRIBUTES DLLIMPORT, DLLEXPORT** - import from/export to DLL
- Character and non-character equivalence
- Double complex data type
- **.XOR.** - exclusive disjunction
- Integer arguments in logical expressions
- **OPEN** statement specifier options:
 - **BLOCKSIZE=** internal buffer size used in I/O
 - **CARRIAGECONTROL=** controls the output of formatted files
 - **MODE=** controls access to file on networked systems
 - **TITLE=, IOFOCUS=** controls QuickWin child windows
 - **SHARE=** controls simultaneous access to file on networked systems
- Default carriage control
- Implicit open - prompt user for filenames
- Special device names for **FILE=** in **OPEN** statements
- **FORM=BINARY** in **INQUIRE/OPEN** statements
- Unformatted sequential file form
- Q edit descriptor - number of characters remaining in the input record
- \ descriptor - prevents writing an end-of-record mark
- \$ edit descriptor - suppresses the carriage return at the end of a record
- X edit descriptor default - 1
- Ew.dDe and Gw.dDe edit descriptors - similar to Ew.dEe and Gw.dEe
- Variable Format Expressions (VFEs) - integer expression in **FORMAT** statement
- Expanded missing ','s in **FORMAT** statements - optional commas
- Expanded namelist start/end sequences
- All path names: including driver, compiler, and **INCLUDE** statement MBCS enabled [not W95]
- UNC pathnames
- Long filenames
- 7200 character statement length
- Free form infinite line length
- **\$DECLARE** and **\$NODECLARE == IMPLICIT NONE**
- Logical truth: 0 = false, non-zero = true
- **\$ATTRIBUTES EXTERN** - variable allocated in another source file
- **\$ATTRIBUTES VARYING** - variable number of arguments
- Alternate **PARAMETER** syntax - no parenthesis
- \$ in identifiers
- **INTERFACE TO** - subroutine/function prototype, however global scoping is not supported
- Argument passing modifiers - **%VAL, %REF**
- Argument passing modifiers - **%DESCR** (treated as **%REF**)
- CRAY pointer support for procedure names (for COM/OLE support)

- **\$ATTRIBUTES ALLOCATABLE** - allocatable array
- Mixing subroutines/functions in generic interfaces
- **\$MESSAGE** - output message during compilation
- **\$LINE == C's #line**
- Listing directives - **\$TITLE, \$SUBTITLE**
- **STATIC** attribute-static storage class
- **EOF** checks for end of file
- **LOC** equivalent to **%LOC**
- **HFIX** converts to short integer
- **INT1** converts to one byte integer by truncating
- **INT2** converts to two byte integer by truncating
- **INT4** converts to four byte integer by truncating
- **JFIX** same as **INT4**
- **MALLOC** allocates a memory block of size bytes and returns an integer pointer to the block
- **FREE** frees the memory block specified by the integer pointer
- **COTAN** returns cotangent
- **DCOTAN** returns double precision cotangent
- **IMAG** returns the imaginary part of complex number
- **IBCHNG** reverses value of bit
- **ISHA** shifts arithmetically left or right
- **ISHC** performs a circular shift
- **ISHL** shifts logically left or right

The following known source incompatibilities exist between Microsoft Fortran Powerstation Version 4 and Visual Fortran:

- **DATA** statement style initialization in attribute style declaration (not supported)
- Debug lines (other than D) (not supported)
- **\$OPTIMIZE**-change optimization options (not supported)
- Integer array can contain format (not supported)
- Listing directives - **\$PAGE, \$PAGESIZE, \$LINESIZE, \$[NO]LIST, \$INCLUDE** (not supported)
- **\$DEBUG, \$NODEBUG** - additional runtime checking (not supported)
- Internal files can be any type (not supported)
- Negative I/O unit numbers (not supported)
- Interface blocks using **INTERFACE [TO]** at the beginning of a source file to provide global scoping for subsequent program units. (not supported)
Visual Fortran uses standard Fortran 90 semantic rules about interface block placement and use.
- Tab continuation lines that start with characters other than digits 1 through 9 (not supported)

Compatibility with DIGITAL Fortran on Other Platforms

DIGITAL Visual Fortran supports extensions to the ISO and ANSI standards, including a number of extensions defined by Microsoft Fortran PowerStation 4.0 (see [Compatibility with Microsoft Fortran Powerstation](#)) and DIGITAL Fortran for the various DIGITAL Fortran platforms (operating system/architecture pairs).

In addition to DIGITAL Visual Fortran systems, DIGITAL Fortran platforms include:

- DIGITAL Fortran 90 and DIGITAL Fortran 77 on DIGITAL UNIX (formerly DECOSF/1®) Alpha systems
- DIGITAL Fortran 90 and DIGITAL Fortran 77 on OpenVMS Alpha systems
- DIGITAL Fortran 77 on OpenVMS VAX™ systems

Major additions to the FORTRAN 77 standard introduced by the Fortran 90 standard include:

- Array operations
- Improved facilities for numeric computation
- Parameterized intrinsic data types
- User-defined data types
- Facilities for modular data and procedure definitions
- Pointers (Fortran 90 pointers)
- The concept of language evolution

In addition, the Fortran 90 standard includes the following industry-accepted extensions to the FORTRAN 77 standard:

- Support for recursive subprograms
- **IMPLICIT NONE** statements
- **INCLUDE** statement
- **NAMELIST**-directed I/O
- **DO WHILE** and **ENDDO** statements
- Use of exclamation point (!) for end of line comments
- Support for automatic arrays
- Support for the following **SELECT CASE - CASE - CASE DEFAULT - END SELECT** statements.
- Support for the **EXIT** and **CYCLE** statements and for construct names on **DO - END DO** statements

DIGITAL Visual Fortran includes the following features and enhancements also found on other DIGITAL Fortran platforms:

- Support for linking against static libraries
- Support for linking against dynamically linked libraries (DLL)
- Support for creating code to be put into a dynamically linked library (DLL)
- Support for stack-based storage
- Support for dynamic memory allocation
- Support for reading and writing binary data files in nonnative formats, including IEEE® (little-endian and big-endian), VAX, IBM® System/360, and CRAY® integer and floating point formats
- User control over IEEE floating point exception handling, reporting, and resulting values.
- Control for memory boundary alignment of items in **COMMON** and fields in structures and warnings for misaligned data
- Directives to control listing page titles and subtitles, object file identification field, **COMMON** and record field alignment, and some attributes of **COMMON** blocks
- Composite data declarations using **STRUCTURE**, **END STRUCTURE**, and **RECORD** statements, and access to record components through field references
- Explicit specification of storage allocation units for data types such as:
 - **INTEGER*4**
 - **LOGICAL*4**

- REAL*4
- REAL*8
- COMPLEX*8
- Support for 64-bit signed integers using INTEGER*8 and LOGICAL*8 (on Alpha platforms only)
- A set of data types:
 - BYTE
 - LOGICAL*1, LOGICAL*2, LOGICAL*4
 - INTEGER*1, INTEGER*2, INTEGER*4
 - LOGICAL*8 and INTEGER*8 on Alpha platforms only
 - REAL*4, REAL*8
 - COMPLEX*8, COMPLEX*16, DOUBLE COMPLEX
 - DIGITAL Fortran **POINTER** statement (CRAY style)
- Data statement style initialization in type declaration statements
- **AUTOMATIC** and **STATIC** statements
- Bit constants to initialize LOGICAL, REAL, and INTEGER values and participate in arithmetic and logical expressions
- Built-in functions %LOC, %REF, and %VAL
- **VOLATILE** statement
- Bit manipulation functions
- Binary, hexadecimal, and octal constants and Z and O format edit descriptors applicable to all data types
- I/O unit numbers that can be any nonnegative INTEGER*4 value
- Variable amounts of data can be read from and written to "STREAM" files, which contain no record delimiters
- **ENCODE** and **DECODE** statements
- **ACCEPT**, **TYPE**, and **REWRITE** input/output statements
- **DEFINE FILE**, **UNLOCK**, and **DELETE** statements
- **USEROPEN** subroutine invocation at file **OPEN**
- Debug statements in source
- Generation of a source listing file with optional machine code representation of the executable source
- Variable format expressions in a **FORMAT** statement
- Optional run-time bounds checking of array subscripts and character substrings
- 31-character identifiers that can include dollar sign (\$) and underscore (_)
- Language elements that support the various extended range and extended precision floating point architectural features:
 - 32-bit IEEE S_floating data type, with an 8-bit exponent and 24-bit mantissa and a precision of typically 7 decimal digits
 - 64-bit IEEE T_floating data type, with an 11-bit exponent and 53-bit mantissa and a precision of typically 15 decimal digits
- Command line control for:
 - The size of default INTEGER, REAL, and DOUBLE PRECISION data items
 - The levels and types of optimization to be applied to the program
 - The directories to search for **INCLUDE** and module files
 - Inclusion or suppression of various compile-time warnings
 - Inclusion or suppression of run-time checking for various I/O and computational errors
 - Control over whether compilation terminates after a specific number of errors has been

found

- Choosing whether executing code will be thread-reentrant
- Kind types for all of the hardware-supported data types:
 - For 1-, 2-, and 4-byte LOGICAL data: LOGICAL (KIND=1), LOGICAL(KIND=2), LOGICAL (KIND=4)
 - For 1-, 2-, and 4-byte INTEGER data: INTEGER (KIND=1), INTEGER(KIND=2), INTEGER (KIND=4)
 - For 8-byte LOGICAL and INTEGER data on Alpha platforms only: LOGICAL (KIND=8), INTEGER (KIND=8)
 - For 4- and 8-byte REAL data: REAL (KIND=4), REAL (KIND=8)
 - For single precision and double precision COMPLEX data: COMPLEX (KIND=4), COMPLEX (KIND=8)

Appendix B: FORTRAN 77 Syntax

This section contains the syntax for statements and intrinsics of ANSI FORTRAN 77. All are recognized by Visual Fortran without the use of special compiler options, except in certain special instances.

This section discusses the following topics:

- [FORTRAN 77 Data Types](#)
- [FORTRAN 77 Intrinsic Functions](#)
- [FORTRAN 77 Statements](#)

FORTRAN 77 Data Types

The six data types defined by the ANSI FORTRAN 77 specification are:

- INTEGER
- REAL
- DOUBLE PRECISION
- COMPLEX
- LOGICAL
- CHARACTER [*n], where n is between 1 and 32,767

The data type of a variable, symbolic constant, or function can be declared in a specification statement. If its type is not declared, the compiler determines a data type by the first letter of the variable, constant, or function name. A type statement can also dimension an array variable.

Default requirements for these data types are listed in the following table.

Type	Bytes
INTEGER	4
REAL	4
DOUBLE PRECISION	8
COMPLEX	8
LOGICAL	4
CHARACTER	1
CHARACTER*n	n (See note below)

Note: The maximum n is 32,767.

FORTRAN 77 Intrinsic Functions

Function syntax	Type of return value
ABS (<i>gen</i>)	Same as argument
ACOS (<i>real</i>)	Same as argument
AIMAG (<i>cmp8</i>)	REAL
AIMT (<i>real</i>)	Same as argument

ALOG (<i>real4</i>)	REAL
ALOG10 (<i>real4</i>)	REAL
AMAX0 (<i>intA</i> , <i>intB</i> [, <i>intC</i>] ..)	REAL
AMAX1 (<i>real4A</i> , <i>real4B</i> , [, <i>real4C</i>]...)	REAL
AMIN0 (<i>intA</i> , <i>intB</i> [, <i>intC</i>]...)	REAL
AMIN1 (<i>real4A</i> , <i>real4B</i> [, <i>real4C</i>]...)	REAL
AMOD (<i>value</i> , <i>mod</i>)	REAL
ANINT (<i>value</i>)	REAL
ASIN (<i>real</i>)	Same as argument
ATAN (<i>real</i>)	Same as argument
ATAN2 (<i>realA</i> , <i>realB</i>)	Same as argument
CABS (<i>cmp</i>)	Same as argument; COMPLEX returns REAL
CCOS (<i>cmp8</i>)	COMPLEX
CHAR (<i>int</i>)	CHARACTER
CLOG (<i>cmp8</i>)	COMPLEX
CMPLX (<i>genA</i> [, <i>genB</i>])	COMPLEX
CONJG (<i>cx8value</i>)	COMPLEX
COS (<i>gen</i>)	Same as argument
COSH (<i>real</i>)	Same as argument
CSIN (<i>cmp8</i>)	COMPLEX
CSQRT (<i>cx8value</i>)	COMPLEX
DABS (<i>r8value</i>)	DOUBLE PRECISION
DACOS (<i>dbl</i>)	DOUBLE PRECISION
DASIN (<i>dbl</i>)	DOUBLE PRECISION
DATAN (<i>dbl</i>)	DOUBLE PRECISION
DATAN2 (<i>dblA</i> , <i>dblB</i>)	DOUBLE PRECISION
DBLE (<i>value</i>)	DOUBLE PRECISION
DCOS (<i>dbl</i>)	DOUBLE PRECISION
DCOSH (<i>dbl</i>)	DOUBLE PRECISION
DDIM (<i>dblA</i> , <i>dblB</i>)	DOUBLE PRECISION
DEXP (<i>dbl</i>)	DOUBLE PRECISION
DIM (<i>genA</i> , <i>genB</i>)	Same as arguments
DINT (<i>rvalue</i>)	DOUBLE PRECISION
DLOG (<i>dbl</i>)	DOUBLE PRECISION
DLOG10 (<i>dbl</i>)	DOUBLE PRECISION
DMAX1 (<i>dblA</i> , <i>dblB</i> [, <i>dblC</i>]...)	DOUBLE PRECISION
DMIN1 (<i>dblA</i> , <i>dblB</i> [, <i>dblC</i>]...)	DOUBLE PRECISION
DMOD (<i>value</i> , <i>mod</i>)	DOUBLE PRECISION
DNINT (<i>dbl</i>)	DOUBLE PRECISION
DPROD (<i>real4A</i> , <i>real4B</i>)	DOUBLE PRECISION
DREAL (<i>cxvalue</i>)	DOUBLE PRECISION
DSIGN (<i>dblA</i> , <i>dblB</i>)	DOUBLE PRECISION
DSIN (<i>dbl</i>)	DOUBLE PRECISION

DSINH (<i>dbl</i>)	DOUBLE PRECISION
DSQRT (<i>rvalue</i>)	DOUBLE PRECISION
DTAN (<i>dbl</i>)	DOUBLE PRECISION
DTANH (<i>dbl</i>)	DOUBLE PRECISION
EXP (<i>gen</i>)	Same as argument
FLOAT (<i>ivalue</i>)	REAL
IABS (<i>int</i>)	Same as argument
ICHAR (<i>char</i>)	INTEGER
IDIM (<i>intA</i> , <i>intB</i>)	INTEGER
IDINT (<i>dbl</i>)	INTEGER
IDNINT (<i>dbl</i>)	INTEGER
IFIX (<i>real4</i>)	REAL
INDEX (<i>charA</i> , <i>charB</i>)	INTEGER
INT (<i>gen</i>)	INTEGER
ISIGN (<i>intA</i> , <i>intB</i>)	INTEGER
LEN (<i>char</i>)	INTEGER
LGE (<i>charA</i> , <i>charB</i>)	LOGICAL
LGT (<i>charA</i> , <i>charB</i>)	LOGICAL
LLE (<i>charA</i> , <i>charB</i>)	LOGICAL
LLT (<i>charA</i> , <i>charB</i>)	LOGICAL
LOG (<i>gen</i>)	Same as argument
LOG10 (<i>real</i>)	Same as argument
MAX (<i>genA</i> , <i>genB</i> [, <i>genC</i>]...)	INTEGER or REAL
MAX0 (<i>intA</i> , <i>intB</i> [, <i>intC</i>]...)	INTEGER
MAX1 (<i>realA</i> , <i>realB</i> [, <i>realC</i>]...)	INTEGER
MIN (<i>genA</i> , <i>genB</i> [, <i>genC</i>]...)	INTEGER or REAL
MIN0 (<i>intA</i> , <i>intB</i> [, <i>intC</i>]...)	INTEGER
MIN1 (<i>realA</i> , <i>real</i> [, <i>real</i>]...)	INTEGER
MOD (<i>genA</i> , <i>genB</i>)	REAL
NINT (<i>real</i>)	INTEGER
REAL (<i>gen</i>)	REAL
SIGN (<i>genA</i> , <i>genB</i>)	INTEGER or REAL
SIN (<i>gen</i>)	Same as argument
SINH (<i>real</i>)	Same as argument
SNGL (<i>dbl</i>)	REAL
SQRT (<i>gen</i>)	Same as argument
TAN (<i>real</i>)	Same as argument
TANH (<i>real</i>)	Same as argument

FORTRAN 77 Statements

ASSIGN *label* **TO** *variable*

BACKSPACE {*unitspec* |

(**[UNIT=]***unitspec*
 [, **ERR=errlabel**]
 [, **IOSTAT=iocheck**]) }

BLOCK DATA [*blockdataname*]

CALL *sub* [(*actuals*)]

CHARACTER [**chars*] *vname* [**length*] [(*dim*)] [, *vname* [**length*] [(*dim*)]

CLOSE (**[UNIT=]***unitspec*
 [, **ERR=errlabel**]
 [, **IOSTAT=iocheck**]
 [, **STATUS=status**])

COMMON [/*cname* /] *nlist*,] / *cname* /*nlist*] ...

COMPLEX *vnam* [(*dim*)] [, *vname* [(*dim*)]] ...

CONTINUE

DATA *nlist* /*clist*/ [[,] *nlist* /*clist*/] ...

DIMENSION *array* ([*lower:*]*upper* [, { [*lower:*]*upper* }])

DO [*label* [,]] *dovar* = *start*, *stop* [, *inc*]

DOUBLE PRECISION *vname* [(*dim*)] [, *vname* [(*dim*)]] ...

ELSE
statementblock

ELSE IF (*expression*) **THEN**
statementblock

END

END IF

ENDFILE {*unitspec* |
 (**[UNIT=]** *unitspec*
 [, **ERR=errlabel**]
 [, **IOSTAT=iocheck**]) }

ENTRY *ename* [([*formal* [, *formal*] ...])]

EQUIVALENCE (*nlist*) [, (*nlist*)] ...

EXTERNAL *name* [, *name*] ...

FORMAT [*editlist*]

[*type*] **FUNCTION** *func* ([*formal*] [, *formal*] ...)

GOTO *variable* [[,] (*labels*)]

GOTO (*labels*) [,] *n*

GOTO *label*

IF (*expression*) *label1*, *label2*, *label3*

IF (*expression*) *statement*

IF (*expression*) **THEN**

statementblock1

[**ELSE IF** (*expression*) **THEN**

statementblock2] ...

[**ELSE**

statementblock3]

END IF

IMPLICIT *type (letters)* [, *type (letters)*]...

INQUIRE ({[**UNIT**=]*unitspec* | [**FILE**=]*file* }

[, [**ACCESS**=]*access*]

[, [**BLANK**=]*blank*] [, [**DIRECT**=]*direct*] [, [**ERR**=]*errlabel*] [, [**EXIST**=]*exist*] [,

[**FORM**=]*form*]

[, [**FORMATTED**=]*formatted*] [, [**IOSTAT**=]*iocheck*]

[, [**NAME**=]*name*] [, [**NAMED**=]*named*]

[, [**NEXTREC**=]*nextrec*] [, [**NUMBER**=]*num*] [, [**OPENED**=]*opened*]

[, [**RECL**=]*recl*] [, [**SEQUENTIAL**=]*seq*] [, [**UNFORMATTED**=]*unformatted*])

INTEGER *vname* [(*dim*)] [, *vname* [(*dim*)]] ...

INTRINSIC *names*

LOGICAL *vname* [(*dim*)] [, *vname* [(*dim*)]] ...

OPEN ([**UNIT**=]*unitspec* [, [**ACCESS**=]*access*]

[, [**BLANK**=]*blanks*]

[, [**ERR**=]*errlabel*] [, [**FILE**=]*file*]

[, [**FORM**=]*form*] [, [**IOSTAT**=]*iocheck*]

[, [**RECL**=]*recl*] [, [**STATUS**=]*status*])

PARAMETER (*name=constexpr* [, *name=constexpr*]...)

PAUSE [*prompt*]

PRINT { *, |*formatspec* | } [, *iolist*]

PROGRAM *program-name*

READ { *formatspec*, | ([**UNIT**=] *unitspec* [, [**FMT**=]

formatspec] [, [**END**=]*endlabel*] [, [**ERR**=]*errlabel*])

[, **IOSTAT**=*iocheck*] [, **REC**=*rec*])} *iolist*

REAL *vname* [(*dim*)] [, *vname* [(*dim*)]] ...

RETURN [*ordinal*]

REWIND { *unitspec* |
([**UNIT**=]*unitspec*
[, **ERR**=*errlabel*]
[, **IOSTAT**=*iocheck*])}

SAVE [*names*]

STOP [*message*]

SUBROUTINE *subr* [([*formal* [, *formal*] ...])]

WRITE ([**UNIT**=] *unitspec*
[, [**FMT**=] *formatspec*]
[, **ERR**=*errlabel*]
[, **IOSTAT**=*iocheck*]
[, **REC**=*rec*])
iolist

Appendix C: ASCII and Key Code Charts (WNT and W95)

This section contains the following character charts:

- [ASCII Character Codes](#)
- [ANSI Character Codes](#)
- [Key Codes](#)

ASCII Character Codes (WNT and W95)

The ASCII character code tables 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".

For charts of the ASCII character codes, see:

- [ASCII Character Codes Chart 1](#)
- [ASCII Character Codes Chart 2](#)

ASCII Character Codes Chart 1

Ctrl	Dec	Hex	Char	Code	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
^@	0	00		NUL	32	20	sp	64	40	Q	96	60	`
^A	1	01	␣	SOH	33	21	!	65	41	A	97	61	a
^B	2	02	␣	SIX	34	22	"	66	42	B	98	62	b
^C	3	03	␣	EIX	35	23	#	67	43	C	99	63	c
^D	4	04	␣	EOI	36	24	\$	68	44	D	100	64	d
^E	5	05	␣	ENQ	37	25	%	69	45	E	101	65	e
^F	6	06	␣	ACK	38	26	&	70	46	F	102	66	f
^G	7	07	␣	BEL	39	27	'	71	47	G	103	67	g
^H	8	08	␣	BS	40	28	(72	48	H	104	68	h
^I	9	09	␣	HI	41	29)	73	49	I	105	69	i
^J	10	0A	␣	LF	42	2A	*	74	4A	J	106	6A	j
^K	11	0B	␣	VI	43	2B	+	75	4B	K	107	6B	k
^L	12	0C	␣	FF	44	2C	,	76	4C	L	108	6C	l
^M	13	0D	␣	CR	45	2D	-	77	4D	M	109	6D	m
^N	14	0E	␣	SD	46	2E	.	78	4E	N	110	6E	n
^O	15	0F	␣	SI	47	2F	/	79	4F	O	111	6F	o
^P	16	10	␣	SLE	48	30	0	80	50	P	112	70	p
^Q	17	11	␣	CS1	49	31	1	81	51	Q	113	71	q
^R	18	12	␣	DC2	50	32	2	82	52	R	114	72	r
^S	19	13	␣	DC3	51	33	3	83	53	S	115	73	s
^T	20	14	␣	DC4	52	34	4	84	54	T	116	74	t
^U	21	15	␣	NAK	53	35	5	85	55	U	117	75	u
^V	22	16	␣	SYN	54	36	6	86	56	V	118	76	v
^W	23	17	␣	EIB	55	37	7	87	57	W	119	77	w
^X	24	18	␣	CAN	56	38	8	88	58	X	120	78	x
^Y	25	19	␣	EM	57	39	9	89	59	Y	121	79	y
^Z	26	1A	␣	SIB	58	3A	:	90	5A	Z	122	7A	z
^[27	1B	␣	ESC	59	3B	;	91	5B	[123	7B	{
^\	28	1C	␣	FS	60	3C	<	92	5C	\	124	7C	
]`	29	1D	␣	GS	61	3D	=	93	5D]	125	7D	}
^^	30	1E	␣	RS	62	3E	>	94	5E	^	126	7E	~
_	31	1F	␣	US	63	3F	?	95	5F	_	127	7F	Δ†

† ASCII code 127 has the code DEL. Under MS-DOCS, this code has the same effect as ASCII 8 (BS). The DEL code can be generated by the CTRL+BKSP key.

ASCII Character Codes Chart 2 (IBM character set)

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80	À	160	A0	á	192	C0	Ĺ	224	E0	«
129	81	Á	161	A1	â	193	C1	Ľ	225	E1	»
130	82	Â	162	A2	ã	194	C2	ŀ	226	E2	ƒ
131	83	Ã	163	A3	ä	195	C3	Ł	227	E3	Œ
132	84	Ä	164	A4	å	196	C4	ł	228	E4	Š
133	85	Å	165	A5	ä	197	C5	Ń	229	E5	ŧ
134	86	Ā	166	A6	å	198	C6	ń	230	E6	Ũ
135	87	Ā	167	A7	æ	199	C7	Ņ	231	E7	Ų
136	88	Ē	168	A8	ȃ	200	C8	Ŷ	232	E8	Ÿ
137	89	Ē	169	A9	Ȅ	201	C9	ŷ	233	E9	Ž
138	8A	Ĕ	170	AA	ȅ	202	CA	Ź	234	EA	Ω
139	8B	Ĕ	171	AB	Ȇ	203	CB	Ż	235	EB	δ
140	8C	Ė	172	AC	ȇ	204	CC	Ų	236	EC	θ
141	8D	Ė	173	AD	Ȉ	205	CD	Ŵ	237	ED	ϑ
142	8E	Ħ	174	AE	ȉ	206	CE	Ŷ	238	EE	€
143	8F	Ħ	175	AF	Ȋ	207	CF	ŷ	239	EF	ƒ
144	90	Ī	176	B0	ȋ	208	D0	Ź	240	F0	≡
145	91	Ī	177	B1	Ȍ	209	D1	Ź	241	F1	+
146	92	Ī	178	B2	ȍ	210	D2	Ź	242	F2	>
147	93	Ī	179	B3	Ȏ	211	D3	Ź	243	F3	<
148	94	Ī	180	B4	ȏ	212	D4	Ź	244	F4	Ź
149	95	Ī	181	B5	Ȑ	213	D5	Ź	245	F5	Ź
150	96	Ī	182	B6	ȑ	214	D6	Ź	246	F6	÷
151	97	Ī	183	B7	Ȓ	215	D7	Ź	247	F7	Ź
152	98	Ī	184	B8	ȓ	216	D8	Ź	248	F8	Ź
153	99	Ī	185	B9	Ȕ	217	D9	Ź	249	F9	Ź
154	9A	Ī	186	BA	ȕ	218	DA	Ź	250	FA	Ź
155	9B	Ī	187	BB	Ȗ	219	DB	Ź	251	FB	Ź
156	9C	Ī	188	BC	ȗ	220	DC	Ź	252	FC	Ź
157	9D	Ī	189	BD	Ș	221	DD	Ź	253	FD	Ź
158	9E	Ī	190	BE	ș	222	DE	Ź	254	FE	Ź
159	9F	Ī	191	BF	Ț	223	DF	Ź	255	FF	Ź

ANSI Character Codes (WNT and W95)

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.

For a chart of the ANSI character codes, see:

- [ANSI Character Codes Chart](#)

ANSI Character Codes Chart

0	■	32	64	@	96	`	128	■	160	192	à	224	à	
1	■	33	65	A	97	a	129	■	161	á	193	á	225	á
2	■	34	66	B	98	b	130	■	162	â	194	â	226	â
3	■	35	67	C	99	c	131	■	163	ã	195	ã	227	ã
4	■	36	68	D	100	d	132	■	164	ä	196	ä	228	ä
5	■	37	69	E	101	e	133	■	165	å	197	å	229	å
6	■	38	70	F	102	f	134	■	166	æ	198	æ	230	æ
7	■	39	71	G	103	g	135	■	167	ç	199	ç	231	ç
8	■	40	72	H	104	h	136	■	168	è	200	è	232	è
9	■	41	73	I	105	i	137	■	169	é	201	é	233	é
10	■	42	74	J	106	j	138	■	170	ê	202	ê	234	ê
11	■	43	75	K	107	k	139	■	171	ë	203	ë	235	ë
12	■	44	76	L	108	l	140	■	172	ì	204	ì	236	ì
13	■	45	77	M	109	m	141	■	173	í	205	í	237	í
14	■	46	78	N	110	n	142	■	174	î	206	î	238	î
15	■	47	79	O	111	o	143	■	175	ï	207	ï	239	ï
16	■	48	80	P	112	p	144	■	176	ð	208	ð	240	ð
17	■	49	81	Q	113	q	145	■	177	ñ	209	ñ	241	ñ
18	■	50	82	R	114	r	146	■	178	ò	210	ò	242	ò
19	■	51	83	S	115	s	147	■	179	ó	211	ó	243	ó
20	■	52	84	T	116	t	148	■	180	ô	212	ô	244	ô
21	■	53	85	U	117	u	149	■	181	õ	213	õ	245	õ
22	■	54	86	U	118	u	150	■	182	ö	214	ö	246	ö
23	■	55	87	W	119	w	151	■	183	÷	215	÷	247	÷
24	■	56	88	X	120	x	152	■	184	ø	216	ø	248	ø
25	■	57	89	Y	121	y	153	■	185	ù	217	ù	249	ù
26	■	58	90	Z	122	z	154	■	186	ú	218	ú	250	ú
27	■	59	91	[123	{	155	■	187	û	219	û	251	û
28	■	60	92	\	124		156	■	188	ü	220	ü	252	ü
29	■	61	93]	125	}	157	■	189	ý	221	ý	253	ý
30	■	62	94	^	126	~	158	■	190	þ	222	þ	254	þ
31	■	63	95	_	127	■	159	■	191	ÿ	223	ÿ	255	ÿ

■ Indicates that this character isn't supported by Windows.

† Indicates that this character is available only in TrueType fonts.

Key Codes (WNT and W95)

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 charts of the key codes, see:

- [Key Codes Chart 1](#)

• Key Codes Chart 2

Key Codes Chart 1

Key	Scan Code		ASCII or Extended†			ASCII or Extended† with SHIFT			ASCII or Extended† with CTRL			ASCII or Extended† with ALT		
	Dec	Hex	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
ESC	1	01	27	1B	ESC	27	1B	ESC	27	1B	ESC	1	01	NUL§
1!	2	02	49	31	1	33	21	!				120	78	NUL
2@	3	03	50	32	2	64	40	@	3	03	NUL	121	79	NUL
3#	4	04	51	33	3	35	23	#				122	7A	NUL
4\$	5	05	52	34	4	36	24	\$				123	7B	NUL
5%	6	06	53	35	5	37	25	%				124	7C	NUL
6^	7	07	54	36	6	94	5E	^	30	1E	RS	125	7D	NUL
7&	8	08	55	37	7	38	26	&				126	7E	NUL
8*	9	09	56	38	8	42	2A	*				127	7F	NUL
9(10	0A	57	39	9	40	28	(128	80	NUL
0)	11	0B	48	30	0	41	29)				129	81	NUL
_	12	0C	45	2D	-	95	5F	_	31	1F	US	130	82	NUL
=	13	0D	61	3D	=	43	2B	=				131	83	NUL
ESC	14	0E	8	08		8	08		127	7F		14	0E	NUL§
IAB	15	0F	9	09		15	0F	NUL	148	94	NUL§	15	A5	NUL§
Q	16	10	113	71	q	81	51	Q	17	11	DC1	16	10	NUL
W	17	11	119	77	w	87	57	W	23	17	E1B	17	11	NUL
E	18	12	101	65	e	69	45	E	5	05	ENQ	18	12	NUL
R	19	13	114	72	r	82	52	R	18	12	DC2	19	13	NUL
I	20	14	116	74	i	84	54	I	20	14	SO	20	14	NUL
Y	21	15	121	79	y	89	59	Y	25	19	EM	21	15	NUL
U	22	16	117	75	u	85	55	U	21	15	NAK	22	16	NUL
I	23	17	105	69	i	73	49	I	9	09	IAB	23	17	NUL
O	24	18	111	6F	o	79	4F	O	15	0F	SI	24	18	NUL
P	25	19	112	70	p	80	50	P	16	10	DLE	25	19	NUL
[26	1A	91	5B	[123	7B	{	27	1B	ESC	26	1A	NUL§
]	27	1B	98	5D]	125	7D	}	29	1D	GS	27	1B	NUL§
ENTER	28	1C	13	0D	CR	13	0D	CR	10	0A	LF	28	1C	NUL§
ENTER	28	1C	13	0D	CR	13	0D	CR	10	0A	LF	166	A6	NUL§
LCIRL	29	1D												
RCIRL	29	1D												
A	30	1E	97	61	a	65	41	A	1	01	SOH	30	1E	NUL
S	31	1F	115	73	s	83	53	S	19	13	DC3	31	1F	NUL
D	32	20	100	64	d	68	44	D	4	04	EOI	32	20	NUL
F	33	21	102	66	f	70	46	F	6	06	ACK	33	21	NUL
G	34	22	103	67	g	71	47	G	7	07	BEL	34	22	NUL
H	35	23	104	68	h	72	48	H	8	08	BS	35	23	NUL
J	36	24	106	6A	j	74	4A	J	10	0A	LF	36	24	NUL
K	37	25	107	6B	k	75	4B	K	11	0B	VI	37	25	NUL
L	38	26	108	6C	l	76	4C	L	12	0C	FF	38	26	NUL
:	39	27	59	3B	:	58	3A	:				39	27	NUL§
"	40	28	39	27	"	34	22	"				40	28	NUL§
~	41	29	96	60	~	126	7E	~				41	29	NUL§
L SHIFT	42	2A												
\	43	2B	92	5C	\	124	7C		28	1C	FS			
Z	44	2C	122	7A	z	90	5A	Z	26	1A	SUB	44	2C	NUL
X	45	2D	120	78	x	88	58	X	24	18	CAN	45	2D	NUL
C	46	2E	99	63	c	67	43	C	3	03	EIX	46	2E	NUL
V	47	2F	118	76	v	86	56	V	22	16	SYN	47	2F	NUL
B	48	30	98	62	b	66	42	B	2	02	SIX	48	30	NUL
N	49	31	110	6E	n	78	4E	N	14	0E	SO	49	31	NUL
M	50	32	109	6D	m	77	4D	M	13	0D	CR	50	32	NUL
,	51	33	44	2C	,	60	3C	<				51	33	NUL§
.	52	34	46	2E	.	62	3E	>				52	34	NUL§

Key Codes Chart 2

Key	Scan Code		ASCII or Extended†			ASCII or Extended† with SHIFT			ASCII or Extended† with CTRL			ASCII or Extended† with ALT		
	Dec	Hex	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
/?	53	35	47	2F	/	63	3F	?				53	34	NUL‡
GRAY #	53	35	47	2F	/	63	3F	?	149	95	NUL	164	A5	NUL
R SHIFT	54	36												
*PRISC	55	37	42	2A	*	PRISC		↑↑	16	10				
L ALT	56	38												
R ALT#	56	38												
SPACE	57	39	32	20	SPC	32	20	SPC	32	20	SPC	32	20	SPC
CAPS	58	3A												
F1	59	3B	59	3B	NUL	84	54	NUL	94	5E	NUL	104	68	NUL
F2	60	3C	60	3C	NUL	85	55	NUL	95	5F	NUL	105	69	NUL
F3	61	3D	61	3D	NUL	86	56	NUL	96	60	NUL	106	6A	NUL
F4	62	3E	62	3E	NUL	87	57	NUL	97	61	NUL	107	6B	NUL
F5	63	3F	63	3F	NUL	88	58	NUL	98	62	NUL	108	6C	NUL
F6	64	40	64	40	NUL	89	59	NUL	99	63	NUL	109	6D	NUL
F7	65	41	65	41	NUL	90	5A	NUL	100	64	NUL	110	6E	NUL
F8	66	42	66	42	NUL	91	5B	NUL	101	65	NUL	111	6F	NUL
F9	67	43	67	43	NUL	92	5C	NUL	102	66	NUL	112	70	NUL
F10	68	44	68	44	NUL	93	5D	NUL	103	67	NUL	113	71	NUL
F11#	87	57	133	85	E0	135	87	E0	137	89	E0	139	8B	E0
F12#	88	58	134	86	E0	136	88	E0	138	8A	E0	140	8C	E0
NUM	69	45												
SCROLL	70	46												
HOME	71	47	71	47	NUL	55	37	7	119	77	NUL			
HOME#	71	47	71	47	E0	71	47	E0	119	77	E0	151	97	NUL
UP	72	48	72	48	NUL	56	38	8	141	8D	NUL§			
UP#	72	48	72	48	E0	72	48	E0	141	8D	E0	152	98	NUL
PGUP	73	49	73	49	NUL	57	39	9	132	84	NUL			
PGUP#	73	49	73	49	E0	73	49	E0	132	84	E0	153	99	NUL
GRAY-	74	4A				45	2D	-						
LEFT	75	4B	75	4B	NUL	52	34	4	115	73	NUL			
LEFT#	75	4B	75	4B	E0	75	4B	E0	115	73	E0	155	9B	NUL
CENTER	76	4C				53	35	5						
RIGHT	77	4D	77	4D	NUL	54	36	6	116	74	NUL			
RIGHT#	77	4D	77	4D	E0	77	4D	E0	116	74	E0	157	9D	NUL
GRAY+	78	4E				43	2B	+						
END	79	4F	79	4F	NUL	49	31	1	117	75	NUL			
END#	79	4F	79	4F	E0	79	4F	E0	117	75	E0	159	9F	NUL
DOWN	80	50	80	50	NUL	50	32	2	145	91	NUL§			
DOWN#	80	50	80	50	E0	80	50	E0	145	91	E0	160	A0	NUL
PGDN	81	51	81	51	NUL	51	33	3	118	76	NUL			
PGDN#	81	51	81	51	E0	81	51	E0	118	76	E0	161	A1	NUL
INS	82	52	82	52	NUL	48	30	0	146	92	NUL§			
INS#	82	52	82	52	E0	82	52	E0	146	92	E0	162	A2	NUL
DEL	83	53	83	53	NUL	46	2E	.	147	93	NUL§			
DEL#	83	53	83	53	E0	83	53	E0	147	93	E0	163	A3	NUL

† Extended codes return 0 (NUL) or E0 (decimal 224) as the initial character. This is a signal that a second (extended) code is available in the keystroke buffer.

‡ These key combinations are only recognized on extended keyboards.

§ These keys are only available on extended keyboards. Most are in the Cursor/Control cluster. If the raw scan code is read from the keyboard port (60h), it appears as two bytes (E0h) followed by the normal scan code. However, when the keypad ENTER and / keys are read through the BIOS interrupt 16h, only E0h is seen since the interrupt only gives one-byte scan codes.

†† Under MS-DOS, SHIFT + PRISC causes interrupt 5, which prints the screen unless an interrupt handler has been defined to replace the default interrupt 5 handler.

Appendix D: Hexadecimal-Binary-Octal-Decimal Conversions

The following table lists hexadecimal, binary, octal, and decimal conversion:

Hex Number	Binary Number	Octal Number	Decimal Number
0	0000	00	0
1	0001	01	1
2	0010	02	2
3	0011	03	3
4	0100	04	4
5	0101	05	5
6	0110	06	6
7	0111	07	7
8	1000	10	8
9	1001	11	9
A	1010	12	10
B	1011	13	11
C	1100	14	12
D	1101	15	13
E	1110	16	14
F	1111	17	15

Appendix E: Data Representation

This appendix discusses the following topics:

- [Data representation models](#)
- [Data representation](#)

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 integer ranges, see [Integer Data Type](#); on real ranges, see [Real Constants](#).

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(KIND=4) (or INTEGER*4) follows:

$$i = s \times \sum_{k=1}^{31} w_k \times 2^{k-1}$$

The following example demonstrates the general integer model for $i = -20$ using a base (r) of 2:

$$i = (-1) \times (0 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 0 \times 2^3 + 1 \times 2^4)$$

$$i = (-1) \times (4 + 16)$$

$$i = -1 \times 20$$

$$i = -20$$

Model for Real Data

The model set for reals, in general, is defined as one of the following:

$$x = 0$$

$$x = s \times b^e \times \sum_{k=1}^p f_k \times b^{-k}$$

The following values apply to this model set:

- x is the real value.
- s is the sign (either +1 or -1).
- b is the base (real radix; an integer greater than 1).
- p is the number of mantissa digits (an integer greater than 1). The number of digits differs depending on the real format, as follows:

IEEE S_floating	24
DIGITAL VAX F_floating (VMS only)	24
IEEE T_floating	53
DIGITAL VAX D_floating (VMS only)	53 ¹
DIGITAL VAX G_floating (VMS only)	53

¹ The memory format for VAX D_floating format is 56 mantissa digits, but computationally it is 53 digits. It is considered to have 53 digits by DIGITAL Fortran.

- e is an integer in the range e_{\min} to e_{\max} inclusive. This range differs depending on the real format, as follows:

	e_{\min}	e_{\max}
IEEE S_floating	-125	128
DIGITAL VAX F_floating (VMS only)	-127	127
IEEE T_floating	-1021	1024
DIGITAL VAX D_floating (VMS only)	-127	127
DIGITAL VAX G_floating (VMS only)	-1023	1023

- 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(KIND=4) or REAL*4) is defined as one of the following:

$$x = 0$$

$$x = s \times 2^e \times \left[1/2 + \sum_{k=2}^{24} f_k \times 2^{-k} \right], -125 \leq e \leq 128$$

The following example demonstrates 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 demonstrates the bit model for $j = 1001$ (integer 9) using a bit number (s) of 4:

$$\begin{array}{cccc} 1 & 0 & 0 & 1 \\ / & | & | & \backslash \\ w_3 & w_2 & w_1 & w_0 \end{array}$$

$$j = (w_0 \times 2^0) + (w_1 \times 2^1) + (w_2 \times 2^2) + (w_3 \times 2^3)$$

$$j = 1 + 0 + 0 + 8$$

$$j = 9$$

Data Representation

DIGITAL Fortran expects numeric data to be in native little endian order, in which the least-significant, right-most zero bit (bit 0) or byte has a lower address than the most-significant, left-most bit (or byte). For information on using nonnative big endian and DIGITAL VAX

floating-point formats, see [Converting Unformatted Numeric Data](#).

The symbol :A in any figure specifies the address of the byte containing bit 0, which is the starting address of the represented data element.

The following table lists the intrinsic data types used by Visual Fortran, the storage required, and valid ranges.

DIGITAL Fortran Data Types and Storage

Data Type	Storage	Description
BYTE (INTEGER(KIND=1))	1 byte (8 bits)	A BYTE declaration is a signed integer data type equivalent to INTEGER(KIND=1).
INTEGER	See INTEGER(KIND=2), INTEGER(KIND=4), and INTEGER(KIND=8).	Signed integer, either INTEGER(KIND=2) or INTEGER(KIND=4) on x86 systems, or INTEGER(KIND=2), INTEGER(KIND=4), or INTEGER(KIND=8) on Alpha systems. The size is controlled by the /integer_size:nn compiler option. The default is /integer_size:32 (INTEGER(KIND=4)).
INTEGER(KIND=1)	1 byte (8 bits)	Signed integer value from -128 to 127.
INTEGER(KIND=2)	2 bytes (16 bits)	Signed integer value from -32,768 to 32,767.
INTEGER(KIND=4)	4 bytes (32 bits)	Signed integer value from -2,147,483,648 to 2,147,483,647.
INTEGER(KIND=8)	8 bytes (64 bits)	Signed integer value from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
REAL(KIND=4) (REAL)	4 bytes (32 bits)	Single-precision real floating-point values in IEEE S_floating format ranging from 1.17549435E-38 to 3.40282347E38. Values between 1.17549429E-38 and 1.40129846E-45 are denormalized (subnormal).
REAL(KIND=8) DOUBLE PRECISION	8 bytes (64 bits)	Double-precision real floating-point values in IEEE T_floating format ranging from 2.2250738585072013D-308 to 1.7976931348623158D308. Values between 2.2250738585072008D-308 and 4.94065645841246544D-324 are denormalized (subnormal).
COMPLEX(KIND=4) (COMPLEX)	8 bytes (64 bits)	Single-precision complex floating-point values in a pair of IEEE S_floating format parts: real and imaginary. The real and imaginary parts range from 1.17549435E-38 to 3.40282347E38. Values between 1.17549429E-38 and 1.40129846E-45 are denormalized (subnormal).
		Double-precision complex floating-point values in a pair of IEEE T_floating format parts: real and imaginary. The real and imaginary parts each

COMPLEX(KIND=8) DOUBLE COMPLEX	16 bytes (128 bits)	range from 2.2250738585072013D-308 to 1.7976931348623158D308. Values between 2.2250738585072008D-308 and 4.94065645841246544D-324 are denormalized (subnormal).
LOGICAL	See LOGICAL(KIND=2), LOGICAL(KIND=4), and LOGICAL(KIND=8).	Logical value, either LOGICAL(KIND=2) or LOGICAL(KIND=4) on x86 systems, or LOGICAL(KIND=2), LOGICAL(KIND=4), or LOGICAL(KIND=8) on Alpha systems. The size is controlled by the /integer_size:nn compiler option. The default is /integer_size:32 (LOGICAL(KIND=4)).
LOGICAL(KIND=1)	1 byte (8 bits)	Logical values .TRUE. or .FALSE.
LOGICAL(KIND=2)	2 bytes (16 bits)	Logical values .TRUE. or .FALSE.
LOGICAL(KIND=4)	4 bytes (32 bits)	Logical values .TRUE. or .FALSE.
LOGICAL(KIND=8)	8 bytes (64 bits)	Logical values .TRUE. or .FALSE.
CHARACTER	1 byte (8 bits) per character	Character data represented by character code convention. Character declarations can be in the form CHARACTER*n, where n is the number of bytes or n is (*) to indicate passed-length format.
HOLLERITH	1 byte (8 bits) per Hollerith character	Hollerith constants.

In addition, you can define bit constants as explained in the *DIGITAL Fortran Language Reference Manual*.

The following sections discuss the intrinsic data types in more detail:

- [Integer Data Representations](#)
- [Logical Data Representations](#)
- [Native IEEE Floating-Point Representations](#)
- [Character Representation](#)
- [Hollerith Representation](#)

Integer Data Representations

On x86 systems, integer data lengths can be 1-, 2-, or 4-bytes in length.

On Alpha systems, integer data lengths can be 1-, 2- 4-, or 8-bytes in length.

The default data size used for an INTEGER data declaration is INTEGER(KIND=4), unless the /integer_size:16 or (on Alpha systems) the /integer_size:64 option was specified.

Integer data is signed with the sign bit being 0 (zero) for positive numbers and 1 for negative numbers.

On Alpha systems, to improve performance use INTEGER(KIND=4) (or INTEGER(KIND=8)) rather than INTEGER(KIND=2) or INTEGER(KIND=1).

The following sections discuss integer data:

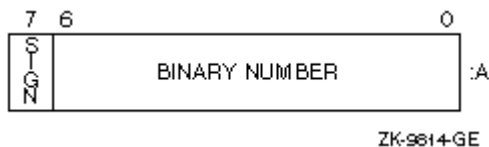
- [INTEGER\(KIND=1\) Representation](#)

- INTEGER(KIND=2) Representation
- INTEGER(KIND=4) Representation
- INTEGER(KIND=8) Representation (Alpha systems)

INTEGER(KIND=1) Representation

INTEGER(KIND=1) values range from -128 to 127 and are stored in 1 byte, as shown below.

Figure: INTEGER(KIND=1) Data Representation



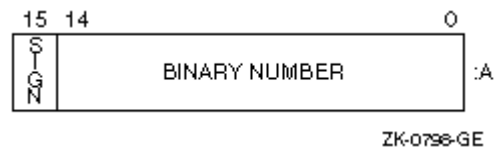
Integers are stored in a two's complement representation. For example:

+22 = 16 (hex)
-7 = F9 (hex)

INTEGER(KIND=2) Representation

INTEGER(KIND=2) values range from -32,768 to 32,767 and are stored in 2 contiguous bytes, as shown below:

Figure: INTEGER(KIND=2) Data Representation



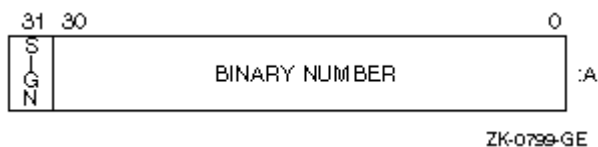
Integers are stored in a two's complement representation. For example:

+22 = 0016 (hex)
-7 = FFF9 (hex)

INTEGER(KIND=4) Representation

INTEGER(KIND=4) values range from -2,147,483,648 to 2,147,483,647 and are stored in 4 contiguous bytes, as shown below.

Figure: INTEGER(KIND=4) Data Representation

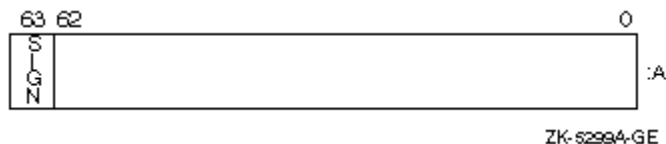


INTEGER(KIND=4) values are stored in a two's complement representation.

INTEGER(KIND=8) Representation (Alpha only)

On Alpha systems only, INTEGER(KIND=8) values range from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 and are stored in 8 contiguous bytes, as shown below.

Figure: INTEGER(KIND=8) Data Representation



INTEGER(KIND=8) values are stored in a two's complement representation.

Logical Data Representations

On x86 systems, logical data lengths can be 1-, 2-, or 4-bytes in length.

On Alpha systems, logical data lengths can be 1-, 2-, 4-, or 8-bytes in length.

The default data size used for a LOGICAL data declaration is LOGICAL(KIND=4), unless the /integer_size:16 or /integer_size:64 (Alpha systems) option was specified.

To improve performance on Alpha systems, use LOGICAL(KIND=4) (or LOGICAL(KIND=8)) rather than LOGICAL(KIND=2) or LOGICAL(KIND=1).

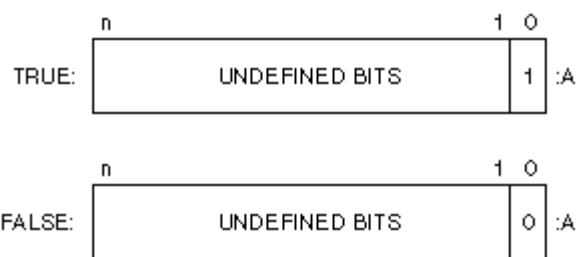
LOGICAL(KIND=1) values are stored in 1 byte. In addition to having logical values .TRUE. and .FALSE., LOGICAL(KIND=1) data can also have values in the range -128 to 127. Logical variables can also be interpreted as integer data.

In addition to LOGICAL(KIND=1), logical values can also be stored in 2 (LOGICAL(KIND=2)), 4 (LOGICAL(KIND=4)), or 8 (LOGICAL(KIND=8)) contiguous bytes, starting on an arbitrary byte boundary. LOGICAL(KIND=8) data is available on Alpha systems only.

If the /fpscomp:nological option is set, the low-order bit determines whether the logical value is true or false. Specify /fpscomp:logical for Microsoft Fortran Powerstation logical values, where 0 (zero) is false and non-zero values are true.

LOGICAL(KIND=1), LOGICAL(KIND=2), LOGICAL(KIND=4), and LOGICAL(KIND=8) data representation (when /fpscomp:nological option was set) appears below.

LOGICAL(KIND=1), LOGICAL(KIND=2), LOGICAL(KIND=4), and LOGICAL(KIND=8) Data Representation



Key: n = 7, 15, 31, or 63 depending on LOGICAL declaration size

ZK-83004-GE

Native IEEE Floating-Point Representations

The REAL(KIND=4) (S_floating) and REAL(KIND=8) (T_floating) formats are stored in standard little endian IEEE binary floating-point notation. (See IEEE Standard 754 for additional information about IEEE binary floating point notation.) COMPLEX(KIND=4) and COMPLEX(KIND=8) formats use a pair of REAL(KIND=4) or REAL(KIND=8) values to denote the real and imaginary parts of the data.

For IEEE S_floating and T_floating formats, fractions are represented in sign-magnitude notation, with the binary radix point to the right of the most-significant bit. Fractions are assumed to be normalized, and therefore the most-significant bit is not stored (this is called "hidden bit normalization"). This bit is assumed to be 1 unless the exponent is 0. If the exponent equals 0, then the value represented is denormalized (subnormal) or plus or minus zero.

The following sections discuss floating-point data:

- [REAL\(KIND=4\) \(REAL\) Representation](#)
- [REAL\(KIND=8\) \(DOUBLE PRECISION\) Representation](#)
- [COMPLEX\(KIND=4\) \(COMPLEX\) Representation](#)
- [COMPLEX\(KIND=8\) \(DOUBLE COMPLEX\) Representation](#)

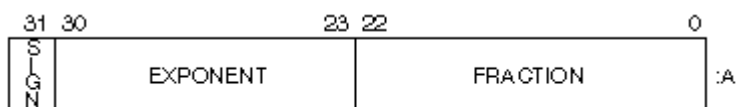
For more information on:

- Using the Bitviewer tool, see [Viewing Floating-Point Representations with BitViewer](#).
- Reading or writing floating-point data other than native IEEE little endian data, see [Converting Unformatted Numeric Data](#).
- Using floating-point numbers, [The Floating-Point Environment](#).

REAL(KIND=4) (REAL) Representation

REAL(KIND=4) data occupies 4 contiguous bytes stored in IEEE S_floating format. Bits are labeled from the right, 0 through 31, as shown below.

REAL(KIND=4) Floating-Point Data Representation



ZK-9816-GE

The form of REAL(KIND=4) data is sign magnitude, with bit 31 the sign bit (0 for positive numbers,

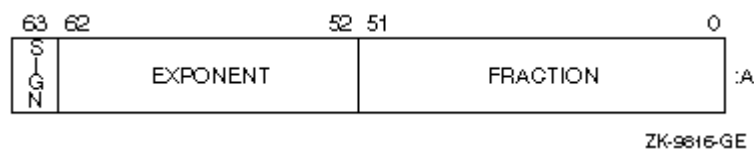
1 for negative numbers), bits 30:23 a binary exponent in excess 127 notation, and bits 22:0 a normalized 24-bit fraction including the redundant most-significant fraction bit not represented.

The value of data is in the approximate range: 1.17549435E-38 (normalized) to 3.40282347E38. The IEEE denormalized (subnormal) limit is 1.40129846E-45. The precision is approximately one part in 2^{23} ; typically 7 decimal digits.

REAL(KIND=8) (DOUBLE PRECISION) Representation

REAL(KIND=8) data occupies 8 contiguous bytes stored in IEEE T_floating format. Bits are labeled from the right, 0 through 63, as shown below.

REAL(KIND=8) Floating-Point Data Representation



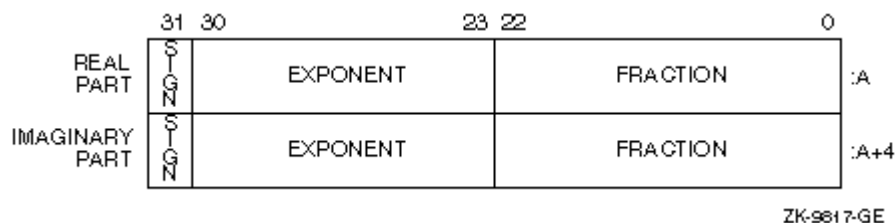
The form of REAL(KIND=8) data is sign magnitude, with bit 63 the sign bit (0 for positive numbers, 1 for negative numbers), bits 62:52 a binary exponent in excess 1023 notation, and bits 51:0 a normalized 53-bit fraction including the redundant most-significant fraction bit not represented.

The value of data is in the approximate range: 2.2250738585072013D-308 (normalized) to 1.7976931348623158D308. The IEEE denormalized (subnormal) limit is 4.94065645841246544D-324. The precision is approximately one part in 2^{52} ; typically 15 decimal digits.

COMPLEX(KIND=4) (COMPLEX) Representation

COMPLEX(KIND=4) data is 8 contiguous bytes containing a pair of REAL(KIND=4) values stored in IEEE S_floating format. The low-order 4 bytes contain REAL(KIND=4) data that represents the real part of the complex number. The high-order 4 bytes contain REAL(KIND=4) data that represents the imaginary part of the complex number, as shown below.

COMPLEX(KIND=4) Floating-Point Data Representation

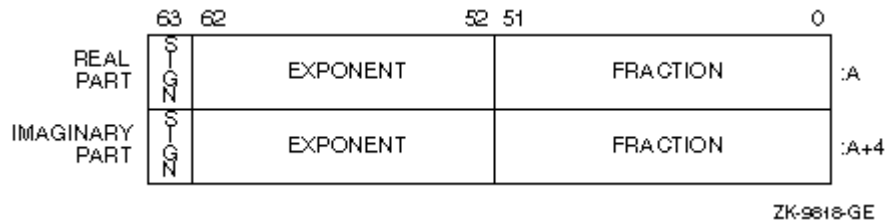


The limits and underflow characteristics for REAL(KIND=4) apply to the two separate real and imaginary parts of a COMPLEX(KIND=4) number. Like REAL(KIND=4) numbers, the sign bit representation is 0 (zero) for positive numbers and 1 for negative numbers.

COMPLEX(KIND=8) (DOUBLE COMPLEX) Representation

COMPLEX(KIND=8) (same as COMPLEX*16) data is 16 contiguous bytes containing a pair of REAL(KIND=8) values stored in IEEE T_floating format. The low-order 8 bytes contain REAL(KIND=8) data that represents the real part of the complex data. The high-order 8 bytes contain REAL(KIND=8) data that represents the imaginary part of the complex data, as shown below.

COMPLEX(KIND=8) Floating-Point Data Representation

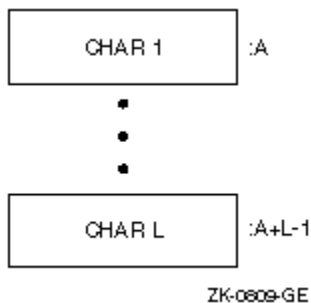


The limits and underflow characteristics for REAL(KIND=8) apply to the two separate real and imaginary parts of a COMPLEX(KIND=8) number. Like REAL(KIND=8) numbers, the sign bit representation is 0 (zero) for positive numbers and 1 for negative numbers.

Character Representation

A character string is a contiguous sequence of bytes in memory, as shown below.

Figure: CHARACTER Data Representation



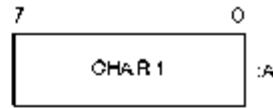
A character string is specified by two attributes: the address A of the first byte of the string, and the length L of the string in bytes. The length L of a string is in the range 1 through 65,535.

Hollerith Representation

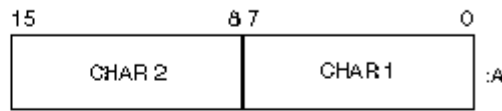
Hollerith constants are stored internally, one character per byte, as shown below.

Figure: Hollerith Data Representation

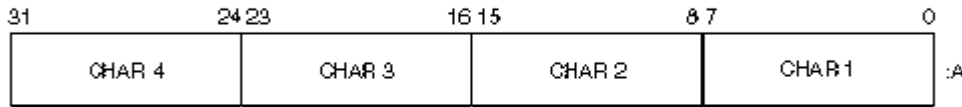
1 Byte



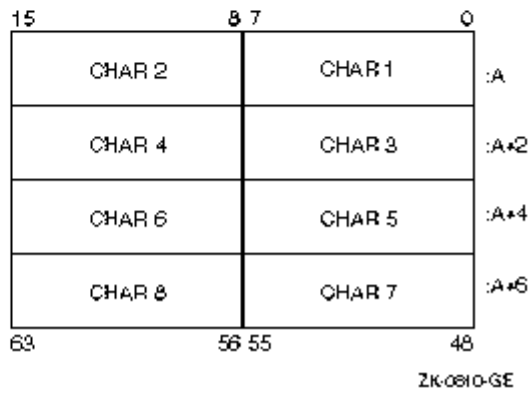
2 Bytes



4 Bytes



8 Bytes



Appendix F: Summary of Language Extensions

This appendix summarizes the DIGITAL Fortran language extensions to the ANSI/ISO Fortran 90 Standard.

For more information, see the following sections:

- [DIGITAL Fortran Language Extensions](#)
- [High Performance Fortran Language Extensions](#)

DIGITAL Fortran Language Extensions

This section summarizes the DIGITAL Fortran language extensions. Most extensions are available on all systems, but some extensions are limited to certain systems. If an extension is limited, it is labeled.

Extensions in the following topics are discussed:

- [Source Forms](#)
- [Characters in Names](#)
- [Character Sets](#)
- [Intrinsic Data Types](#)
- [Constants](#)
- [Derived Data Types](#)
- [Arrays](#)
- [Expressions](#)
- [Specification Statements](#)
- [Procedures](#)
- [Compilation Control Statements](#)
- [Built-In Functions](#)
- [I/O Statements](#)
- [I/O Formatting](#)
- [File Operation Statements](#)
- [Compiler Directives](#)
- [Additional Language Features](#)
- [Intrinsic Procedures](#)

For more information, see [High Performance Fortran Language Extensions](#).

Source Forms

The following are allowed as 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

Characters in Names

As an extension, the dollar sign (\$) is allowed as a valid character in names.

Character Sets

The following are extensions to the Fortran 90 character set:

- The Tab character
- The DEC Multinational extension to the ASCII character set (VMS, U*X)¹
- ASCII Character Code Chart 2--IBM Character Set (WNT, W95)
- ANSI Character Codes (WNT, W95)
- Key Codes (WNT, W95)

¹ See the Language Reference Manual.

Intrinsic Data Types

The following are data-type extensions:

```
BYTE LOGICAL*1 INTEGER*1 REAL*4 COMPLEX*8
      LOGICAL*2 INTEGER*2 REAL*8 COMPLEX*16
      LOGICAL*4 INTEGER*4 REAL*16
      LOGICAL*8 INTEGER*8
```

Constants

Hollerith constants are allowed as an extension.

C Strings are allowed as extensions in character constants.

Derived Data Types

As an extension, default initial values for derived-type components can be specified in a derived-type definition.

Arrays

As an extension, arrays declared using the ALLOCATABLE attribute can be automatically deallocated.

Expressions

When operands of different intrinsic data types are combined in expressions, conversions are performed as necessary.

Binary, octal, hexadecimal, and Hollerith constants can appear wherever numeric constants are allowed.

The following are extensions allowed in logical expressions:

- `.XOR.` as a synonym for `.NEQV.`
- Integers as valid logical items

As an extension, the **WHERE** construct can include nested **WHERE** constructs and a masked **ELSEWHERE** statement. **WHERE** constructs can also be named.

Specification Statements

The following specification attributes and statements are extensions:

- AUTOMATIC attribute and statement
- STATIC attribute and statement
- VOLATILE attribute and statement

Procedures

The ELEMENTAL and PURE prefixes are allowed in user-defined functions and subroutines as extensions.

As an extension, the END INTERFACE statement of an interface block defining a generic routine can specify a generic identifier.

Compilation Control Statements

The following statements are extensions that can influence compilation:

- INCLUDE statement format (VMS only):

```
INCLUDE '[text-lib] (module-name) [/[NO]LIST]'
```

- OPTIONS statement:

<code>/ASSUME =</code>	<code>[NO]UNDERSCORE (Alpha only)</code>
<code>/CHECK =</code>	<code>ALL</code>
	<code>[NO]BOUNDS</code>
	<code>[NO]OVERFLOW</code>
	<code>[NO]UNDERFLOW</code>
	<code>NONE</code>
<code>/NOCHECK</code>	
<code>/CONVERT =</code>	<code>BIG_ENDIAN</code>
	<code>CRAY</code>
	<code>FDX</code>
	<code>FGX</code>
	<code>IBM</code>

```

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

```

Built-In Functions

The %VAL, %REF, %DESCR, and %LOC built-in functions are extensions.

I/O Statements

The following I/O statements and specifiers are extensions:

- ACCEPT statement
- REWRITE statement
- TYPE statements as synonyms for PRINT statements
- A key-field-value specifier as a control list parameter (VMS only)
- A key-of-reference specifier as a control list parameter (VMS only)
- Indexed READ Statement (VMS only)
- Indexed WRITE Statement (VMS only)

As an extension, comments (beginning with !) are allowed in namelist input data.

I/O Formatting

The following are extensions allowed in I/O editing:

- The letter Q as an edit descriptor
- The dollar sign (\$) as an edit descriptor
- The backslash (\) as an edit descriptor
- The dollar sign as a carriage control character
- ASCII NUL as a carriage control character
- Variable format expressions
- On output, when using I, B, O, Z, and F edit descriptors, the specified value of the field width can be zero.

File Operation Statements

The following statement specifiers and statements are extensions:

- CLOSE statement specifiers:
 - STATUS values: 'SAVE' (as a synonym for 'KEEP'), 'PRINT', 'PRINT/DELETE', 'SUBMIT', 'SUBMIT/DELETE'
 - DISPOSE (or DISP)
- DELETE statement
- INQUIRE statement specifiers:
 - ACCESS value: 'KEYED' (VMS only)
 - BINARY (WNT, W95)
 - BLOCKSIZE (WNT, W95)
 - CARRIAGECONTROL
 - CONVERT
 - DEFAULTFILE
 - FORM value: 'UNKNOWN' and 'BINARY' (WNT, W95)
 - KEYED (VMS only)
 - IOFOCUS (WNT, W95)
 - MODE as a synonym for ACTION (WNT, W95)
 - ORGANIZATION
 - RECORDTYPE
 - SHARE (WNT, W95)
- OPEN statement specifiers:
 - ACCESS values: 'KEYED' (VMS only) and 'APPEND'
 - ASSOCIATEVARIABLE
 - BLOCKSIZE
 - BUFFERCOUNT
 - CARRIAGECONTROL
 - CONVERT
 - DEFAULTFILE
 - DISPOSE
 - EXTENDSIZE (VMS only)
 - FORM value: 'BINARY' (WNT, W95)
 - INITIALSIZE (VMS only)
 - IOFOCUS (WNT, W95)
 - KEY (VMS only)
 - MAXREC
 - MODE as a synonym for ACTION (WNT, W95)
 - NAME as a synonym for FILE
 - NOSPANBLOCKS (VMS only)
 - ORGANIZATION
 - READONLY
 - RECORDSIZE as a synonym for RECL
 - RECORDTYPE
 - SHARE (WNT, W95)
 - SHARED
 - TITLE (WNT, W95)
 - TYPE as a synonym for STATUS
 - USEROPEN
- UNLOCK statement

Compiler Directives

The following General Directives are extensions:

- ALIAS
- ATTRIBUTES
- DECLARE and NODECLARE
- DEFINE and UNDEFINE
- FIXEDFORMLINESIZE
- FREEFORM and NOFREEFORM
- IDENT
- IF and IF DEFINED
- INTEGER
- MESSAGE
- OBJCOMMENT
- OPTIONS
- PACK
- PSECT
- REAL
- STRICT and NOSTRICT
- SUBTITLE
- TITLE

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
- An alternative syntax for octal and hexadecimal constants
- An alternative syntax for an I/O record specifier
- An alternate syntax for the DELETE statement
- Alternative form for namelist external records
- The DIGITAL Fortran 77 POINTER statement
- Structures and records

Intrinsic Procedures

The following intrinsic procedures are extensions:

<u>ACOSD</u>	<u>COTAN</u>	<u>IARGCOUNT</u> ²	<u>NWORKERS</u>
<u>ASIND</u>	<u>CPU_TIME</u>	<u>IARGPTR</u>	<u>QEXT</u> ³
<u>ATAND</u>	<u>DATE</u>	<u>IBCHNG</u>	<u>QFLOAT</u> ³
<u>ATAN2D</u>	<u>DCMPLX</u>	<u>IDATE</u>	<u>RAN</u>
<u>CDABS</u> ¹	<u>DFLOAT</u>	<u>ISHA</u>	<u>RANDU</u>

<u>CDCOS</u> ¹	<u>DREAL</u>	<u>ISHC</u>	<u>SECNDS</u>
<u>CDEXP</u> ¹	<u>EOF</u>	<u>ISHL</u>	<u>SIND</u>
<u>CDLOG</u> ¹	<u>ERRSNS</u>	<u>ISNAN</u>	<u>SIZEOF</u>
<u>CDSIN</u> ¹	<u>EXIT</u>	<u>LOC</u>	<u>TAND</u>
<u>CDSQRT</u> ¹	<u>FP_CLASS</u>	<u>MALLOC</u>	<u>TIME</u>
<u>COSD</u>	<u>FREE</u>	<u>NULL</u>	<u>ZEXT</u>

¹Double precision complex intrinsics can also begin with the letter Z. For example, **CDABS** can also be spelled **ZABS**.

² VMS only

³ VMS, U*X

The following INTEGER(8) specific functions are extensions available on Alpha processors:

<u>AKMAX0</u>	<u>KIBCLR</u>	<u>KIFIX</u>	<u>KMIN0</u>
<u>AKMIN0</u>	<u>KIBITS</u>	<u>KINT</u>	<u>KMIN1</u>
<u>BKTEST</u>	<u>KIBSET</u>	<u>KIOR</u>	<u>KMOD</u>
<u>DFLOTK</u>	<u>KIDIM</u>	<u>KISHFT</u>	<u>KNINT</u>
<u>FLOATK</u>	<u>KIDINT</u>	<u>KISIGN</u>	<u>KNOT</u>
<u>KIABS</u>	<u>KIDNNT</u>	<u>KMAX0</u>	<u>KZEXT</u>
<u>KIAND</u>	<u>KIEOR</u>	<u>KMAX1</u>	

As an extension, the keyword **KIND** can be specified for CEILING and FLOOR.

As an extension, SIGN can distinguish between positive and negative zero.

High Performance Fortran Language Extensions

This section summarizes the High Performance Fortran language extensions to the Fortran 90 standard.

The following extensions are discussed:

- Data Parallel Statements
- Procedure Prefixes
- Intrinsic Procedures

For more information, see DIGITAL Fortran Language Extensions.

Data Parallel Statements

The following statement and construct are extensions:

- FORALL statement
- FORALL construct with multiple assignments

Procedure Prefixes

The following prefixes are allowed in functions and subroutines as extensions:

- PURE
- EXTRINSIC (HPF) - functional only on DIGITAL UNIX systems
- EXTRINSIC (HPF_LOCAL) - functional only on DIGITAL UNIX systems
- EXTRINSIC (HPF_SERIAL) - functional only on DIGITAL UNIX systems

Intrinsic Procedures

System Inquiry Intrinsic Procedures

The following intrinsic procedures are extensions:

- NUMBER OF PROCESSORS intrinsic function
- PROCESSORS_SHAPE intrinsic function

Computational Intrinsic Functions

The argument DIM is an extension in the MAXLOC and MINLOC intrinsic functions.

Bit Manipulation Functions

The ILEN intrinsic function is an extension.

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 DIGITAL UNIX, Windows NT, and Windows 95 systems, a directory path specified in fixed relationship to the root directory. On DIGITAL UNIX systems, the first character is a slash (/). On Windows NT and Windows 95 systems, the first character is a backslash (\).

Active screen buffer

The screen buffer that is currently displayed in a console's window.

Active window

A top-level window of the application with which the user is working. Windows identifies the active window by highlighting its title bar and border.

Actual argument

An expression, variable, procedure, or alternate return specifier which is specified in a subroutine or function reference. The value is passed from the calling program unit to a subprogram.

Adjustable array

An explicit-shape array that is a dummy argument to a subprogram. The term is from FORTRAN-77. *See also* [Explicit-shape array](#).

Aggregate reference

A reference to a record structure field.

Allocatable array

A named array that has the ALLOCATABLE attribute. Once space has been allocated for this type of array, the array has a shape and can be defined (and redefined) or referenced. It is an error to allocate an allocatable array that is currently allocated.

Allocation status

Indicates whether an allocatable array or pointer is allocated. An allocation status is one of: allocated, deallocated, or undefined. An undefined allocation status means an array can no longer be referenced, defined, allocated, or deallocated. *See also* [Association status](#).

Alphanumeric

Pertaining to letters and digits.

Alternate return

A subroutine argument that permits control to branch immediately to some position other than the statement following the call. The actual argument in an alternate return is the statement label to which control should be transferred. (An alternate return is an obsolescent feature in Fortran 90.)

ANSI

The American National Standards Institute. An organization through which accredited organizations create and maintain voluntary industry standards.

Argument

See [Actual argument](#) and [Dummy argument](#).

Argument association

The relationship (or "matching up") between an actual argument and dummy argument during the execution of a procedure reference.

Argument keyword

The name of a dummy (formal) argument. It can be used in a procedure reference before the equals sign [keyword = actual argument] provided the procedure has an explicit interface. This association allows actual arguments to appear in any order.

Argument keywords are supplied for many of the intrinsic procedures.

Array

A set of scalar data that all have the same type and kind type parameters. An array can be referenced by element (using a subscript), by section (using a section subscript list), or as a whole.

An array has a rank (up to 7), bounds, size, and a shape.

An individual array element is a scalar object. An array section, which is itself an array, is a subset of the entire array.

Contrast with [Scalar](#). See also [Bounds](#), [Conformable](#), [Shape](#), and [Size](#).

Array constructor

A mechanism used to specify a sequence of scalar values that produce a rank-one array.

To construct an array of rank greater than one, you must apply the RESHAPE intrinsic function to the array constructor.

Array element

A scalar item in an array. An array element is identified by the array name followed by one or more subscripts in parentheses, indicating the element's position in the array. For example, B(3) or A(2,5).

Array pointer

A pointer to an array. *See also [Array](#) and [Pointer](#).*

Array section

A subobject (or portion) of an array. It consists of the set of array elements or substrings of this set. The set (or section subscript list) is specified by subscripts, subscript triplets, and vector subscripts. If the set does not contain at least one subscript triplet or vector subscript, the reference indicates an array element, not an array.

Array specification

A program statement specifying an array name and the number of dimensions the array contains (its rank). An array specification can appear in a DIMENSION or COMMON statement, or in a type declaration statement.

ASCII

The American Standard Code for Information Interchange. A 7-bit character encoding scheme associating an integer from 0 through 127 with 128 characters.

Assignment

A statement in the form variable = expression. The statement assigns (stores) the value of an expression on the right of an equal sign to the storage location of the variable to the left of the equal sign. In the case of Fortran 90 pointers, the storage location is assigned, not the pointer itself.

Association

An assignment of names, pointers, or storage locations which identifies one entity with several names in the same or different scoping units. The principal kinds of association are argument association, host association, pointer association, storage association, and use association.

Association status

Indicates whether or not a pointer is associated with a target. An association status is one of: undefined, associated, or disassociated. An undefined association status means a pointer can no longer be referenced, defined, or deallocated. An undefined pointer can, however, be allocated, nullified, or pointer assigned to a new target. *See also [Allocation status](#).*

Assumed-length character argument

A dummy argument that assumes the length attribute of the corresponding actual argument. An asterisk (*) specifies the length of the dummy character argument.

Assumed-shape array

A dummy argument array that assumes the shape of its associated actual argument array.

Assumed-size array

A dummy array that takes the size of the actual argument passed to it. The rank, extents, and bounds of the dummy array are specified in its declaration, except for the upper bound (which is specified by a *) and the extent of the last dimension.

Attribute

A property of a data object that can be specified in a type declaration statement. These properties determine how the data object can be used in a program.

Glossary B

Background window

Any window created by a thread other than the foreground thread.

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.

Comment

Text that documents or explains a program. In free source form, a comment begins with an exclamation point (!), unless it appears in a Hollerith or character constant.

In fixed and tab source form, a comment begins with a letter C or an asterisk (*) in column 1. A comment can also begin with an exclamation point anywhere in a source line (except in a Hollerith or character constant) or in column 6 of a fixed-format line. The comment extends from the exclamation point to the end of the line.

The compiler does not process comments, but shows them in program listings. *See also* [Compiler directive](#).

Common block

A physical storage area shared by one or more program units. This storage area is defined by a COMMON statement. If the common block is given a name, it is a named common block; if it is not given a name, it is a blank common.

Compilation unit

The source file or files that are compiled together to form a single object file, possibly using interprocedural optimization across source files. Only one f90 command is used for each compilation, but one f90 command can specify that multiple compilation units be used.

Compiler directive

A structured comment that tells the compiler to perform certain tasks when it compiles a source program unit. Compiler directives are usually compiler-specific. (Some Fortran compilers call these directives "metacommands".)

Complex constant

A constant that is a pair of real or integer constants representing a complex number; the pair is separated by a comma and enclosed in parentheses. The first constant represents the real part of the number; the second constant represents the imaginary part. In DIGITAL Fortran, there are two types of complex constants: COMPLEX (COMPLEX(KIND=4)) and DOUBLECOMPLEX (COMPLEX(KIND=8)).

Complex type

A data type that represents the values of complex numbers. The value is expressed as a complex constant. *See also* [Data type](#).

Component

A part of a derived-type definition. There must be at least one component (intrinsic or derived type) in every derived-type definition.

Concatenate

The combination of two items into one by placing one of the items after the other. In Fortran 90, the concatenation operator (//) is used to combine character items. *See also* [Character expression](#).

Conformable

Pertains to dimensionality. Two arrays are conformable if they have the same shape. A scalar is conformable with any array.

Console

An interface that provides input and output to character-mode applications.

Constant

A data object whose value does not change during the execution of a program; the value is defined at the time of compilation. A constant can be named (using the PARAMETER attribute or statement) or unnamed. An unnamed constant is called a literal constant. The value of a constant can be numeric or logical, or it can be a character string. *Contrast with* [Variable](#).

Constant expression

An expression whose value does not change during program execution.

Construct

A block of statements, beginning with CASE, DO, IF, FORALL, or WHERE statement, and ending with the appropriate termination statement.

Contiguous

Pertaining to entities that are adjacent (next to one another) without intervening blanks (spaces); for example, contiguous characters or contiguous areas of storage.

Control character

A character string, usually with an ASCII value between 0 and 31, used to communicate with devices such as printers, modems, and the like.

Control edit descriptor

A format descriptor that directly displays text or affects the conversions performed by subsequent data edit descriptors. Except for the slash descriptor, control edit descriptors are nonrepeatable.

Control statement

A statement that alters the normal order of execution by transferring control to another part of a program unit or a subprogram. A control statement can be conditional (such as the IF construct or computed GO TO statement) or unconditional (such as the STOP or GO TO statement).

Critical section

An object used to synchronize the threads of a single process. Only one thread at a time can own a critical-section object.

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 90 modules, derived types, and defined operators, support this programming paradigm.

Data edit descriptor

A repeatable format descriptor that causes the transfer or conversion of data to or from its internal representation. In FORTRAN-77, this term was called a field descriptor.

Data entity

A data object that has a data type. It is the result of the evaluation of an expression, or the result of the execution of a function reference (the function result).

Data item

A unit of data (or value) to be processed. Includes constants, variables, arrays, character substrings, or records.

Data object

A constant, variable, or part (subobject) of a constant or variable. Its type may be specified implicitly or explicitly.

Data type

The properties and internal representation that characterize data and functions. Each intrinsic and user-defined data type has a name, a set of operators, a set of values, and a way to show these values in a program. The basic intrinsic data types are integer, real, complex, logical, and character. The data value of an intrinsic data type depends on the value of the type parameter. *See also* [Type parameter](#).

Data type length specifier

The form *n appended to DIGITAL Fortran-specific data type names. For example, in REAL*4, the *4 is the data type length specifier.

Deadlock

A bug where the execution of thread A is blocked indefinitely waiting for thread B to perform some action, while thread B is blocked waiting for thread A. For example, two threads on opposite ends of a named pipe can become deadlocked if each thread waits to read data written by the other thread. A single thread can also deadlock itself. *See also* [Thread](#)

Declaration

A statement or series of statements which specify attributes and properties of named entities, such as specifying the data type of named data objects. Declaration is a synonym for specification.

Decorated name

An internal representation of a procedure name or variable name that contains information about

where it is declared; for procedures, the information includes how it is called. Decorated names are mainly of interest in mixed-language programming, when calling Fortran routines from other languages.

Default character

The kind type for character constants if no kind type parameter is specified. Currently, the only kind type parameter for character constants is CHARACTER(KIND=1), the default character kind.

Default complex

The kind type for complex constants if no kind type parameter is specified. The default complex kind is affected by the compiler option specifying real size. If no compiler option is specified, default complex is COMPLEX(KIND=8) (COMPLEX*8). *See also* [Default real](#).

Default integer

The kind type for integer constants if no kind type parameter is specified. The default integer kind is affected by compiler options specifying integer size. If no compiler option is specified, default integer is INTEGER(KIND=4) (INTEGER*4).

If a command line option affecting integer size has been specified, the integer has the kind specified, unless it is outside the range of the kind specified by the option. In this case, the kind type of the integer is the smallest integer kind which can hold the integer.

Default logical

The kind type for logical constants if no kind type parameter is specified. The default logical kind is affected by compiler options specifying integer size. If no compiler option is specified, default logical is LOGICAL(KIND=4) (LOGICAL*4). *See also* [Default integer](#).

Default real

The kind type for real constants if no kind type parameter is specified. The default real kind is affected by the compiler option specifying real size. If no compiler option is specified, default real is REAL(KIND=4) (REAL*4).

If a real constant is encountered that is outside the range for the default, an error occurs.

Deferred-shape array

An array pointer (an array with the POINTER attribute) or an allocatable array (an array with the ALLOCATABLE attribute). The size in each dimension is determined by pointer assignment or when the array is allocated.

The declared bounds are specified by a colon (:).

Definable

A property of variables. A variable is definable if its value can be changed by the appearance of its name or designator on the left of an assignment statement. An example of a variable that is not definable is an allocatable array that has not been allocated.

Define

(1) To give a value to a data object during program execution. (2) To declare derived types and procedures.

Defined assignment

An assignment statement that is not intrinsic, but is defined by a subroutine and an interface block. *See also* [Derived type](#).

Defined operation

An operation that is not intrinsic, but is defined by a function subprogram containing a generic interface block with the specifier OPERATOR. *See also* [Interface block](#).

Denormalized number

A computational floating-point result smaller than the lowest value in the normal range of a data type (the smallest representable normalized number). You cannot write a constant for a

denormalized number.

Derived type

A data type that is user-defined and not intrinsic. It requires a type definition to name the type and specify its components (which can be intrinsic or user-defined types). A structure constructor can be used to specify a value of derived type. A component of a structure is referenced using a percent sign (%).

Operations on objects of derived types (structures) must be defined by a function with an OPERATOR interface. Assignment for derived types can be defined intrinsically, or be redefined by a subroutine with an ASSIGNMENT interface. Structures can be used as procedure arguments and function results, and can appear in input and output lists. Also called a user-defined type. *See also* [Record](#), the first definition.

Designator

A name that references a subobject (part of an object). A designator is the name of the object followed by a selector that selects the subobject. For example, B(3) is a designator for an array element. Also called a subobject designator. *See also* [Selector](#) and [Subobject](#).

Dimension

A range of values for one subscript or index of an array. An array can have from 1 to 7 dimensions. The number of dimensions is the rank of the array.

Dimension bounds

See [Bounds](#).

Direct access

A method for retrieving or storing data in which the data (record) is identified by the record number, or the position of the record in the file. The record is accessed directly (nonsequentially); therefore, all information is equally accessible. Also called random access. *Contrast with* [Sequential access](#).

DLL

See [Dynamic Link Library](#).

Double-byte character set (DBCS)

A mapping of characters to their identifying numeric values, in which each value is 2 bytes wide. Double-byte character sets are sometimes used for languages that have more than 256 characters. *See also* [Multibyte Character Set](#).

Double-precision constant

A processor approximation to the value of a real number that occupies 8 bytes of memory and can assume a positive, negative, or zero value. The precision is greater than a constant of real (single-precision) type. For the precise ranges of the double-precision constants, see your user manual. *See also* [Denormalized number](#).

Driver program

On Windows NT, Windows 95, and DIGITAL UNIX systems, a program that is the user interface to the language compiler. It accepts command line options and file names and causes one or more language utilities or system programs to process each file.

Dummy aliasing

The sharing of memory locations between dummy (formal) arguments and other dummy arguments or COMMON variables that are assigned.

Dummy argument

A variable whose name appears in the parenthesized list following the procedure name in a FUNCTION statement, a SUBROUTINE statement, an ENTRY statement, or a statement function statement. A dummy argument takes the value of the corresponding actual argument in the calling program unit (through argument association). Also called a formal argument.

Dummy array

A dummy argument that is an array.

Dummy pointer

A dummy argument that is a pointer.

Dummy procedure

Is a dummy argument that is specified as a procedure or appears in a procedure reference. The corresponding actual argument must be a procedure.

Dynamic Link Library (DLL)

A separate source module compiled and linked independently of the applications that use it. Applications access the DLL through procedure calls. The code for a DLL is not included in the user's executable image, but the compiler automatically modifies the executable image to point to DLL procedures at run time.

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 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, operators, 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 90 file access modes are sequential and direct. On OpenVMS systems, you can also use a keyed mode of access.

File organization

The way records in a file are physically arranged on a storage device. Fortran 90 files can have sequential or relative organization. On OpenVMS systems, files can also have indexed organization.

Fixed-length record type

A file format in which all the records are the same length.

Focus window

Window to which keyboard input is directed.

Foreground window

The window with which the user is currently working. The system assigns a slightly higher priority to the thread that created the foreground window than it does to other threads.

Foreign file

An unformatted file that contains data from a foreign platform, such as data from a CRAY, IBM, or big endian IEEE machine.

Format

A specific arrangement of data. A FORMAT statement specifies how data is to be read or written.

Format specification

The part of a FORMAT statement that specifies explicit data arrangement. It is a list within parentheses that can include edit descriptors and field separators. A character expression can also specify format; the expression must evaluate to a valid format specification.

Formatted data

Data written to a file by using formatted I/O statements. Such data contains ASCII representations of binary values.

Formatted I/O statement

An I/O statement specifying a format for data transfer. The format specified can be explicit (specified in a format specification) or implicit (specified using list-directed or namelist formatting). *Contrast with [Unformatted I/O statement](#). See also [List-directed I/O statement](#) and [Namelist I/O statement](#).*

Function

A series of statements that perform some operation and return a single value (through the function or result name) to the calling program unit. A function is invoked by a function reference in a main program unit or a subprogram unit.

In Fortran 90, a function can be used to define a new operator or extend the meaning of an intrinsic operator symbol. The function is invoked by the appearance of the new or extended operator in the expression (along with the appropriate operands). For example, the symbol * can be defined for logical operands, extending its intrinsic definition for numeric operands. *See also [Function subprogram](#), [Statement function](#), and [Subroutine](#).*

Function reference

Used in an expression to invoke a function, it consists of the function name and its actual arguments. A function reference returns a value (through the function or result name) which is used to evaluate the calling expression.

Function result

The result value associated with a particular execution or call to a function. This result can be of any data type (including derived type) and can be array-valued. In a FUNCTION statement, the RESULT option can be used to give the result a name different from the function name. This

option is required for a recursive function that directly calls itself.

Function subprogram

A sequence of statements beginning with a FUNCTION (or optional OPTIONS) statement that is not in an interface block and ending with the corresponding END statement. *See also* [Function](#).

Glossary G**Generic identifier**

AA generic name, operator, or assignment specified in an INTERFACE statement that is associated with all of the procedures within the interface block. Also called a generic specification.

Global entity

An entity (a program unit, common block, or external procedure) that can be used with the same meaning throughout the executable program. A global entity has global scope; it is accessible throughout an executable program. *See also* [Local entity](#).

Global section

A data structure (for example, global COMMON) or shareable image section potentially available to all processes in the system.

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. DIGITAL Fortran 90 supports full High Performance Fortran (HPF), and compiles HPF programs for parallel execution.

Hollerith constant

A constant that is a string of printable ASCII characters preceded by 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.

Initialize

The assignment of an initial value to a variable.

Initialization expression

A form of constant expression that is used to specify an initial value for an entity.

Inlining

An optimization that replaces a subprogram reference (CALL statement or function invocation) with the replicated code of the subprogram.

Input/output (I/O)

The data that a program reads or writes. Also, devices to read and write data.

Inquiry function

An intrinsic function whose result depends on properties of the principal argument, not the value of the argument.

Integer constant

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

Intent

An attribute of a dummy argument that is not a procedure or a pointer. It indicates whether the argument is used to transfer data into the procedure, out of the procedure, or both.

Interface

The properties of a procedure, consisting of: specifications of the attributes for a function result, the specification of dummy argument attributes, and the information in the procedure heading.

Interface block

The sequence of statements starting with an INTERFACE statement and ending with the corresponding END INTERFACE statement.

Interface body

The sequence of statements in an interface block starting with a FUNCTION or SUBROUTINE statement and ending with the corresponding END statement. Also called a procedure interface body.

Internal file

The designated internal storage space (or variable buffer) that is manipulated during input and output. An internal file can be a character variable, character array, character array element, or

character substring. In general, an internal file contains one record. However, an internal file that is a character array has one record for each array element.

Internal procedure

A procedure (other than a statement function) that is contained within an internal subprogram. The program unit containing an internal procedure is called the host of the internal procedure. The internal procedure (which appears between a CONTAINS and END statement) is local to its host and inherits the host's environment through host association.

Internal subprogram

A subprogram contained in a main program or another subprogram.

Intrinsic

Describes entities defined by the Fortran 90 language (such as data types and procedures). Intrinsic entities can be used freely in any scoping unit.

Intrinsic procedure

A subprogram supplied as part of the Fortran 90 library that performs array, mathematical, numeric, character, bit manipulation, and other miscellaneous functions. Intrinsic procedures are automatically available to any Fortran 90 program unit (unless specifically overridden by an EXTERNAL statement or a procedure interface block). Also called a built-in or library procedure.

Invoke

To call upon; used especially with reference to subprograms. For example, to invoke a function is to execute the function.

Iteration count

The number of executions of the DO range, which is determined as follows:

$$[(\text{terminal value} - \text{initial value} + \text{increment value}) / \text{increment value}]$$

Glossary K

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 DIGITAL Fortran language element or interpretation that is not part of the Fortran 90 standard.

Lexical token

A sequence of one or more characters that have an indivisible interpretation. A lexical token is the smallest meaningful unit (a basic language element) of a Fortran 90 statement; for example, constants, and statement keywords.

Line

A source form record consisting of 0 or more characters. A standard Fortran 90 line is limited to a maximum of 132 characters.

Linker

A system program that creates an executable program from one or more object files produced by a language compiler or assembler. The linker resolves external references, acquires referenced library routines, and performs other processing required to create OpenVMS executable images or DIGITAL UNIX, Windows NT, and Windows 95 executable files.

List-directed I/O statement

An implicit, formatted I/O statement that uses an asterisk (*) specifier rather than an explicit format specification. *See also* [Formatted I/O statement](#) and [Namelist I/O statement](#).

Listing

A printed copy of a program.

Literal constant

A constant without a name. In Fortran 77, this was called simply a constant.

Little endian

A method of data storage in which the least significant bit of a numeric value spanning multiple bytes is in the lowest addressed byte. This is the method used on DIGITAL systems. *Contrast with* [Big endian](#).

Local entity

An entity that can be used only within the context of a subprogram (its scoping unit); for example, a statement label. A local entity has local scope. *See also* [Global entity](#).

Local optimization

Refers to enabling local optimizations within the source program unit, recognition of common expressions, and integer multiplication and division expansion (using shifts). The order of compilation of procedures is determined from the call graph. *See also* [Optimization](#).

Local symbol

A name defined in a program unit that is not accessible outside of that program unit.

Logical constant

A constant that specifies the value `.TRUE.` or `.FALSE.`.

Logical expression

An integer or logical constant, variable, function value, or another constant expression, joined by a relational or logical operator. The logical expression is evaluated to a value of either true or false. For example, `.NOT. 6.5 + (B .GT. D)`.

Logical operator

A symbol that represents an operation on logical expressions. The logical operators are `.AND.`, `.OR.`, `.NEQV.`, `.XOR.`, `.EQV.`, and `.NOT.`.

Logical unit

A channel in memory through which data transfer occurs between the program and the device or file. *See also* [Unit identifier](#).

Longword

Four contiguous bytes (32 bits) starting on any addressable byte boundary. Bits are numbered 0 to 31. The address of the longword is the address of the byte containing bit 0. When the longword is interpreted as a signed integer, bit 31 is the sign bit. The value of signed integers is in the range -2^{31} to $2^{31}-1$. The value of unsigned integers is in the range 0 to $2^{32}-1$.

Loop

A group of statements that are executed repeatedly until an ending condition is reached.

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 DIGITAL UNIX systems, an argument to the make command containing a sequence of entries that specify dependences. On Windows NT and Windows 95 systems, a file passed to the NMAKE utility containing a sequence of entries that specify dependences. The contents of a makefile override the system built-in rules for maintaining, updating, and regenerating groups of programs. For more information, see make(1) on DIGITAL UNIX systems, or see help on the NMAKE utility on Windows NT and Windows 95 systems.

Many-one array section

An array section with a vector subscript having two or more elements with the same value.

Metacommand

See [Compiler directive](#).

Misaligned data

Data not aligned on a natural boundary. See also [Natural boundary](#).

Module

A program unit that contains specifications and definitions that other program units can access (unless the module entities are declared PRIVATE). Modules are referenced in USE statements.

Module procedure

A subroutine or function defined within a module subprogram (the module procedure's host). The module procedure appears between a CONTAINS and END statement in its host module, and inherits the host module's environment through host association. A module procedure can be declared PRIVATE to the module; it is public by default.

Module subprogram

A subprogram that is contained in a module. (It cannot be an internal subprogram.)

Multibyte character set

A character set in which each character is identified by using more than one byte. Although Unicode characters are 2 bytes wide, the Unicode character set is not referred to by this term.

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(KIND=8) (REAL*8) data item aligned on natural boundaries has an address that is a multiple of eight.

Naturally aligned record

A record that is aligned on a hardware-specific natural boundary; each field is naturally aligned. (For more information, see the Programmer's Guide.) *Contrast with* [Packed record](#).

Nesting

The placing of one entity (such as a construct, subprogram, format specification, or loop) inside another entity of the same kind. For example, nesting a loop within another loop (a nested loop), or nesting a subroutine within another subroutine (a nested subroutine).

Nonexecutable statement

A Fortran 90 statement that describes program attributes, but does not cause any action to be taken when the program is executed.

Nonsignaled

The state of an object used for synchronization in one of the wait functions is either signaled or nonsignaled. A nonsignaled state can prevent the wait function from returning. See also [Wait function](#).

Numeric expression

A numeric constant, variable, or function value, or combination of these, joined by numeric operators and parentheses, so that the entire expression can be evaluated to produce a single numeric value. For example, -L or X+(Y-4.5*Z).

Numeric operator

A symbol designating an arithmetic operation. In Fortran 90, the symbols +, -, *, /, and ** are used to designate addition, subtraction, multiplication, division, and exponentiation, respectively.

Numeric storage unit

The unit of storage for holding a non-pointer scalar value of type default real, default integer, or default logical. One numeric storage unit corresponds to 4 bytes of memory.

Numeric type

Integer, real, or complex type.

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.

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 90 named constant.

Parent window

A window that has one or more child windows.

Pathname

On Windows NT, Windows 95, and DIGITAL UNIX systems, the path from the root directory to a subdirectory or file. *See also [Root](#).*

Pipe

A connection that allows one program to get its input directly from the output of another program

Platform

A combination of operating system and hardware that provides a distinct environment in which to use a software product (for example, Microsoft Windows 95 on Intel processors).

Pointer

Is one of the following:

- A Fortran 90 pointer
A data object that has the POINTER attribute. To be referenced or defined, it must be

"pointer-associated" with a target (have storage space associated with it). If the pointer is an array, it must be pointer-associated to have a shape. *See also* Pointer association.

- A DIGITAL Fortran 77 (or Integer) pointer
A data object that contains the address of its paired variable.

Pointer assignment

The association of a pointer with a target by the execution of a pointer assignment statement or the execution of an assignment statement for a data object of derived type having the pointer as a subobject.

Pointer association

The association of storage space to a Fortran 90 pointer by means of a target. A pointer is associated with a target after pointer assignment or the valid execution of an ALLOCATE statement.

Precision

The number of significant digits in a real number. *See also* [Double-precision constant](#), [Kind type parameter](#), and [Single-precision constant](#).

Primary

The simplest form of an expression. A primary can be any of the following data objects:

- A constant
- A constant subobject (parent is a constant)
- A variable (scalar, structure, array, or pointer; an array cannot be assumed size)
- An array constructor
- A structure constructor
- A function reference
- An expression in parentheses

Primary 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 and DIGITAL UNIX systems, it can also have quad precision (REAL(16)).

Record

Can be either of the following:

- A set of logically related data items (in a file) that is treated as a unit; such a record contains one or more fields. This definition applies to I/O records and items that are declared in a record structure.
- One or more data items that are grouped in a structure declaration and specified in a RECORD statement.

Record access

The method used to store and retrieve records in a file.

Record structure declaration

A block of statements that define the fields in a record. The block begins with a STRUCTURE statement and ends with END STRUCTURE. The name of the structure must be specified in a RECORD statement.

Record type

The property that determines whether records in a file are all the same length, of varying length, or use other conventions to define where one record ends and another begins.

Recursion

Pertains to a subroutine or function that directly or indirectly references itself.

Reference

Can be any of the following:

- For a data object, the appearance of its name, designator, or associated pointer where the value of the object is required. When an object is referenced, it must be defined.
- For a procedure, the appearance of its name, operator symbol, or assignment symbol that causes the procedure to be executed. Procedure reference is also called "calling" or "invoking" a procedure.
- For a module, the appearance of its name in a USE statement.

Relational expression

An expression containing one relational operator and two operands of numeric or character type. The result is a value that is true or false. For example, A-C .GE. B+2 or DAY .EQ. 'MONDAY'.

Relational operator

The symbols used to express a relational condition or expression. The relational operators are (.EQ., .NE., .LT., .LE., .GT., and .GE.).

Relative file organization

A file organization that consists of a series of component positions, called cells, numbered consecutively from 1 to n. DIGITAL Fortran 90 uses these numbered, fixed-length cells to calculate the component's physical position in the file.

Relative pathname

A directory path expressed in relation to any directory other than the root directory. *Contrast with [Absolute pathname](#).*

Root

On DIGITAL UNIX systems, the top-level directory in the file system; it is represented by aslash (/).

On Windows NT and Windows 95 systems, the top-level directory on a disk drive; it is represented by a backslash (\). For example, C:\ is the root directory for drive C.

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 = 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 your user manual. *See also* [Denormalized number](#).

Size

The total number of elements in an array (the product of the extents).

Source file

A program or portion of a program library, such as an object file, or image file.

Specification expression

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

Specification statement

A nonexecutable statement that provides information about the data used in the source program. Such a statement can be used to allocate and initialize variables, arrays, records, and structures, and define other characteristics of names used in a program.

Statement

An instruction in a programming language that represents a step in a sequence of actions or a set of declarations. In Fortran 90, an ampersand can be used to continue a statement from one line to another, and a semicolon can be used to separate several statements on one line.

There are two main classes of statements: executable and nonexecutable.

Statement entity

An entity identified by a lexical token whose scope is a single statement or part of a statement.

Statement function

A function whose definition is contained in a single statement.

Statement function definition

A statement that defines a statement function. Its form is the statement function name (followed by its optional dummy arguments in parentheses), followed by an equal sign (=), followed by a numeric, logical, or character expression.

A statement function definition must precede all executable statements and follow all specification statements.

Statement keyword

A word that begins the syntax of a statement. All program statements (except assignment statements and statement function definitions) begin with a statement keyword. Examples are INTEGER, DO, IF, and WRITE.

Statement label

See [Label](#).

Static variable

A variable whose storage is allocated for the entire execution of a program.

Storage association

The relationship between two storage sequences when the storage unit of one is the same as the storage unit of the other. Storage association is provided by the COMMON and EQUIVALENCE statements. For modules, pointers, allocatable arrays, and automatic data objects, the SEQUENCE statement defines a storage order for structures.

Storage location

An addressable unit of main memory.

Storage sequence

A sequence of any number of consecutive storage units. The size of a storage sequence is the number of storage units in the storage sequence. A sequence of storage sequences forms a

composite storage sequence. *See also* [Storage association](#) and [Storage unit](#).

Storage unit

In a storage sequence, the number of storage units needed to represent one real, integer, logical, or character value. *See also* [Character storage unit](#), [Numeric storage unit](#), and [Storage sequence](#).

Stride

The increment between subscript values, specified in a subscript triplet. If it is omitted, it is assumed to be one.

String edit descriptor

A format descriptor that transfers characters to an output record.

Structure

Can be either of the following:

- A scalar data object of derived (user-defined) type.
- An aggregate entity containing one or more fields or components.

Structure component

Can be either of the following:

- One of the components of a structure.
- An array whose elements are components of the elements of an array of derived type.

Structure constructor

A mechanism that is used to specify a scalar value of a derived type. A structure constructor is the name of the type followed by a parenthesized list of values for the components of the type.

Subobject

Part of a data object (parent object) that can be referenced and defined separately from other parts of the data object. A subobject can be an array element, an array section, a substring, a derived type, or a structure component. Subobjects are referenced by designators and can be considered to be data objects themselves. *See also* [Designator](#).

Subobject designator

See [Designator](#).

Subprogram

A user-written function or subroutine subprogram that can be invoked from another program unit to perform a specific task. Note that in FORTRAN 77, a block data program unit was also called a subprogram.

Subroutine

procedure that can return many values, a single value, or no value to the calling program unit (through arguments). A subroutine is invoked by a `CALL` statement in another program unit. In Fortran 90, a subroutine can also be used to define a new form of assignment (defined assignment), which is different from those intrinsic to Fortran 90. Such assignments are invoked with assignment syntax (using the `=` symbol) rather than the `CALL` statement. *See also* [Function](#), [Statement function](#), and [Subroutine subprogram](#).

Subroutine subprogram

A sequence of statements starting with a `SUBROUTINE` (or optional `OPTIONS`) statement and ending with the corresponding `END` statement. *See also* [Subroutine](#).

Subscript

A scalar integer expression (enclosed in parentheses and appended to the array name) indicating the position of an array element. The number of subscripts is the rank of the array. *See also* [Array element](#).

Subscript triplet

An item in a section subscript list specifying a range of values for the array section. A subscript triplet contains at least one colon and has three optional parts: a lower bound, an upper bound, and a stride. *Contrast with* [Vector subscript](#). *See also* [Array section](#) and [Section subscript](#).

Substring

A contiguous portion of a scalar character string. Do not confuse this with the substring selector in an array section, where the result is another array section, not a substring.

Symbolic name

See [Name](#).

Syntax

The formal structure of a statement or command string.

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.