Intel® Fortran Language Reference

Copyright © 2003 Intel Corporation

Document Number: 253261-001
World Wide Web: http://developer.intel.com

Information in this document is provided in connection with Intel products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, or life sustaining applications.

This Reference as well as the software described in it is furnished under license and may only be used or copied in accordance with the terms of the license. The information in this manual is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Intel Corporation. Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

The software described in this Reference may contain software defects which may cause the product to deviate from published specifications. Current characterized software defects are available on request.

Intel SpeedStep, Intel Thread Checker, Celeron, Dialogic, i386, i486, iCOMP, Intel, Intel logo, Intel386, Intel486, Intel740, IntelDX2, IntelDX4, IntelSX2, Intel Inside, Intel Inside logo, Intel NetBurst, Intel NetStructure, Intel Xeon, Intel XScale, Itanium, MMX, MMX logo, Pentium, Pentium II Xeon, Pentium III Xeon, Pentium M, and VTune are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

* Other names and brands may be claimed as the property of others.

Copyright © 2003 Intel Corporation.

Portions © Copyright 2001 Hewlett-Packard Development Company, L.P.

Contents

About This Manual

Product Website and Support xxvi
Related Publications..... xxvii
Conventions xxix
Platform Labels..... xxxi

Chapter 1 Overview

Language Standards Conformance 1-2
Language Compatibility..... 1-2
Fortran 2003 Features..... 1-2
Improved Features..... 1-3
Fortran 95 Features..... 1-3
New Features 1-3
Improved Features..... 1-4
Fortran 90 Features..... 1-5
New Features 1-5
Improved Features..... 1-7

Chapter 2 Program Structure, Characters, and Source Forms

Program Structure..... 2-1
Statements 2-2
Names 2-4
Character Sets 2-5
Source Forms..... 2-6
Free Source Form..... 2-9

Fixed and Tab Source Forms 2-11
Fixed-Format Lines 2-13
Tab-Format Lines 2-13
Source Code Useable for All Source Forms 2-15

Chapter 3 Data Types, Constants, and Variables

Intrinsic Data Types 3-2
Integer Data Types 3-4
Real Data Types 3-6
General Rules for Real Constants 3-7
REAL(4) Constants 3-8
REAL(8) or DOUBLE PRECISION Constants 3-9
REAL(16) Constants 3-10
Complex Data Types 3-10
General Rules for Complex Constants 3-11
COMPLEX(4) Constants 3-11
COMPLEX(8) or DOUBLE COMPLEX Constants 3-12
COMPLEX(16) Constants 3-13
Logical Data Types 3-14
Character Data Type 3-14
C Strings in Character Constants 3-16
Character Substrings 3-17
Derived Data Types 3-19
Derived-Type Definition 3-20
Default Initialization 3-22
Structure Components 3-23
Structure Constructors 3-26
Binary, Octal, Hexadecimal, and Hollerith Constants 3-28
Binary Constants 3-28
Octal Constants 3-29
Hexadecimal Constants 3-29
Hollerith Constants 3-30
Determining the Data Type of Nondecimal Constants 3-31
Variables 3-33

Data Types of Scalar Variables 3-34
Specification of Data Type 3-34
Implicit Typing Rules 3-35
Arrays 3-35
Whole Arrays 3-38
Array Elements 3-38
Array Sections 3-41
Array Constructors 3-44

Chapter 4 Expressions and Assignment Statements

Expressions 4-1
Numeric Expressions 4-2
Using Parentheses in Numeric Expressions 4-4
Data Type of Numeric Expressions 4-5
Character Expressions 4-6
Relational Expressions 4-7
Logical Expressions 4-8
Data Types Resulting from Logical Operations 4-9
Evaluation of Logical Expressions 4-9
Defined Operations 4-10
Summary of Operator Precedence 4-11
Initialization and Specification Expressions 4-11
Initialization Expressions 4-12
Specification Expressions 4-13
Assignment Statements 4-15
Intrinsic Assignments 4-16
Numeric Assignment Statements 4-17
Logical Assignment Statements 4-18
Character Assignment Statements 4-18
Derived-Type Assignment Statements 4-19
Array Assignment Statements 4-20
Defined Assignments 4-21
Pointer Assignments 4-22
WHERE Statement and Construct 4-23

FORALL Statement and Construct 4-26

Chapter 5 Specification Statements

Type Declaration Statements 5-2
Declaration Statements for Noncharacter Types 5-6
Declaration Statements for Character Types 5-8
Declaration Statements for Derived Types 5-10
Declaration Statements for Arrays 5-10
Explicit-Shape Specifications 5-11
Assumed-Shape Specifications 5-14
Assumed-Size Specifications 5-15
Deferred-Shape Specifications 5-16
ALLOCATABLE Attribute and Statement 5-17
AUTOMATIC and STATIC Attributes and Statements 5-18
COMMON Statement 5-21
DATA Statement 5-24
DIMENSION Attribute and Statement 5-27
EQUIVALENCE Statement 5-29
Making Arrays Equivalent 5-31
Making Substrings Equivalent 5-33
EQUIVALENCE and COMMON Interaction 5-35
EXTERNAL Attribute and Statement 5-38
IMPLICIT Statement 5-39
INTENT Attribute and Statement 5-41
INTRINSIC Attribute and Statement 5-43
NAMELIST Statement 5-45
OPTIONAL Attribute and Statement 5-46
PARAMETER Attribute and Statement 5-48
POINTER Attribute and Statement 5-50
PRIVATE and PUBLIC Attributes and Statements 5-51
SAVE Attribute and Statement 5-54
TARGET Attribute and Statement 5-55
VOLATILE Attribute and Statement 5-57

Chapter 6 Dynamic Allocation

ALLOCATE Statement 6-2
Allocation of Allocatable Arrays 6-3
Allocation of Pointer Targets 6-4
DEALLOCATE Statement 6-5
Deallocation of Allocatable Arrays 6-6
Deallocation of Pointer Targets 6-7
NULLIFY Statement 6-8

Chapter 7 Execution Control

Branch Statements 7-2
Unconditional GO TO Statement 7-2
Computed GO TO Statement 7-3
The ASSIGN and Assigned GO TO Statements 7-4
ASSIGN Statement 7-4
Assigned GO TO Statement 7-5
Arithmetic IF Statement 7-6
CALL Statement 7-7
CASE Constructs 7-9
CONTINUE Statement 7-14
DO Constructs 7-14
Forms for DO Constructs 7-15
Execution of DO Constructs 7-17
Iteration Loop Control 7-17
Nested DO Constructs 7-19
Extended Range 7-21
DO WHILE Statement 7-23
CYCLE Statement 7-24
EXIT Statement 7-24
END Statement 7-25
IF Construct and Statement 7-26
IF Construct 7-26
IF Statement 7-31
PAUSE Statement 7-32

RETURN Statement 7-33
STOP Statement 7-35

Chapter 8 Program Units and Procedures

Main Program 8-2
Modules and Module Procedures 8-4
Module References 8-7
USE Statement 8-8
Block Data Program Units 8-10
Functions, Subroutines, and Statement Functions 8-12
General Rules for Function and Subroutine Subprograms 8-13
Recursive Procedures 8-13
Pure Procedures 8-14
Elemental Procedures 8-17
Functions 8-18
RESULT Keyword 8-23
Function References 8-23
Subroutines 8-24
Statement Functions 8-27
External Procedures 8-28
Internal Procedures 8-29
Argument Association 8-30
Optional Arguments 8-32
Array Arguments 8-33
Pointer Arguments 8-34
Assumed-Length Character Arguments 8-35
Character Constant and Hollerith Arguments 8-36
Alternate Return Arguments 8-37
Dummy Procedure Arguments 8-37
References to Generic Procedures 8-38
References to Generic Intrinsic Functions 8-39
References to Elemental Intrinsic Procedures 8-42
References to Non-Fortran Procedures 8-43
%REF and %VAL Argument List Functions 8-43

%LOC Function 8-44
Procedure Interfaces 8-45
Determining When Procedures Require Explicit Interfaces 8-46
Defining Explicit Interfaces 8-46
Defining Generic Names for Procedures 8-49
Defining Generic Operators 8-50
Defining Generic Assignment 8-51
CONTAINS Statement 8-53
ENTRY Statement 8-53
ENTRY Statements in Function Subprograms 8-55
ENTRY Statements in Subroutine Subprograms 8-56

Chapter 9 Intrinsic Procedures

Argument Keywords in Intrinsic Procedures 9-3
Overview of Intrinsic Procedures 9-4
Categories of Intrinsic Functions 9-4
Intrinsic Subroutines 9-15
Bit Functions 9-16
Descriptions of Intrinsic Procedures 9-18
ABS 9-18
ACHAR 9-19
ACOS 9-20
ACOSD 9-20
ACOSH 9-21
ADJUSTL 9-21
ADJUSTR 9-22
AIMAG 9-22
AINT 9-23
ALL 9-23
ALLOCATED 9-24
ANINT 9-25
ANY 9-25
ASIN 9-26
ASIND 9-27

ASINH 9-27
ASSOCIATED 9-28
ATAN 9-29
ATAN2 9-30
ATAN2D 9-31
ATAND 9-31
ATANH 9-32
BADDRESS 9-32
BIT_SIZE 9-33
BTEST 9-33
CACHESIZE (i64 only) 9-34
CEILING 9-35
CHAR 9-35
CMPLX 9-36
CONJG 9-37
COS 9-38
COSD 9-38
COSH 9-39
COTAN 9-39
COTAND 9-40
COUNT 9-40
CPU_TIME 9-42
CSHIFT 9-42
DATE 9-44
DATE_AND_TIME 9-44
DBLE 9-46
DCMPLX 9-47
DFLOAT 9-48
DIGITS 9-48
DIM 9-49
DNUM 9-49
DOT_PRODUCT 9-50
DPROD 9-51
DREAL 9-51

DSHIFTL 9-52
DSHIFTR 9-52
EOF 9-53
EOSHIFT 9-54
EPSILON 9-56
ERF 9-57
ERFC 9-57
ERRSNS 9-58
EXIT 9-59
EXP 9-59
EXPONENT 9-60
FLOOR 9-60
FP_CLASS 9-61
FRACTION 9-61
FREE 9-62
GETARG 9-62
HUGE 9-64
IACHAR 9-64
IAND 9-65
IARGC 9-66
IARGPTR 9-66
IBCHNG 9-67
IBCLR 9-67
IBITS 9-68
IBSET 9-69
ICHAR 9-70
IDATE 9-71
IEOR 9-71
ILEN 9-72
INDEX 9-73
INT 9-73
INT_PTR_KIND 9-75
INUM 9-76
IOR 9-76

ISHA 9-77
ISHC 9-78
ISHFT 9-79
ISHFTC 9-80
ISHL 9-81
ISNAN 9-82
JNUM 9-82
KIND 9-82
LBOUND 9-83
LEADZ 9-84
LEN 9-85
LEN_TRIM 9-85
LGE 9-86
LGT 9-87
LLE 9-87
LLT 9-88
LOC 9-89
LOG 9-89
LOG10 9-90
LOGICAL 9-91
MALLOC 9-91
MATMUL 9-92
MAX 9-93
MAXEXPONENT 9-95
MAXLOC 9-95
MAXVAL 9-97
MCLOCK 9-98
MERGE 9-98
MIN 9-99
MINEXPONENT 9-100
MINLOC 9-100
MINVAL 9-102
MM_PREFETCH 9-103
MOD 9-105

MODULO 9-106
MULT_HIGH (i64 only) 9-106
MVBITS 9-107
NARGS 9-108
NEAREST 9-110
NINT 9-110
NOT 9-111
NULL 9-112
PACK 9-113
POPCNT 9-114
POPPAR 9-114
PRECISION 9-114
PRESENT 9-115
PRODUCT 9-116
QCMPLX 9-117
QEXT 9-117
QFLOAT 9-118
QNUM 9-119
QREAL 9-119
RADIX 9-120
RAN 9-120
RANDOM_NUMBER 9-121
RANDOM_SEED 9-122
RANDU 9-124
RANGE 9-124
REAL 9-125
REPEAT 9-126
RESHAPE 9-127
RNUM 9-128
RRSPACING 9-128
SCALE 9-128
SCAN 9-129
SECNDS 9-130
SELECTED_INT_KIND 9-130

SELECTED_REAL_KIND 9-131
SET_EXPONENT 9-132
SHAPE 9-132
SHIFTL 9-133
SHIFTR 9-133
SIGN 9-134
SIN 9-135
SIND 9-135
SINH 9-136
SIZE 9-136
SIZEOF 9-137
SPACING 9-138
SPREAD 9-138
SQRT 9-139
SUM 9-140
SYSTEM_CLOCK 9-141
TAN 9-142
TAND 9-143
TANH 9-143
TIME 9-144
TINY 9-144
TRAILZ 9-145
TRANSFER 9-145
TRANSPOSE 9-146
TRIM 9-147
UBOUND 9-147
UNPACK 9-148
VERIFY 9-149
ZEXT 9-150

Chapter 10 Data Transfer I/O Statements

Records and Files 10-1
Components of Data Transfer Statements 10-2
I/O Control List 10-3

Unit Specifier 10-4
Format Specifier 10-5
Namelist Specifier 10-6
Record Specifier 10-6
I/O Status Specifier 10-6
Branch Specifiers 10-7
Advance Specifier 10-8
Character Count Specifier 10-9
I/O Lists 10-9
Simple List Items in I/O Lists 10-10
implied-DO Lists in I/O Lists 10-12
READ Statements 10-13
Forms for Sequential READ Statements 10-13
Rules for Formatted Sequential READ Statements 10-15
Rules for List-Directed Sequential READ Statements 10-15
Rules for Namelist Sequential READ Statements 10-18
Rules for Unformatted Sequential READ Statements 10-23
Forms for Direct-Access READ Statements 10-24
Rules for Formatted Direct-Access READ Statements 10-25
Rules for Unformatted Direct-Access READ Statements 10-26
Forms and Rules for Internal READ Statements 10-26
ACCEPT Statement 10-28
WRITE Statements 10-29
Forms for Sequential WRITE Statements 10-29
Rules for Formatted Sequential WRITE Statements 10-30
Rules for List-Directed Sequential WRITE Statements 10-31
Rules for Namelist Sequential WRITE Statements 10-33
Rules for Unformatted Sequential WRITE Statements 10-34
Forms for Direct-Access WRITE Statements 10-35
Rules for Formatted Direct-Access WRITE Statements 10-36
Rules for Unformatted Direct-Access WRITE Statements 10-36
Forms and Rules for Internal WRITE Statements 10-36
PRINT and TYPE Statements 10-38
REWRITE Statement 10-39

Chapter 11 I/O Formatting

Format Specifications 11-2
Data Edit Descriptors 11-6
Forms for Data Edit Descriptors 11-6
General Rules for Numeric Editing 11-8
Integer Editing 11-9
I Editing 11-9
B Editing 11-11
O Editing 11-12
Z Editing 11-13
Real and Complex Editing 11-14
F Editing 11-15
E and D Editing 11-16
EN Editing 11-19
ES Editing 11-20
G Editing 11-22
Complex Editing 11-24
Logical Editing (L) 11-25
Character Editing (A) 11-26
Default Widths for Data Edit Descriptors 11-28
Terminating Short Fields of Input Data 11-29
Control Edit Descriptors 11-30
Forms for Control Edit Descriptors 11-30
Positional Editing 11-31
T Editing 11-31
TL Editing 11-32
TR Editing 11-32
X Editing 11-32
Sign Editing 11-33
SP Editing 11-33
SS Editing 11-33
S Editing 11-33
Blank Editing 11-33
BN Editing 11-34

BZ Editing 11-34
Scale Factor Editing (P) 11-34
Slash Editing (/) 11-36
Colon Editing (:) 11-37
Dollar Sign ($) and Backslash (\) Editing 11-37
Character Count Editing (Q) 11-38
Character String Edit Descriptors 11-38
Character Constant Editing 11-39
H Editing 11-39
Nested and Group Repeat Specifications 11-40
Variable Format Expressions 11-41
Printing of Formatted Records 11-42
Interaction Between Format Specifications and I/O Lists 11-43

Chapter 12 File Operation I/O Statements

BACKSPACE Statement 12-2
CLOSE Statement 12-3
DELETE Statement 12-4
ENDFILE Statement 12-5
INQUIRE Statement 12-7
ACCESS Specifier 12-8
ACTION Specifier 12-9
BINARY Specifier (W*32, W*64) 12-9
BLANK Specifier 12-9
BLOCKSIZE Specifier 12-10
BUFFERED Specifier 12-10
CARRIAGECONTROL Specifier 12-10
CONVERT Specifier 12-11
DELIM Specifier 12-12
DIRECT Specifier 12-12
EXIST Specifier 12-12
FORM Specifier 12-13
FORMATTED Specifier 12-13
IOFOCUS Specifier (W*32, W*64) 12-13

MODE Specifier 12-14
NAME Specifier 12-14
NAMED Specifier 12-14
NEXTREC Specifier 12-15
NUMBER Specifier 12-15
OPENED Specifier 12-15
ORGANIZATION Specifier 12-16
PAD Specifier 12-16
POSITION Specifier 12-16
READ Specifier 12-17
READWRITE Specifier 12-17
RECL Specifier 12-17
RECORDTYPE Specifier 12-18
SEQUENTIAL Specifier 12-18
SHARE Specifier (W*32, W*64) 12-19
UNFORMATTED Specifier 12-19
WRITE Specifier 12-19
OPEN Statement 12-20
ACCESS Specifier 12-24
ACTION Specifier 12-25
ASSOCIATEVARIABLE Specifier 12-25
BLANK Specifier 12-26
BLOCKSIZE Specifier 12-26
BUFFERCOUNT Specifier 12-26
BUFFERED Specifier 12-27
CARRIAGECONTROL Specifier 12-28
CONVERT Specifier 12-28
DEFAULTFILE Specifier 12-30
DELIM Specifier 12-30
DISPOSE Specifier 12-31
FILE Specifier 12-32
FORM Specifier 12-32
IOFOCUS Specifier (W*32, W*64) 12-33
MAXREC Specifier 12-33

MODE Specifier 12-33
NAME Specifier 12-33
ORGANIZATION Specifier 12-34
PAD Specifier 12-34
POSITION Specifier 12-35
READONLY Specifier 12-35
RECL Specifier 12-36
RECORDSIZE Specifier 12-37
RECORDTYPE Specifier 12-37
SHARE Specifier (W*32, W*64) 12-38
SHARED Specifier 12-39
STATUS Specifier 12-39
TITLE Specifier (W*32, W*64) 12-40
TYPE Specifier 12-40
USEROPEN Specifier 12-40
REWIND Statement 12-41
UNLOCK Statement 12-42

Chapter 13 Compilation Control Statements

INCLUDE Statement 13-1
OPTIONS Statement 13-3

Chapter 14 Directive Enhanced Compilation

Syntax Rules for Compiler Directives 14-1
General Compiler Directives 14-2
Rules for General Directives that Affect DO Loops 14-4
ALIAS Directive 14-5
ATTRIBUTES Directive 14-5
ATTRIBUTES ALIAS 14-8
ATTRIBUTES ALIGN 14-9
ATTRIBUTES ALLOCATABLE 14-9
ATTRIBUTES ALLOW_NULL 14-9
ATTRIBUTES ARRAY_VISUALIZER (W*32 only) 14-10
ATTRIBUTES C and STDCALL 14-10

ATTRIBUTES DECORATE 14-12
ATTRIBUTES DEFAULT 14-12
ATTRIBUTES DLLEXPORT and DLLIMPORT (W*32, W*64) 14-13
ATTRIBUTES EXTERN 14-13
ATTRIBUTES IGNORE_LOC 14-13
ATTRIBUTES INLINE, NOINLINE, and FORCEDINLINE 14-14
ATTRIBUTES NO_ARG_CHECK 14-14
ATTRIBUTES NOMIXED_STR_LEN_ARG 14-15
ATTRIBUTES REFERENCE and VALUE 14-15
ATTRIBUTES VARYING 14-16
DECLARE and NODECLARE Directives 14-16
DEFINE and UNDEFINE Directives 14-17
DISTRIBUTE POINT Directive 14-18
FIXEDFORMLINESIZE Directive 14-19
FREEFORM and NOFREEFORM Directives 14-20
IDENT Directive 14-20
IF and IF DEFINED Directives 14-20
INTEGER Directive 14-22
IVDEP Directive 14-23
LOOP COUNT Directive 14-25
MESSAGE Directive 14-26
OBJCOMMENT Directive 14-26
OPTIONS Directive 14-27
PACK Directive 14-30
PARALLEL and NOPARALLEL Directives 14-31
PREFETCH and NOPREFETCH Directives 14-32
PSECT Directive 14-33
REAL Directive 14-35
STRICT and NOSTRICT Directives 14-36
SWP and NOSWP Directives (i64 only) 14-37
TITLE and SUBTITLE Directives 14-38
UNROLL and NOUNROLL Directives 14-39
VECTOR ALIGNED and VECTOR UNALIGNED Directives (i32 only) 14-39

VECTOR ALWAYS and NOVECTOR Directives (i32 only)..... 14-40
VECTOR NONTEMPORAL Directive (i32 only)..... 14-41
OpenMP* Fortran Compiler Directives..... 14-42
Data Scope Attribute Clauses..... 14-44
COPYIN Clause 14-44
COPYPRIVATE Clause 14-44
DEFAULT Clause..... 14-45
FIRSTPRIVATE Clause 14-46
LASTPRIVATE Clause..... 14-46
PRIVATE Clause..... 14-46
REDUCTION Clause..... 14-47
SHARED Clause 14-49
Conditional Compilation Rules..... 14-49
Nesting and Binding Rules..... 14-50
ATOMIC Directive..... 14-52
BARRIER Directive 14-53
CRITICAL Directive..... 14-54
DO Directive..... 14-55
FLUSH Directive 14-59
MASTER Directive 14-60
ORDERED Directive 14-61
PARALLEL Directive 14-62
PARALLEL DO Directive 14-64
PARALLEL SECTIONS Directive 14-66
SECTIONS Directive..... 14-67
SINGLE Directive..... 14-68
THREADPRIVATE Directive..... 14-69

Chapter 15 Scope and Association

Scope..... 15-1
Unambiguous Generic Procedure References 15-4
Resolving Procedure References 15-5
References to Generic Names..... 15-5
References to Specific Names..... 15-7

References to Nonestablished Names 15-8
Association 15-9
Name Association 15-10
Argument Association 15-10
Use and Host Association 15-11
Pointer Association 15-12
Storage Association 15-13
Storage Units and Storage Sequence 15-13
Array Association 15-15

Appendix A Deleted and Obsolescent Language Features

Deleted Language Features in Fortran 95 ..... A-1
Obsolescent Language Features in Fortran 95 ..... A-2
Obsolescent Language Features in Fortran 90 ..... A-3

Appendix B Additional Language Features

DEFINE FILE Statement B-1
ENCODE and DECODE Statements B-3
FIND Statement B-5
INTERFACE TO Statement B-5
FORTRAN-66 Interpretation of the EXTERNAL Statement B-6
Alternative Syntax for the PARAMETER Statement B-8
VIRTUAL Statement B-9
Alternative Syntax for Octal and Hexadecimal Constants B-10
Alternative Syntax for a Record Specifier B-10
Alternative Syntax for the DELETE Statement B-10
Alternative Form for Namelist External Records B-11
Integer POINTER Statement B-12
Record Structures B-13
Structure Declarations B-14
Type Declarations B-18
Substructure Declarations B-18
Union Declarations B-19
RECORD Statement B-21

References to Record Fields B-22
Aggregate Assignment B-24

Appendix C The ASCII Character Set for Linux Systems
The ASCII Character Set (L*X) C-1

Appendix D Data Representation Models
Model for Integer Data D-2
Model for Real Data D-3
Model for Bit Data D-4

Appendix E Run-Time Library Routines
Module Routines E-1
Portability Routines E-2
National Language Support Routines (W*32, W*64) E-8
POSIX* Routines E-10
QuickWin Routines (W*32, W*64) E-14
Graphics Routines (W*32, W*64) E-16
Dialog Routines (W*32) E-20
Miscellaneous Run-Time Routines E-22
COM Routines (W*32) E-23
AUTO Routines (W*32) E-24
OpenMP* Fortran Routines E-25

Appendix F Summary of Language Extensions
Source Forms F-1
Names F-1
Character Sets F-1
Intrinsic Data Types F-2
Constants F-2
Expressions and Assignment F-2
Specification Statements F-2
Execution Control F-3
Compilation Control Statements F-3
Built-In Functions F-3

I/O Statements..... F-3
I/O Formatting..... F-3
File Operation Statements..... F-4
Compiler Directives F-5
Intrinsic Procedures..... F-7
Additional Language Features..... F-9
Run-Time Library Routines..... F-10

Glossary

Index

About This Manual

This manual contains the complete description of the *Intel® Fortran* programming language, which includes Fortran 95, Fortran 90, and some Fortran 2000 language features. It contains information on language syntax and semantics, on adherence to various Fortran standards, and on extensions to those standards.

It applies to the following:

- Intel Fortran for Linux* on IA-32 systems
- Intel Fortran for Linux on Intel® Itanium® systems
- Intel Visual Fortran on IA-32 and Intel Itanium systems

For details on the features of the compilers and how to improve the run-time performance of Fortran programs, see your user's guide.

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

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

This manual is organized as follows:

- Chapter 1, "Overview," describes language standards, language compatibility, and Fortran 95/90 features.
- Chapter 2, "Program Structure, Characters, and Source Forms," describes program structure, the Fortran 95/90 character set, and source forms.
- Chapter 3, "Data Types, Constants, and Variables," describes intrinsic and derived data types, constants, variables (scalars and arrays), and substrings.
- Chapter 4, "Expressions and Assignment Statements," describes Fortran expressions and assignment statements, which are used to define or redefine variables.
- Chapter 5, "Specification Statements," describes specification statements, which are used to declare the attributes of data objects.

- Chapter 6, “Dynamic Allocation,” describes statements used in dynamic allocation.
- Chapter 7, “Execution Control,” describes constructs and statements that can transfer control within a program.
- Chapter 8, “Program Units and Procedures,” describes program units (including modules), subroutines and functions, and procedure interfaces.
- Chapter 9, “Intrinsic Procedures,” summarizes all intrinsic procedures.
- Chapter 10, “Data Transfer I/O Statements,”describes data transfer input/output (I/O) statements.
- Chapter 11, “I/O Formatting,” describes the rules for I/O formatting.
- Chapter 12, “File Operation I/O Statements,” describes auxiliary I/O statements you can use to perform file operations.
- Chapter 13, “Compilation Control Statements,” describes compilation control statements.
- Chapter 14, “Directive Enhanced Compilation,” describes general and parallel compiler directives.
- Chapter 15, “Scope and Association,” describes scope and association.
- Appendix A, “Deleted and Obsolescent Language Features,” describes deleted features in Fortran 95 and obsolescent language features in Fortran 95 and Fortran 90.
- Appendix B, “Additional Language Features,” describes some statements and language features supported for programs written in older versions of Fortran.
- Appendix C, “The ASCII Character Set for Linux Systems,” describes the ASCII character set available on Linux* systems. For information on character sets available on Windows* systems, see the online documentation for those systems.
- Appendix D, “Data Representation Models,” describes data representation models for numeric intrinsic functions.
- Appendix E, “Run-Time Library Routines,” summarizes the many run-time library routines.
- Appendix F, “Summary of Language Extensions,” summarizes Intel Fortran extensions to the Fortran 95 Standard.
- The Glossary contains abbreviated definitions of some commonly used terms in this manual.

Product Website and Support

Intel® Fortran provides a product web site that offers timely and comprehensive product information, including product features, white papers, and technical articles. For the latest information, visit:

http://developer.intel.com/software/products/

Intel also provides a support web site that contains a rich repository of self help information, including getting started tips, known product issues, product errata, license information, user forums, and more.

Registering your product entitles you to one year of technical support and product updates through Intel® Premier Support. Intel Premier Support is an interactive issue management and communication web site providing these services:

- Submit issues and review their status.
- Download product updates anytime of the day.

To register your product, contact Intel, or seek product support, please visit:

http://www.intel.com/software/products/support

Related Publications

The following is an alphabetical list of some commercially published documents that provide reference or tutorial information on Fortran 95 and Fortran 90:

- Compaq Visual Fortran by N. Lawrence; published by Digital Press* (Butterworth-Heinemann), ISBN: 1-55558-249-4.
- Digital Visual Fortran Programmer's Guide by M. Etzel and K. Dickinson; published by Digital Press* (Butterworth-Heinemann), ISBN: 1-55558-218-4.
- *Fortran 90 Explained* by M. Metcalf and J. Reid; published by Oxford University Press, ISBN 0-19-853772-7.
- *Fortran 90/95 Explained* by M. Metcalf and J. Reid; published by Oxford University Press, ISBN 0-19-851888-9.
- *Fortran 90/95 for Scientists and Engineers* by S. Chapman; published by McGraw-Hill, ISBN 0-07-011938-4.
- *Fortran 90 Handbook* by J. Adams, W. Brainerd, J. Martin, B. Smith, and J. Wagener; published by Intertext Publications (McGraw-Hill), ISBN 0-07-000406-4.
- *Fortran 90 Programming* by T. Ellis, I. Philips, and T. Lahey; published by Addison-Wesley, ISBN 0201-54446-6.
- *Introduction to Fortran 90/95* by Stephen J. Chapman; published by WCB McGraw-Hill, ISBN 0-07-011969-4.
- *Programmer's Guide to Fortran 90, Second Edition* by W. Brainerd, C. Goldberg, and J. Adams; published by Unicomp, ISBN 0-07-000248-7.

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

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

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

  This Standard is equivalent to: International Standards Organization Programming Language Fortran, ISO/IEC 1539:1991 (E).
- American National Standard Programming Language Fortran 95, ANSI X3J3/96-007

  This Standard is equivalent to: International Standards Organization Programming Language Fortran, ISO/IEC 1539-1:1997 (E).
- High Performance Fortran Language Specification, Version 1.1, Technical Report CRPC-TR-92225
- OpenMP Fortran Application Program Interface, Version 1.1, November 1999
- OpenMP Fortran Application Program Interface, Version 2.0, November 2000

Information about the target architecture is available from Intel and from most technical bookstores. Most Intel documents are available from the Intel Corporation web site at:

http://www.intel.com

Some helpful titles are:

- *Intel® Fortran Language Reference*
- *Intel® Fortran Libraries Reference*
- *Intel® Fortran Compiler Installing and Getting Started*
- Intel® Array Visualizer online help reference
- Intel® Array Viewer online help reference
- *Using the Intel® License Manager for FLEXlm**
- *Intel® C++ Compiler User's Guide*
- VTune™ Performance Analyzer online help
- Enhanced Debugger online help
- *Intel® Architecture Software Developer's Manual*
  - Vol. 1: Basic Architecture, Intel Corporation, doc. number 243190
  - Vol. 2: Instruction Set Reference Manual, Intel Corporation, doc. number 243191
  - Vol. 3: System Programming, Intel Corporation, doc. number 243192
- *Pentium® Processor Family Developer's Manual*
- *Intel® Processor Identification with the CPUID Instruction*, Intel Corporation, doc. number 241618
- *Intel® Itanium® Architecture* Manuals
- *Intel® Itanium® Architecture Software Conventions & Runtime Architecture Guide*
- *Intel® Itanium® Assembler User's Guide*
- *Intel® Itanium® Architecture Assembly Language Reference Guide*

For more developer's manuals on Intel processors, refer to the Intel's Literature Center.

The following sources might be useful in helping you understand basic optimization and vectorization terminology and technology:

- *Intel® Architecture Optimization Reference Manual*
- *Dependence Analysis*, Utpal Banerjee (A Book Series on Loop Transformations for Restructuring Compilers). Kluwer Academic Publishers. 1997.
- *The Structure of Computers and Computation: Volume I*, David J. Kuck. John Wiley and Sons, New York, 1978.
- *Loop Transformations for Restructuring Compilers: The Foundations*, Utpal Banerjee (A Book Series on Loop Transformations for Restructuring Compilers). Kluwer Academic Publishers. 1993.
- *Loop parallelization*, Utpal Banerjee (A Book Series on Loop Transformations for Restructuring Compilers). Kluwer Academic Publishers. 1994.
- *High Performance Compilers for Parallel Computers*, Michael J. Wolfe. Addison-Wesley, Redwood City. 1996.
- *Supercompilers for Parallel and Vector Computers*, H. Zima. ACM Press, New York, 1990.
- *An Auto-vectorizing Compiler for the Intel® Architecture*, Aart Bik, Paul Grey, Milind Girkar, and Xinmin Tian. Submitted for publication
- *Efficient Exploitation of Parallelism on Pentium® III and Pentium® 4 Processor-Based Systems*, Aart Bik, Milind Girkar, Paul Grey, and Xinmin Tian.

Conventions

The following table describes the typographic and terminology conventions used in this manual:

Typographic Conventions	
Extensions to Fortran 95	This color indicates extensions to the Fortran 95 Standard. These extensions may or may not be implemented by other compilers that conform to the language standard.
AUTOMATIC, INTRINSIC, WRITE	Uppercase letters indicate Fortran95/90 statements, data types, directives, and other syntax keywords. Examples of statement keywords are WRITE, INTEGER, DO, and OPEN.
option, *option*	This italic type indicates an keyword arguments in syntax, new terms, emphasized text, or a book title. Most new terms are defined in the Glossary of the *Language Reference*.
`CALL CPU_TIME`	This courier type indicates a code example, a derived type name, or a pathname.

CTRL	Small capital letters indicate the names of keys and key sequences, such as CTRL+C. A plus indicates a combination of keys. For example, CTRL+E means to hold down the CTRL key while pressing the E key.
{*choice1* | *choice2*}	Braces and vertical bars indicate a choice of items. You can usually only choose one of the items in the braces.
[optional item]	In syntax, single square brackets indicate items that are optional. In code examples, they are used to show arrays.
s[, s]…	A horizontal ellipsis (three dots in a row) following an item indicates that the item preceding the ellipsis can be repeated. In code examples, a horizontal ellipsis means that not all of the statements are shown.
Adobe Acrobat*	An asterisk at the end of a word or name indicates it is a third-party product trademark.

Terminology Conventions

compiler option	This term refers to Linux* options and Windows* options that can be used on the compiler command line.
`cat(1)`	This format refers to an online reference page; the section number of the page is shown in parentheses. For example, a reference to `cat(1)` indicates that you can find the material on the `cat` command in Section 1 of the reference pages. To read online reference pages, use the `man` command. Your operating system documentation also includes reference page descriptions.
Intel Fortran	This term refers to the name of the common compiler language supported by the Intel® Visual Fortran Compiler for Windows* and Intel® Fortran Compiler for Linux* products. For more information on these compilers, see http://developer.intel.com/software/products/.
Fortran	This term refers to language information that is common to ANSI FORTRAN 77, ANSI/ISO Fortran 95/90, and Intel Fortran.
Fortran 95/90	This term refers to language information that is common to ANSI/ISO Fortran 95 and ANSI/ISO Fortran 90.
Fortran 95	This term refers to language features of ANSI/ISO Fortran 95.
Fortran 90	This term refers to language features of ANSI/ISO Fortran 90.
Windows systems	This term refers to all supported Microsoft* Windows operating systems. (See also "Platform Labels".)
Linux systems	This term refers to all supported Linux operating systems. (See also "Platform Labels".)
integer	This term refers to the INTEGER(KIND=1), INTEGER(KIND=2), INTEGER (INTEGER(KIND=4)), and INTEGER(KIND=8) data types as a group.

real	This term refers to the REAL (REAL(KIND=4)), DOUBLE PRECISION (REAL(KIND=8)), and REAL(KIND=16) data types as a group.
REAL	This term refers to the default data type of objects declared to be REAL. REAL is equivalent to REAL(KIND=4), unless a compiler option specifies otherwise.
complex	This term refers to the COMPLEX (COMPLEX(KIND=4)), DOUBLE COMPLEX (COMPLEX(KIND=8)), and COMPLEX(KIND=16) data types as a group.
logical	This term refers to the LOGICAL(KIND=1), LOGICAL(KIND=2), LOGICAL (LOGICAL(KIND=4)), and LOGICAL(KIND=8) data types as a group.
<Tab>	This symbol indicates a nonprinting tab character.
Δ	This symbol indicates a nonprinting blank character.

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

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

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

- The keyword PARAMETER.
- An optional left parenthesis.
- One or more *c* = *expr* items, where *c* is a named constant and *expr* is a value.

  If you want to specify more than one *c* = *expr* item, a comma must separate the items.

  The three dots following the syntax mean you can enter as many of these sequences (a comma, followed by c = expr) as you like.
- An optional terminating right parenthesis. If you used the optional left parenthesis, you must use the terminating right parenthesis.

The colored brackets ([ ]) indicate that the parentheses are optional only as an extension to standard Fortran 95.

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). This manual contains information for the following language platforms:

	Platform[1]	
Language	**Operating System**	**CPU**
Intel® Fortran	Linux	IA-32

	Platform[1]	
Language	**Operating System**	**CPU**
	Linux	Intel® Itanium®
	Microsoft* Windows* 2000	IA-32
	Microsoft Windows NT* 4.0	IA-32
	Microsoft Windows XP*	IA-32
	Microsoft Windows XP	Intel Itanium

1. For the latest information on the current language platforms, see the online *Release Notes*.

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

L*X	Applies to Linux* on Intel® IA-32 processors and Intel® Itanium® processors.
L*X32	Applies to Linux on Intel IA-32 processors.
L*X64	Applies to Linux on Intel Itanium processors.
W*32	Applies to Microsoft Windows* 2000, Windows XP, and Windows NT* 4.0 on Intel IA-32 processors.
W*64	Applies to Microsoft Windows XP operating systems on Intel Itanium processors.
i32	Applies to 32-bit operating systems on Intel IA-32 processors.
i64	Applies to 64-bit operating systems on Intel Itanium processors.

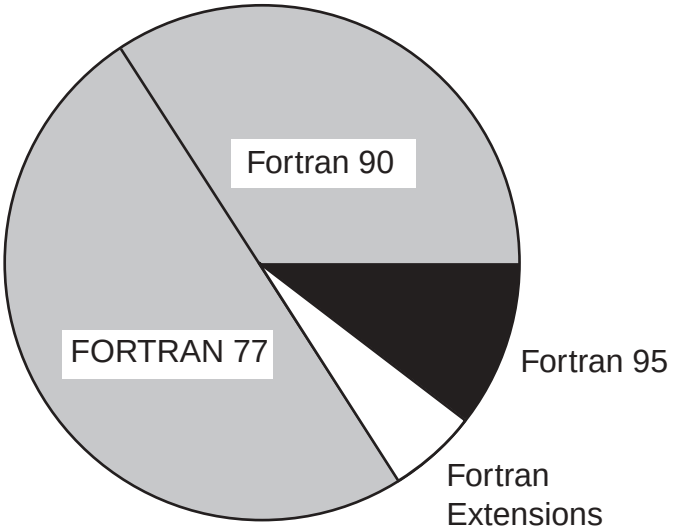
For example, the IOFOCUS specifier (for an OPEN statement) is labeled "(W*32, W*64)", so this specifier is valid only on Windows operating systems.

Overview

1

This chapter discusses Intel® Fortran standards conformance and language compatibility, and provides an overview of Fortran 95, Fortran 90, and proposed Fortran 2003 features.

Figure 1-1 Graphic Representation of Intel Fortran



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

Language Standards Conformance

Intel Fortran conforms to American National Standard Fortran 95 (ANSI X3J3/96-007)[1], American National Standard Fortran 90 (ANSI X3.198-1992)[2], and includes support for some features in proposed standard Fortran 2003.

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

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

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

See Also

Appendix F, “Summary of Language Extensions”, for a summary of Intel Fortran language extensions

Language Compatibility

Intel Fortran is highly compatible with Compaq* Fortran and Compaq Fortran 77 on supported platforms, and it is substantially compatible with PDP-11* and VAX* FORTRAN 77.

See Also

Your user’s guide for specific details on language compatibility, compiler options, and program conversion considerations

Fortran 2003 Features

This section briefly describes the Fortran 2003 features that have been implemented in Intel® Fortran.

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

Improved Features

The following Fortran 2003 features improve previous Fortran features:

- Enhancement to derived-type components, function results, and dummy arguments

  Components of derived types can now be allocatable and function results and dummy arguments can now be allocatable.

  For more information, see "Derived-Type Definition", "Functions", and "Array Arguments".

Fortran 95 Features

This section briefly describes the Fortran 95 language features that have been implemented in Intel Fortran. Some features are new, while others are improvements to previous Fortran features.

New Features

The following Fortran 95 features are new to Fortran:

- The FORALL statement and construct

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

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

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

  For more information, see "FORALL Statement and Construct".

- PURE user-defined procedures

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

  For more information, see "Pure Procedures".

- ELEMENTAL user-defined procedures

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

  For more information, see "Functions" and "Subroutines".

- CPU_TIME intrinsic subroutine

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

  For more information, see "CPU_TIME".
- NULL intrinsic function

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

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

  For more information, see "NULL".
- New obsolescent features

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

  Intel Fortran fully supports features deleted in Fortran 95.

  For more information, see Appendix A, "Deleted and Obsolescent Language Features".

Improved Features

The following Fortran 95 features improve previous Fortran features:

- Derived-type structure default initialization

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

  For more information, see "Default Initialization".
- Pointer initialization

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

  For more information, see "Derived-Type Definition" and "Default Initialization".
- Automatic deallocation of allocatable arrays

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

  For more information, see "Allocation of Allocatable Arrays".
- Enhanced CEILING and FLOOR intrinsic functions

  KIND can now be specified for these intrinsic functions.

  For more information, see "CEILING" and "FLOOR".
- Enhanced MAXLOC and MINLOC intrinsic functions

  DIM can now be specified for these intrinsic functions.

  For more information, see "MAXLOC" and "MINLOC".

- Enhanced SIGN intrinsic function

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

  For more information, see "SIGN".
- Printing of –0.0

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

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

  For more information, see "WHERE Statement and Construct".
- 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.

  For more information, see "Defining Explicit Interfaces".
- 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.

Fortran 90 Features

This section briefly describes the Fortran 90 language features that have been implemented in Intel Fortran. Some features are new, while others are improvements to previous Fortran features.

New Features

The following Fortran 90 features are new to Fortran:

- Free source form

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

  For more information, see "Free Source Form".

- Modules

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

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

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

  For more information, see "Modules and Module Procedures".

- User-defined (derived) data types and operators

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

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

  For more information, see "Derived Data Types" and "Defining Generic Operators".

- Array operations and features

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

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

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

  For more information, see "Arrays" and Chapter 9, "Intrinsic Procedures".

- Generic user-defined procedures

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

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

  For more information, see "Defining Generic Names for Procedures".

- Pointers

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

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

  For more information, see "POINTER Attribute and Statement" and "Pointer Assignments".
- Recursion

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

  For more information, see Chapter 8, "Program Units and Procedures".
- Interface blocks

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

  — Describe the characteristics of an external or dummy procedure

  — Define a generic name for a procedure

  — Define a new operator (or extend an intrinsic operator)

  — Define a new form of assignment

  For more information, see "Procedure Interfaces".
- Extensibility and redundancy

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

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

  For more information, see your user's guide. For a list of obsolescent features, see Appendix A, "Deleted and Obsolescent Language Features".

Improved Features

The following Fortran 90 features improve previous Fortran features:

- Additional features for source text

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

  For more information, see Chapter 2, "Program Structure, Characters, and Source Forms".
- Improved facilities for numerical computation

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

  For more information, see Chapter 3, "Data Types, Constants, and Variables" and Chapter 9, "Intrinsic Procedures".
- Additional input/output features

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

  For more information on formatting, see Chapter 10, "Data Transfer I/O Statements"; on OPEN and INQUIRE, see Chapter 12, "File Operation I/O Statements".
- Additional control constructs

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

  For more information, see Chapter 7, "Execution Control".
- Additional intrinsic procedures

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

  For more information, see Chapter 9, "Intrinsic Procedures".
- Additional specification statements

  The following specification statements are new in Fortran 90:

  — The INTENT statement ("INTENT Attribute and Statement")

  — The OPTIONAL statement ("OPTIONAL Attribute and Statement")

  — The Fortran 90 POINTER statement ("POINTER Attribute and Statement")

  — The PUBLIC and PRIVATE statements ("PRIVATE and PUBLIC Attributes and Statements")

- Additional way to specify attributes

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

  For more information, see "Type Declaration Statements".

- Scope and Association

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

  For more information, see Chapter 15, "Scope and Association".

Program Structure, Characters, and Source Forms

2

This chapter contains information on the following topics:

- An overview of program structure, including general information on statements and names (see "Program Structure")
- "Character Sets"
- "Source Forms"

Program Structure

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

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

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

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

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

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

See Also

Chapter 8, "Program Units and Procedures", for details on program units and procedures

Statements

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

Order of Statements in a Program Unit

Figure 2-1 shows the required order of statements in a Fortran program unit. In this figure, vertical lines separate statement types that can be interspersed. For example, you can intersperse DATA statements with executable constructs.

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

Figure 2-1 Required Order of Statements

<table>
<tr><td rowspan="9">Comment Lines, INCLUDE Statements, and Directives</td><td colspan="3">OPTIONS Statements</td></tr>
<tr><td colspan="3">PROGRAM, FUNCTION. SUBROUTINE, MODULE, or BLOCK DATA Statement</td></tr>
<tr><td colspan="3">USE Statements</td></tr>
<tr><td rowspan="4">NAMELIST, FORMAT, and ENTRY Statements</td><td colspan="2">IMPLICIT NONE Statements</td></tr>
<tr><td>PARAMETER Statements</td><td>IMPLICIT Statements</td></tr>
<tr><td>PARAMETER and DATA Statements</td><td>Derived−Type Definitions, Interface Blocks, Type Declaration Statements, Statement Function Statements, and Specification Statements</td></tr>
<tr><td>DATA Statements</td><td>Executable Statements</td></tr>
<tr><td colspan="3">CONTAINS Statement</td></tr>
<tr><td colspan="3">Internal Subprograms or Module Subprograms</td></tr>
<tr><td colspan="4">END Statement</td></tr>
</table>

ZK−6516A−GE

Note that in this figure, INCLUDE statements, directives, OPTIONS statements, and the order of NAMELIST statements are language extensions.

PUBLIC and PRIVATE statements are only allowed in the scoping units of modules. In Fortran 95/90, NAMELIST statements can appear only among specification statements. However, Intel® Fortran allows them to also appear among executable statements.

Table 2-1 shows other statements restricted from different types of scoping units.

Table 2-1 Statements Restricted in Scoping Units

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

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

See Also

"Scope" for details on scoping units

Names

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

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

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

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

NOTE. *Be careful when defining names that contain dollar signs. On Linux* and Windows* systems, a dollar sign can be a symbol for command or symbol substitution in various shell and utility commands.*

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

- Program units
- External procedures
- Common blocks
- Modules

Examples

The following examples show valid and invalid names:

Valid

NUMBER

FIND_IT

X

Invalid	Explanation
5Q	Begins with a numeral.
B.4	Contains a special character other than _ or $.
_WRONG	Begins with an underscore.

See Also

"Scope" for details on the scope of names

Character Sets

Intel Fortran supports the following characters:

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

Character	Name	Character	Name
Δ or <Tab>	Blank (space) or tab	:	Colon
=	Equal sign	!	Exclamation point
+	Plus sign	"	Quotation mark
–	Minus sign	%	Percent sign
*	Asterisk	&	Ampersand

Character	Name	Character	Name
/	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) and ASCII characters with codes in the range 20(hex) through 7E(hex). Printable characters that are not in the Fortran 95/90 character set can only appear in comments, character constants, Hollerith constants, character string edit descriptors, and input/output records.

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

See Also

- Appendix C, "The ASCII Character Set for Linux Systems", for details on the ASCII character set for Linux systems
- The online documentation for Windows* systems for details on other character sets available for those systems

Source Forms

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

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

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

- Comment indicator

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

  An all blank line is also a comment line.

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

  For more information on comment indicators in free source form, see "Free Source Form"; in fixed and tab source forms, see "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. Intel Fortran allows up to 511 continuation lines in a source program.

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

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

  For more information on continuation indicators in free source form, see "Free Source Form"; in fixed and tab source forms, see "Fixed and Tab Source Forms".

Table 2-2 summarizes characters used as indicators in source forms:

Table 2-2 Indicators in Source Forms

Source Item	Indicator[1]	Source Form	Position
Comment	!	All forms	Anywhere in source code
Comment line	!	Free	At the beginning of the source line
	!, C, or *	Fixed	In column 1
		Tab	In column 1

Table 2-2 Indicators in Source Forms

Source Item	Indicator[1]	Source Form	Position
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 (see "Source Code 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.

See Also

- "Free Source Form" for details on labels in free source form
- "Fixed and Tab Source Forms" for details on labels in fixed and tab source forms

Free Source Form

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

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

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

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

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

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

Optional Blanks	Required Blanks
END TYPE	
END WHERE	
GO TO	
IN OUT	
SELECT CASE	

For information on statement separators (;) in all forms, see "Source Code Useable for All 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 95/90 permits up to 39 continuation lines in free-form programs, Intel Fortran allows up to 511 continuation lines.

The following shows a continued statement:

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

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

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

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

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

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

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

```
ARCHITECT  = "O'Connor, Emerson, and Davis&
               &  Associates"
```

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

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.

See Also

"Source Code Useable for All Source Forms" for details on the general rules for all source forms

Fixed and Tab Source Forms

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

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

By default, a statement can extend to character position 72. In this case, any text following position 72 is ignored and no warning message is printed. You can specify 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, Intel Fortran does not. If portability is a concern, you can use the concatenation operator to prevent source lines from being padded by other Fortran compilers (see the example in "Continuation Indicator" below) or you can force short source lines to be padded by using a compiler option.

Comment Indicator

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

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

Continuation Indicator

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

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

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

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

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

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

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

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

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

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

Debugging Statement Indicator

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

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

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

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

See Also

- "OPTIONS Statement"

- "Source Forms" for details on the general rules for all source forms, statement separators (;) in all forms, and statement labels
- Appendix A, "Deleted and Obsolescent Language Features", for details on obsolescent features in Fortran 95
- Your user's guide for details on compiler options

Fixed-Format Lines

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

The column positions for each field follow:

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

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

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

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

See Also

- "Source Forms" for details on the general rules for all source forms
- "Fixed and Tab Source Forms" for details on the general rules for 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.

Figure 2-2 shows equivalent source lines coded with tab and fixed source form.

Figure 2-2 Line Formatting Example

Format using TAB Character

C [TAB] FIRST VALUE

10 [TAB] I = J + 5*K +

[TAB] 1 L*M

[TAB] IVAL = I+2

Character−per−Column Format

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
C						F	I	R	S	T		V	A	L	U	E			
1	0					I		=		J		+		5	*	K		+	
					1		L	*	M										
						I	V	A	L		=	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 Intel Fortran compiler does not interpret the tab character in this way. It treats the tab character in a statement field the same way it treats a blank character. In the source listing that the compiler produces, the tab causes the character that follows to be printed at the next tab stop (usually located at columns 9, 17, 25, 33, and so on).

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

See Also

- "Source Forms" for details on the general rules for all source forms
- "Fixed and Tab Source Forms" for details on the general rules for fixed and tab source forms

Source Code Useable for All Source Forms

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

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 7 (or after the first tab character).
Comment indicator	Use only !. Place anywhere *except* in column position 6 (or immediately after the first tab character).
Continuation indicator	Use only &. Place in column position 73 of the initial line and each continuation line, and in column 6 of each continuation line (no tab character can precede the ampersand in column 6).

The following example is valid for all source forms:

```
Column:
12345678...                                                              73
_____________________________________________________________________________
! Define the user function MY_SIN
      DOUBLE PRECISION FUNCTION MY_SIN(X)
        MY_SIN = X - X**3/FACTOR(3) + X**5/FACTOR(5)                     &
     &         - X**7/FACTOR(7)
      CONTAINS
        INTEGER FUNCTION FACTOR(N)
         FACTOR = 1
         DO 10 I = N, 1, -1
  10     FACTOR = FACTOR * I
        END FUNCTION FACTOR
      END FUNCTION MY_SIN
```

Data Types, Constants, and Variables

3

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

Each *data type* has the following properties:

- A name

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

  Each data type has a set of valid values. Integer and real data types have a range of valid values. Complex and derived types have sets of values that are combinations of the values of their individual components.
- A way to represent constant values for the data type

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

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

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

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

This chapter contains information on the following topics:

- "Intrinsic Data Types" (This topic also discusses the forms for constants.)

- "Derived Data Types"
- "Binary, Octal, Hexadecimal, and Hollerith Constants"
- "Variables"

See Also

- "Type Declaration Statements"
- "Defined Operations"
- "Expressions" for details on valid operations for data types
- "PARAMETER Attribute and Statement" for details on named constants and the PARAMETER attribute
- Your user's guide for details on ranges for numeric literal constants

Intrinsic Data Types

Intel® Fortran provides the following intrinsic data types:

- INTEGER (see "Integer Data Types")

  There are four kind parameters for data of type integer:

  — INTEGER([KIND=]1) or INTEGER*1

  — INTEGER([KIND=]2) or INTEGER*2

  — INTEGER([KIND=]4) or INTEGER*4

  — INTEGER([KIND=]8) or INTEGER*8
- REAL (see "Real Data Types")

  There are three kind parameters for data of type real:

  — REAL([KIND=]4) or REAL*4

  — REAL([KIND=]8) or REAL*8

  — REAL([KIND=]16) or REAL*16
- DOUBLE PRECISION (see "Real Data Types")

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

  There are three kind parameters for data of type complex:

  — COMPLEX([KIND=]4) or COMPLEX*8

  — COMPLEX([KIND=]8) or COMPLEX*16

  — COMPLEX([KIND=]16) or COMPLEX*32

- DOUBLE COMPLEX (see "Complex Data Types")

  No kind parameter is permitted for data declared with type DOUBLE COMPLEX. This data type is the same as COMPLEX([KIND=]8).
- LOGICAL (see "Logical Data Types")

  There are four kind parameters for data of type logical:
  - LOGICAL([KIND=]1) or LOGICAL*1
  - LOGICAL([KIND=]2) or LOGICAL*2
  - LOGICAL([KIND=]4) or LOGICAL*4
  - LOGICAL([KIND=]8) or LOGICAL*8
- CHARACTER (see "Character Data Type")

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

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

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

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

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

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

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

See Also

- "Type Declaration Statements"
- "KIND"
- "Declaration Statements for Noncharacter Types" and "Declaration Statements for Character Types" for details on declaration statements for intrinsic data types
- "Expressions" for details on operations for intrinsic data types
- Table 15-2 for details on storage requirements for intrinsic data types

Integer Data Types

Integer data types can be specified as follows:

INTEGER

INTEGER([KIND=]*n*)

INTEGER***n*

n

Is kind 1, 2, 4, or 8.

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

- If the integer constant is within the default integer kind range, the kind is *default integer*. Default integer is INTEGER(4). You can change the default behavior by specifying the compiler option that controls the default integer kind.
- If the integer constant is outside the default integer kind range, the kind of the integer constant is the smallest integer kind which holds the constant.

Integer Constants

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

Integer constants take the following form:

[*s*]*n*[*n*...][_*k*]

s

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

n

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

k

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

An unsigned constant is assumed to be nonnegative.

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

[*s*][[*base*] #]nnn...

s

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

base

Is any constant from 2 through 36.

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

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

Examples

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

Valid

0

-127

+32123

47_2

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

The following integers (most of which are not base 10) are all assigned a value equal to 3,994,575 decimal:

```
I     = 2#1111001111001111001111
m     = 7#45644664
J     = +8#17171717
K     = #3CF3CF
n     = +17#2DE110
L     = 3994575
index = 36#2DM8F
```

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

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

See Also

- "Numeric Expressions" for details on integer constants used in expressions
- Your user's guide for details on the ranges for integer types and kinds

Real Data Types

Real data types can be specified as follows:

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

n

Is kind 4, 8, or 16.

If a kind parameter is specified, the real constant has the kind specified. If a kind parameter is not specified, the kind is *default real*. Default real is REAL(4). You can change the default behavior by specifying the compiler option that controls the default real kind.

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

General Rules for Real Constants

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

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

[*s*]*n*[*n*...][_*k*]

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

[*s*]*n*[*n*...]E[*s*]*nn*...[_*k*]

[*s*]*n*[*n*...]D[*s*]*nn*...

[*s*]*n*[*n*...]Q[*s*]*nn*...

s

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

n

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

k

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

Rules and Behavior

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

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

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

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

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

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

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

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

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

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

REAL(4) Constants

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

IEEE* S_floating format is used.

Examples

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

Valid

3.14159

3.14159_4

621712._4

–.00127

+5.0E3

2E–3_4

Invalid	Explanation
1,234,567.	Commas not allowed.
325E–47	Too small for REAL; this is a valid DOUBLE PRECISION constant.
–47.E47	Too large for REAL; this is a valid DOUBLE PRECISION constant.
625._6	6 is not a valid kind for reals.
100	Decimal point missing; this is a valid integer constant.
$25.00	Special character not allowed.

See Also

- "General Rules for Real Constants"

- Your user's guide for details on the format and range of REAL(4) data
- Your user's guide for details on compiler options affecting Real data

REAL(8) or DOUBLE PRECISION Constants

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

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

IEEE T_floating format is used.

Examples

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

Valid

123456789D+5

123456789E+5_8

+2.7843D00

–.522D–12

2E200_8

2.3_8

3.4E7_8

Invalid	Explanation
–.25D0_2	2 is not a valid kind for reals.
+2.7182812846182	No D exponent designator is present; this is a valid single-precision constant.
123456789.D400	Too large for any double-precision format.
123456789.D–400	Too small for any double-precision format.

See Also

- "General Rules for Real Constants"
- Your user's guide for details on the format and range of DOUBLE PRECISION (REAL(8)) data
- Your user's guide for details on compiler options affecting DOUBLE PRECISION data

REAL(16) Constants

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

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

IEEE X_floating format is used.

Examples

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

Valid

123456789Q4000

–1.23Q–400

+2.72Q0

1.88_16

Invalid	Explanation
1.Q5000	Too large.
1.Q–5000	Too small.

See Also

- "General Rules for Real Constants"
- Your user's guide for details on the format and range of REAL(16) data

Complex Data Types

Complex data types can be specified as follows:

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

n

Is kind 4, 8, or 16.

s

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

If a kind parameter is specified, the complex constant has the kind specified. If no kind parameter is specified, the kind of both parts is default real, and the constant is of type *default complex*. Default complex is COMPLEX(4). You can change the default behavior by specifying the compiler option that controls the default real kind.

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

General Rules for Complex Constants

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

The following is the general form of a complex constant:

(*c*,*c*)

c

Is as follows:

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

Note that the comma and parentheses are required.

COMPLEX(4) Constants

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

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

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

The rules for REAL(4) constants apply to REAL(4) constants used in COMPLEX constants. (See "General Rules for Real Constants" and "REAL(4) Constants" for the rules on forming REAL(4) constants.)

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

Examples

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

Valid

(1.7039,–1.70391)

(44.36_4,–12.2E16_4)

(+12739E3,0.)

(1,2)

Invalid	Explanation
(–1.23,)	Missing second integer or single-precision real constant.
(1.0, 2H12)	Hollerith constant not allowed.

See Also

- "General Rules for Complex Constants"
- Your user's guide for details on the format and range of COMPLEX (COMPLEX(4)) data
- Your user's guide for details on compiler options affecting REAL data

COMPLEX(8) or DOUBLE COMPLEX Constants

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

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

The rules for DOUBLE PRECISION (REAL(8)) constants also apply to the double precision portion of COMPLEX(8) or DOUBLE COMPLEX constants. (See "General Rules for Real Constants" and "REAL(8) or DOUBLE PRECISION Constants" for the rules on forming DOUBLE PRECISION constants.)

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

Examples

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

Valid

(1.7039,–1.7039D0)

(547.3E0_8,–1.44_8)

(1.7039E0,–1.7039D0)

(+12739D3,0.D0)

Invalid	Explanation
(1.23D0,)	Missing second constant.
(1D1,2H12)	Hollerith constants not allowed.
(1,1.2)	Neither constant is DOUBLE PRECISION; this is a valid single-precision constant.

See Also

- "General Rules for Complex Constants"
- Your user's guide for details on the format and range of DOUBLE COMPLEX data
- Your user's guide for details on compiler options affecting DOUBLE PRECISION data

COMPLEX(16) Constants

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

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

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

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

Examples

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

Valid

(1.7039,–1.7039Q2)

(547.3E0_16,–1.44)

(+12739Q3,0.Q0)

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

See Also

- "General Rules for Complex Constants"
- Your user's guide for details on the format and range of COMPLEX(16) data
- Your user's guide for details on compiler options affecting REAL data

Logical Data Types

Logical data types can be specified as follows:

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

n

Is kind 1, 2, 4, or 8.

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

Logical Constants

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

.TRUE.[_*k*]
.FALSE.[_*k*]

k

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

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

See Also

Your user's guide for details on integer data type ranges

Character Data Type

The character data type can be specified as follows:

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

n

Is kind 1.

len

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

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

Character Constants

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

[*k*_]'[*ch*...]' [*C*]
[k_]"[*ch*...]" [*C*]

k

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

ch

Is an ASCII character.

C

Is a C string specifier. C strings can be used to define strings with nonprintable characters. For more information, see "C Strings in Character Constants".

Rules and Behavior

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

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

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

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

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

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

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

Examples

The following examples show valid and invalid character constants:

Valid

"WHAT KIND TYPE? "

'TODAY"S DATE IS: '

"The average is: "

' '

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

See Also

"Declaration Statements for Character Types"

C Strings in Character Constants

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

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

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

Table 3-1 shows the escape sequences that are allowed in character constants.

Table 3-1 C-Style Escape Sequences

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

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

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

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

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

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

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

Character Substrings

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

v ([*e1*]:[*e2*])

a (*s* [,*s*] . . .) ([*e1*]:[*e2*])

v

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

e1

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

e2

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

a

Is the name of a character array.

s

Is a subscript expression.

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

Rules and Behavior

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

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

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

Examples

Consider the following example:

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

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

Consider the following example:

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

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

The following are valid substrings based on the above example:

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

Consider the following example:

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

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

See Also

- “Arrays”
- “Array Elements”
- “Structure Components”

Derived Data Types

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

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

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

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

See Also

- “Structure Components”

- "Structure Constructors"
- "Derived-Type Assignment Statements"
- "Record Structures"
- "Defining Generic Operators" for details on OPERATOR interfaces
- "Defining Generic Assignment" for details on ASSIGNMENT interfaces

Derived-Type Definition

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

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

access

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

name

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

component-definition

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

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

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

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

A component definition takes the following form:

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

type

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

attr

Is an optional POINTER attribute for a pointer component, or an optional DIMENSION or ALLOCATABLE attribute for an array component. You cannot specify both the ALLOCATABLE and POINTER attribute. If DIMENSION is specified, it can be followed by an array specification.

Each attribute can only appear once in a given *component-definition*.

component

Is the name of the component being defined.

a-spec

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

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

char-len

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

init-ex

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

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

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

Rules and Behavior

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

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

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

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

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

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

See Also

- "Intrinsic Data Types"
- "Arrays"
- "Structure Components"
- "Declaration Statements for Derived Types" for details on how to declare variables of derived type
- "POINTER Attribute and Statement" for details on pointers
- "Default Initialization"for details on default initialization for derived-type components
- Your user's guide for details on alignment of derived-type data components

Default Initialization

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

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

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

Explicit initialization in a type declaration statement overrides default initialization.

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

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

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

Examples

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

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

Consider the following:

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

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

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

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

TYPE (MGR_REPORT) STARTUP
```

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

Structure Components

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

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

parent

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

component

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

s-list

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

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

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

Rules and Behavior

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

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

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

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

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

Examples

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

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

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

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

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

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

```
CONTRACT%ID
```

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

```
TYPE DOT
  REAL X, Y
END TYPE DOT
 ....
```

```
TYPE SCREEN
  TYPE(DOT) C, D
END TYPE SCREEN
```

The following declares a variable of type SCREEN:

```
TYPE(SCREEN) M
```

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

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

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

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

```
MY_CAR%YEAR = 1985
```

The following shows an array structure component:

```
MY_CAR%MAKER
```

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

Consider the following:

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

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

Consider the following:

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

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

Some valid array references for this type follow:

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

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

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

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

The following example shows a private type:

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

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

See Also

- "Array Elements" for details on references to array elements
- "Array Sections" for details on references to array sections
- "Modules and Module Procedures" for examples of derived types in modules

Structure Constructors

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

d-name (*expr-list*)

d-name

Is the name of the derived type.

expr-list

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

Rules and Behavior

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

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

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

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

Examples

Consider the following derived-type definition:

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

This can be used to produce the following structure constructor:

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

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

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

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

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

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

See Also

"Pointer Assignments"

Binary, Octal, Hexadecimal, and Hollerith Constants

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

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

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

Binary Constants

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

B'*d*[*d*...]'

B"*d*[*d*...]"

d

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

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

Examples

The following examples show valid and invalid binary constants:

Valid

B'0101110'

B"1"

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

Octal Constants

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

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

d

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

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

Examples

The following examples show valid and invalid octal constants:

Valid

O'07737'

O"1"

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

See Also

"Alternative Syntax for Octal and Hexadecimal Constants"

Hexadecimal Constants

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

Z'*d*[*d*...]'
Z"*d*[*d*...]"

d

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

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

Examples

The following examples show valid and invalid hexadecimal constants:

Valid

Z'AF9730'

Z"FFABC"

Z'84'

Invalid	Explanation
Z'999.'	Decimal not allowed.
"ZF9"	No quotation mark after the Z.

See Also

"Alternative Syntax for Octal and Hexadecimal Constants"

Hollerith Constants

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

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

Examples

The following examples show valid and invalid Hollerith constants:

Valid

16HTODAY'S DATE IS:

1HB

4H ABC

Invalid	Explanation
3HABCD	Wrong number of characters.
0H	Hollerith constants must contain at least one character.

Determining the Data Type of Nondecimal Constants

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

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

Statement	Data Type of Constant	Length of Constant[1]
INTEGER(2) ICOUNT		
INTEGER(4) jCOUNT		
INTEGER(4) N		
REAL(8) DOUBLE		
REAL(4) RAFFIA, RALPHA		
RAFFIA = B'1001100111111010011'	REAL(4)	4
RAFFIA = Z'99AF2'	REAL(4)	4
RALPHA = 4HABCH	REAL(4)	4
DOUBLE = B'11111111111100110011010'	REAL(8)	8
DOUBLE = Z'FFF99A'	REAL(8)	8
DOUBLE = 8HABCDEFGH	REAL(8)	8
JCOUNT = ICOUNT + B'011101110111'	INTEGER(2)	2
JCOUNT = ICOUNT + O'777'	INTEGER(2)	2
JCOUNT = ICOUNT + 2HXY	INTEGER(2)	2
IF (N .EQ. B'1010100') GO TO 10	INTEGER(4)	4
IF (N .EQ. O'123') GO TO 10	INTEGER(4)	4
IF (N .EQ. 1HZ) GO TO 10	INTEGER(4)	4

1. In bytes.

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

Statement	Data Type of Constant	Length of Constant[1]
Y(1X) = Y (O'15') + 3	INTEGER(4)	4
Y(1X) = Y (1HA) + 3	INTEGER(4)	4

1. In bytes.

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

- For binary, octal, and hexadecimal constants, INTEGER(8) is assumed on Intel® Itanium® processors. On IA-32 processors, INTEGER(4) is used.
- For Hollerith constants, no data type is assumed.

For example:

Statement	Data Type of Constant	Length of Constant[1]
CALL APAC (Z'34BC2')	INTEGER(4)	4
CALL APAC (9HABCDEFGHI)	None	9

1. In bytes.

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

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

1. In bytes.
2. When two typeless constants are used in an operation, they both take default integer type.

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

- Binary, octal, and hexadecimal constants

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

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

  Table 15-2 lists the number of bytes that each data type requires.
- Hollerith constants

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

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

Each Hollerith character occupies one byte of memory.

See Also

Your user's guide for details on compiler options

Variables

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

- A scalar

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

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

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

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

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

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

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

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

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

See Also

- "Arrays"
- "Type Declaration Statements"
- "DATA Statement"
- "Data Type of Numeric Expressions"
- "Storage Association" for details on storage association of variables

Data Types of Scalar Variables

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

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

Specification of Data Type

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

```
COMPLEX VAR1
DOUBLE PRECISION VAR2
```

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

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

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

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

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

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

The argument CHARDUMMY assumes the length of the actual argument.

See Also

- "Type Declaration Statements"
- "Declaration Statements for Character Types"
- "Assumed-Length Character Arguments"
- "IMPLICIT Statement"

Implicit Typing Rules

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

Real Variables	Integer Variables
ALPHA	JCOUNT
BETA	ITEM_1
TOTAL_NUM	NTOTAL

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

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

See Also

- "Type Declaration Statements"
- "IMPLICIT Statement"

Arrays

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

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

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

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

An array has the following properties:

- Data type

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

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

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

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

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

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

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

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

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

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

Examples

The following are examples of valid array declarations:

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

Consider the following array declaration:

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

The properties of array L are as follows:

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

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

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

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

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

```
B = 99                  ! Reference to a whole array consisting of
                        !   elements: B(1), B(2), B(3), B(4), B(5),
                        !   B(6), B(7), B(8), B(9), and B(10)
```

See Also

- "DIMENSION Attribute and Statement"
- "Intrinsic Data Types"
- "Derived Data Types"
- "Whole Arrays"
- "Array Elements"
- "Array Sections"
- "Declaration Statements for Arrays" for details on array specifications
- Table 9-2 for details on intrinsic functions that perform array operations

Whole Arrays

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

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

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

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

```
L = 10              ! The value 10 is assigned to all the
                    !   elements in array L
WRITE *, L          ! Prints all the elements in array L
```

Array Elements

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

array(*subscript-list*)

array

Is the name of the array.

subscript-list

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

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

Rules and Behavior

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

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

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

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

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

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

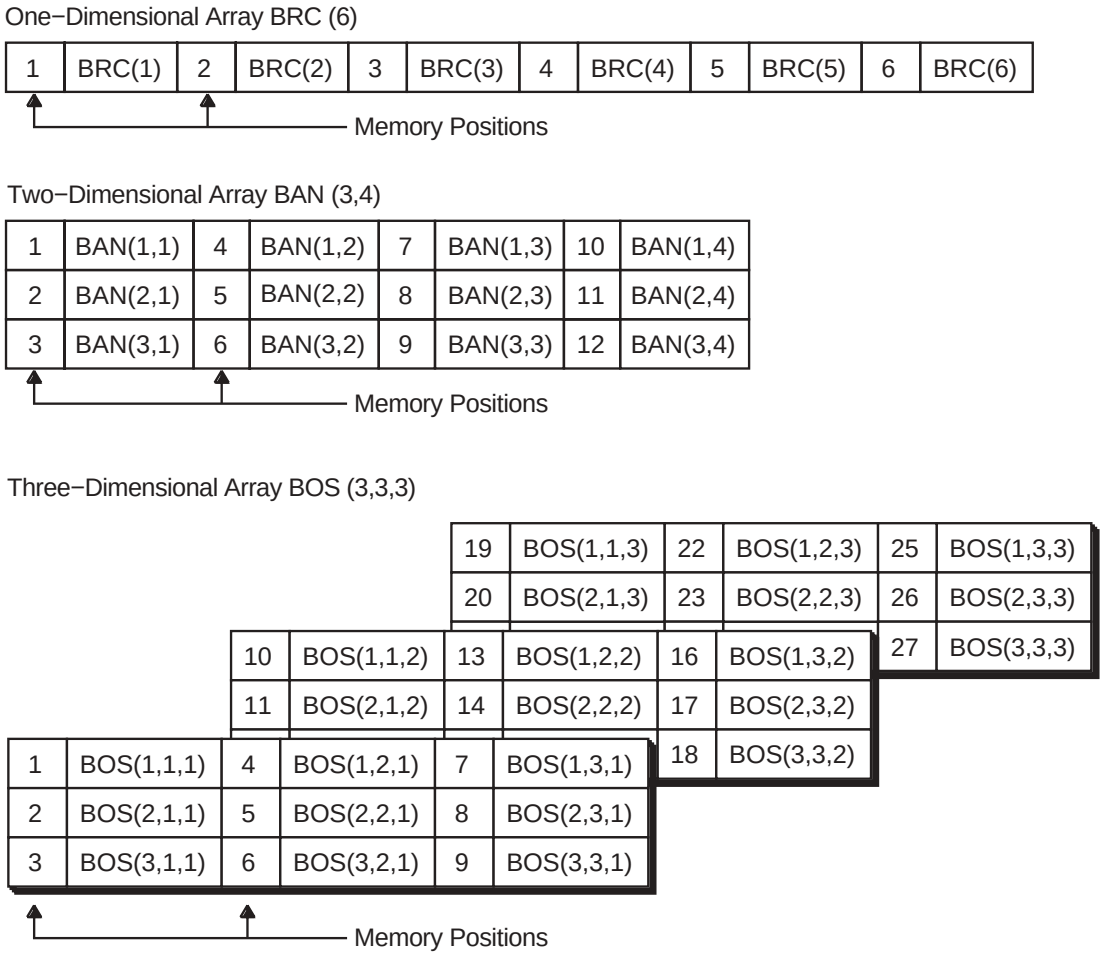
Array Element Order

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

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

Figure 3-1 shows array storage in one, two, and three dimensions

Figure 3-1 Array Storage



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

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

See Also

- "Character Substrings"
- "Array Association"
- "Structure Components" for details on arrays as structure components
- "Storage Association" for details on storage sequence association

Array Sections

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

array(*sect-subscript-list*)

array

Is the name of the array.

sect-subscript-list

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

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

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

Rules and Behavior

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

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

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

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

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

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

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

The following shows valid references to array sections. Note that the syntax (/.../) denotes an array constructor (see "Array Constructors"):

```
REAL, DIMENSION(20) :: B
...
PRINT *, B(2:20:5)  ! The section consists of elements
                    !     B(2), B(7), B(12), and B(17)
K = (/3, 1, 4/)
B(K) = 0.0         ! Section B(K) is a rank-one array with shape (3) and
                   !  size 3. (0.0 is assigned to B(1), B(3), and B(4).)
```

Subscript Triplets

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

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

first-bound

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

last-bound

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

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

stride

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

The stride has the following effects:

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

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

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

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

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

  ```
  A(10)
  A(8)
  A(6)
  A(4)
  ```

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

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

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

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

Vector Subscripts

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

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

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

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

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

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

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

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

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

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

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

See Also

- "INTENT Attribute and Statement"
- "PARAMETER Attribute and Statement"
- "TARGET Attribute and Statement"
- "Character Substrings"
- "Array Constructors"
- "Structure Components" for details on array sections as structure components

Array Constructors

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

(/*ac-value-list*/)

ac-value-list

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

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

(*ac-value-list*, *do-variable* = *expr1*, *expr2* [, *expr3*])

do-variable

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

expr

Is a scalar integer expression. The e*xpr1* and *expr2* specify a range of values for the loop; *expr3* specifies the stride. The *expr3* must be a nonzero value; if it is omitted, it is assumed to be 1.

Rules and Behavior

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

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

An *ac-value* is interpreted as follows:

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

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

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

You can also mix these forms, for example:

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

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

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

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

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

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

This causes the following to be displayed:

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

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

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

The following are alternative forms for array constructors:

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

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

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

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

Examples

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

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

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

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

```
TYPE EMPLOYEE
  INTEGER ID
  CHARACTER(LEN=30) NAME
END TYPE EMPLOYEE
TYPE(EMPLOYEE) CC_4T(4)
CC_4T = (/EMPLOYEE(2732,"JONES"), EMPLOYEE(0217,"LEE"),     &
          EMPLOYEE(1889,"RYAN"), EMPLOYEE(4339,"EMERSON")/)
```

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

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

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

```
3.5    1.0    4.7
2.0    2.3    6.6
```

See Also

- "DO Constructs"
- "RESHAPE"
- "Subscript Triplets"
- "Derived Data Types"
- "Structure Constructors"
- "Array Elements" for details on array element order
- "Array Assignment Statements" for details on another way to assign values to arrays
- "Declaration Statements for Arrays" for details on array specifications

Expressions and Assignment Statements

4

This chapter contains information on the following topics:

- "Expressions"
- "Assignment Statements"

Expressions

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

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

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

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

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

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

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

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

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

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

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

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

See Also

- "Arrays"
- "Derived Data Types"
- "Defining Generic Operators" for details on function subprograms that define operators
- "POINTER Attribute and Statement" for details on pointers

Numeric Expressions

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

The term *numeric* includes logical data, because logical data is treated as integer data when used in a numeric context. The default for .TRUE. is –1; .FALSE. is 0. The default can change if a specific compiler option is used.

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

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

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

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

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

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

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

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

Examples

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

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

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

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

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

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

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

Using Parentheses in Numeric Expressions

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

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

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

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

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

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

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

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

```
4 + (3 * 2) - (6/2)
```

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

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

Data Type of Numeric Expressions

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

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

Data Type	Ranking
LOGICAL(1) and BYTE	Lowest
LOGICAL(2)	.
LOGICAL(4)	.
LOGICAL(8)	.
INTEGER(1)	.
INTEGER(2)	.
INTEGER(3)	.
INTEGER(4)	.
REAL(4)	.
REAL(8)[1]	.
REAL(16)	.
COMPLEX(4)	.
COMPLEX(8)	.
COMPLEX(16)[2]	Highest

1. DOUBLE PRECISION
2. DOUBLE COMPLEX

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

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

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

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

  When a single-precision real operand is converted to a double-precision real operand, low-order binary digits are set to zero. This conversion does not increase accuracy; conversion of a decimal number does not produce a succession of decimal zeros. For example, a REAL variable having the value `0.3333333` is converted to approximately `0.3333333134651184D0`. It is not converted to either `0.3333333000000000D0` or `0.3333333333333333D0`.
- Complex operations: In operations that contain any complex operands, integer operands are converted to real type, as previously described. The resulting single-precision or double-precision operand is designated as the real part of a complex number and the imaginary part is assigned a value of zero. The expression is then evaluated using complex arithmetic and the resulting value is of complex type. Operations involving a COMPLEX(4) or COMPLEX(8) operand and a DOUBLE PRECISION operand are performed as COMPLEX(8) operations; the DOUBLE PRECISION operand is not rounded.

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

Character Expressions

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

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

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

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

Each of these expressions has the value `' ABCDEF'`.

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

Relational Expressions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

See Also

"Data Type of Numeric Expressions" for details on the ranking of data types

Logical Expressions

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

Operator	Example	Meaning
.AND.	A .AND. B	Logical conjunction: the expression is true if both A and B are true.
.OR.	A .OR. B	Logical disjunction (inclusive OR): the expression is true if either A, B, or both, are true.

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

1. .NOT. is a unary operator.

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

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

Data Types Resulting from Logical Operations

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

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

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

Logical operations cannot be performed on other data types.

Evaluation of Logical Expressions

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

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

This expression is evaluated in the following sequence:

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

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

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

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

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

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

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

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

See Also

"Summary of Operator Precedence" for details on the precedence of numeric, relational, and logical operators

Defined Operations

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

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

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

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

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

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

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

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

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

The following examples show expressions containing defined operators:

```
.COMPLEMENT. A
X .PLUS. Y .PLUS. Z
```

```
M * .MINUS. N
```

See Also

- "Defining Generic Operators"
- "Summary of Operator Precedence"

Summary of Operator Precedence

Table 4-1 shows the precedence of all intrinsic and defined operators:

Table 4-1 Precedence of Expression Operators

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

Initialization and Specification Expressions

A constant expression contains intrinsic operations and parts that are all constants. An initialization expression is a constant expression that is evaluated when a program is compiled. A specification expression is a scalar, integer expression that is restricted to declarations of array bounds and character lengths.

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

Initialization Expressions

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

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

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

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

- Each function argument must be one of the following:
  - An initialization expression
  - A variable whose kind type parameter and bounds are not assumed or defined by an ALLOCATE statement, pointer assignment, or an expression that is not an initialization expression
- A reference to one of the following transformational functions (each argument must be an initialization expression): functions:

REPEAT	SELECTED_REAL_KIND
RESHAPE	TRANSFER
SELECTED_INT_KIND	TRIM

- A reference to the transformational function NULL
- An implied-DO variable within an array constructor where the bounds and strides of the corresponding implied-DO are initialization expressions

- Another initialization expression enclosed in parentheses

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

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

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

Examples

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

Valid

–1 + 3	
SIZE(B)	! B is a named constant
7_2	
INT(J, 4)	! J is a named constant
SELECTED_INT_KIND (2)	

Invalid	**Explanation**
SUM(A)	Not an allowed function.
A/4.1 – K**1.2	Exponential does not have integer power (A and K are named constants).
HUGE(4.0)	Argument is not an integer.

See Also

- "Array Constructors"
- "Structure Constructors"
- "Intrinsic Procedures" for details on intrinsic functions

Specification Expressions

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

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

- A constant or subobject of a constant
- A variable that is one of the following:

— A dummy argument that does not have the OPTIONAL or INTENT (OUT) attribute (or the subobject of such a variable)
— In a common block (or the subobject of such a variable)
— Made accessible by use or host association (or the subobject of such a variable)

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

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

Each function argument must be one of the following:
— A restricted expression
— A variable whose properties inquired about are not dependent on the upper bound of the last dimension of an assumed-size array, are not defined by an expression that is not a restricted expression, or are not definable by an ALLOCATE or pointer assignment statement.

- A reference to any other intrinsic function where each argument is a restricted expression.
- A reference to a specification function where each argument is a restricted expression
- An array constructor where each element and the bounds and strides of each implied-DO, are expressions whose primaries are restricted expressions
- Another restricted expression enclosed in parentheses

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

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

- An intrinsic function

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

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

- By a previous declaration in the same scoping unit
- By the implicit typing rules currently in effect for the scoping unit
- By host or use association

If a variable in a specification expression is typed by the implicit typing rules, its appearance in any subsequent type declaration statement must confirm the implied type and type parameters.

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

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

Examples

The following shows valid specification expressions:

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

See Also

- "Array Constructors"
- "Implicit Typing Rules"
- "Structure Constructors"
- "Use and Host Association"
- "Pure Procedures"
- Chapter 9, "Intrinsic Procedures", for details on intrinsic functions

Assignment Statements

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

The ASSIGN statement assigns a label to an integer variable. It is discussed in "The ASSIGN and Assigned GO TO Statements".

Intrinsic Assignments

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

An intrinsic assignment statement takes the following form:

variable = *expression*

variable

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

expression

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

Rules and Behavior

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

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

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

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

See Also

- "Arrays"
- "Derived Data Types"
- "Defining Generic Assignment" for details on subroutine subprograms that define assignment

- "POINTER Attribute and Statement" for details on pointers

Numeric Assignment Statements

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

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

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

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

Table 4-2 summarizes the data conversion rules for numeric assignment statements.

Table 4-2 Conversion Rules for Numeric Assignment Statements

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

Examples

The following examples show valid and invalid numeric assignment statements:

Valid

BETA = -1./(2.*X)+A*A /(4.*(X*X))

PI = 3.14159

SUM = SUM + 1.

ARRAY_A = ARRAY_B + ARRAY_C + SCALAR_I ! Valid if all arrays conform in shape.

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

See Also

- "INT"
- "REAL"
- "DBLE"
- "QEXT"
- "CMPLX"
- "AIMAG"

Logical Assignment Statements

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

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

Examples

The following examples show valid logical assignment statements:

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

Character Assignment Statements

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

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

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

Examples

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

Valid

```
FILE = 'PROG2'
REVOL(1) = 'MAR'//'CIA'
LOCA(3:8) = 'PLANT5'
TEXT(I,J+1)(2:N-1) = NAME/ /X
```

Invalid	Explanation
'ABC'= CHARS	Left element must be a character variable, array element, or substring reference.
CHARS = 25	Expression does not have a character data type.
STRING = 5HBEGIN	Expression does not have a character data type. (Hollerith constants are numeric, not character.)

Derived-Type Assignment Statements

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

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

Examples

The following example shows derived-type assignment:

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

TYPE(DATE) TODAY, THIS_WEEK(7)

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

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

See Also

- "Derived Data Types"
- "Pointer Assignments"

Array Assignment Statements

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

If the expression is a scalar, and the variable is an array, the scalar value is assigned to every element of the array.

If the expression is an array, the variable must also be an array. The array element values of the expression are assigned (element by element) to corresponding elements of the array variable.

A *many-one array section* is a vector-valued subscript that has two or more elements with the same value. In intrinsic assignment, the variable cannot be a many-one array section because the result of the assignment is undefined.

Examples

In the following example, X and Y are arrays of the same shape:

```
X = Y
```

The corresponding elements of Y are assigned to those of X element by element; the first element of Y is assigned to the first element of X, and so forth. The processor can perform the element-by-element assignment in any order.

The following example shows a scalar assigned to an array:

```
B(C+1:N, C) = 0
```

This sets the elements B (C+1,C), B (C+2,C),...B (N,C) to zero.

The following example causes the values of the elements of array A to be reversed:

```
REAL A(20) ... A(1:20) = A(20:1:-1)
```

See Also

- "Arrays"
- "WHERE Statement and Construct" for details on masked array assignment
- "FORALL Statement and Construct" for details on element array assignment

Defined Assignments

Defined assignment specifies an assignment operation. It is defined by a subroutine subprogram containing a generic interface block with the specifier ASSIGNMENT(=). The subroutine is specified by a SUBROUTINE or ENTRY statement that has two nonoptional dummy arguments.

Defined elemental assignment is indicated by specifying ELEMENTAL in the SUBROUTINE statement.

The dummy arguments represent the variable and expression, in that order. The rank (and shape, if either or both are arrays), type, and kind parameters of the variable and expression in the assignment statement must match those of the corresponding dummy arguments.

The dummy arguments must not both be numeric, or of type logical or character with the same kind parameter.

If the variable in an elemental assignment is an array, the defined assignment is performed element-by-element, in any order, on corresponding elements of the variable and expression. If the expression is scalar, it is treated as if it were an array of the same shape as the variable with every element of the array equal to the scalar value of the expression.

See Also

- "Derived Data Types"
- "Subroutines" for details on subroutine subprograms
- "Defining Generic Assignment" for details on subroutine subprograms that define assignment
- "Numeric Expressions" and "Character Expressions" for details on intrinsic operations

Pointer Assignments

In ordinary assignment involving pointers, the pointer is an alias for its target. In pointer assignment, the pointer is associated with a target. If the target is undefined or disassociated, the pointer acquires the same status as the target. The pointer assignment statement has the following form:

pointer-object => target

pointer-object

Is a variable name or structure component declared with the POINTER attribute.

target

Is a variable or expression. Its type and kind parameters, and rank must be the same as *pointer-objec*t. It cannot be an array section with a vector subscript.

Rules and Behavior

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

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

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

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

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

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

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

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

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

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

Examples

The following are examples of pointer assignments:

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

The following example shows a target as a pointer:

```
INTEGER, POINTER :: P, N
INTEGER, TARGET :: M
INTEGER S
M = 14
N => M                         ! N is associated with M
P => N                         ! P is associated with M through N
S = P + 5
```

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

See Also

- "Arrays"
- "Defined Assignments"
- "NULL"
- "POINTER Attribute and Statement" for details on pointers
- Chapter 6, "Dynamic Allocation", for details on the ALLOCATE, DEALLOCATE, and NULLIFY statements
- "Intrinsic Assignments" for details on derived-type intrinsic assignments

WHERE Statement and Construct

The WHERE statement and construct let you use masked array assignment, which performs an array operation on selected elements. This kind of assignment applies a logical test to an array on an element-by-element basis.

The WHERE statement takes the following form:

WHERE (*mask-expr1*) *assign-stmt*

The WHERE construct takes the following form:

[*name*:] WHERE (*mask-expr1*)
  [*where-body-stmt*]...
[ELSE WHERE (*mask-expr2*) [*name*]
  [*where-body-stmt*]...]

[ELSE WHERE [*name*]
[*where-body-stmt*]...]
END WHERE [*name*]

mask-expr1, *mask-expr2*

Are logical array expressions (called mask expressions).

assign-stmt

Is an assignment statement of the form: array variable = array expression.

name

Is the name of the WHERE construct.

where-body-stmt

Is one of the following:

- An *assign-stmt*

  The assignment can be a defined assignment only if the routine implementing the defined assignment is elemental.
- A WHERE statement or construct

Rules and Behavior

If a construct name is specified in a WHERE statement, the same name must appear in the corresponding END WHERE statement. The same construct name can optionally appear in any ELSE WHERE statement in the construct. (ELSE WHERE cannot specify a different name.)

In each assignment statement, the mask expression, the variable being assigned to, and the expression on the right side, must all be conformable. Also, the assignment statement cannot be a defined assignment.

Only the WHERE statement (or the first line of the WHERE construct) can be labeled as a branch target statement.

The following is an example of a WHERE statement:

```
INTEGER A, B, C
DIMENSION A(5), B(5), C(5)
DATA A /0,1,1,1,0/
DATA B /10,11,12,13,14/
C = -1

WHERE(A .NE. 0) C = B / A
```

The resulting array C contains: –1,11,12,13, and –1.

The assignment statement is only executed for those elements where the mask is true. Think of the mask expression as being evaluated first into a logical array that has the value true for those elements where A is positive. This array of trues and falses is applied to the arrays A, B and C in the assignment statement. The right side is only evaluated for elements for which the mask is true; assignment on the left side is only performed for those elements for which the mask is true. The elements for which the mask is false do not get assigned a value.

In a WHERE construct, the mask expression is evaluated first and only once. Every assignment statement following the WHERE is executed as if it were a WHERE statement with "*mask-expr1*" and every assignment statement following the ELSE WHERE is executed as if it were a WHERE statement with ".NOT. *mask-expr1*". If ELSE WHERE specifies "*mask-expr2*", it is executed as "(.NOT. *mask-expr1*) .AND. *mask-expr2*" during the processing of the ELSE WHERE statement.

You should be careful if the statements have side effects, or modify each other or the mask expression.

The following is an example of the WHERE construct:

```
DIMENSION PRESSURE(1000), TEMP(1000), PRECIPITATION(1000)
WHERE(PRESSURE .GE. 1.0)
  PRESSURE = PRESSURE + 1.0
  TEMP = TEMP - 10.0
ELSEWHERE
  PRECIPITATION = .TRUE.
ENDWHERE
```

The mask is applied to the arguments of functions on the right side of the assignment if they are considered to be elemental functions. Only elemental intrinsics are considered elemental functions. Transformational intrinsics, inquiry intrinsics, and functions or operations defined in the subprogram are considered to be nonelemental functions.

Consider the following example using LOG, an elemental function:

```
WHERE(A .GT. 0)  B = LOG(A)
```

The mask is applied to A, and LOG is executed only for the positive values of A. The result of the LOG is assigned to those elements of B where the mask is true.

Consider the following example using SUM, a nonelemental function:

```
REAL A, B
DIMENSION A(10,10), B(10)
WHERE(B .GT. 0.0)  B = SUM(A, DIM=1)
```

Since SUM is nonelemental, it is evaluated fully for all of A. Then, the assignment only happens for those elements for which the mask evaluated to true.

Consider the following example:

```
REAL A, B, C
DIMENSION A(10,10), B(10), C(10)
WHERE(C .GT. 0.0)  B = SUM(LOG(A), DIM=1)/C
```

Because SUM is nonelemental, all of its arguments are evaluated fully regardless of whether they are elemental or not. In this example, LOG(A) is fully evaluated for all elements in A even though LOG is elemental. Notice that the mask is applied to the result of the SUM and to C to determine the right side. One way of thinking about this is that everything inside the argument list of a nonelemental function does not use the mask, everything outside does.

See Also

"FORALL Statement and Construct" for details on a generalized form of masked array assignment

FORALL Statement and Construct

The FORALL statement and construct is a generalization of the Fortran 95/90 masked array assignment (WHERE statement and construct). It allows more general array shapes to be assigned, especially in construct form.

FORALL is a feature of Fortran 95. It takes the following form:

FORALL (*triplet-spec* [, *triplet-spec*]...[, *mask-expr*]) *assign-stmt*

The FORALL construct takes the following form:

[*name*:] FORALL (*triplet-spec* [, *triplet-spec*]...[, *mask-expr*])
forall-body-stmt
[*forall-body-stmt*]...
END FORALL [*name*]

triplet-spec

Is a triplet specification with the following form:

subscript-name = *subscript-1* : *subscript-2* [:*stride*]

The *subscript-name* must be a scalar of type integer. It is valid only within the scope of the FORALL; its value is undefined on completion of the FORALL.

The *subscript*s and *stride* cannot contain a reference to any *subscript-name* in *triplet-spec*.

The *stride* cannot be zero. If it is omitted, the default value is 1.

Evaluation of an expression in a triplet specification must not affect the result of evaluating any other expression in another triplet specification.

mask-expr

Is a logical array expression (called the mask expression). If it is omitted, the value .TRUE. is assumed. The mask expression can reference the subscript name in *triplet-spec*.

assign-stmt

Is an assignment statement or a pointer assignment statement. It may be a scalar or array assignment statement, or a defined assignment statement. The variable being defined will normally use each *subscript-name* in the *triplet-spec*.

name

Is the name of the FORALL construct.

forall-body-stmt

Is one of the following:

- An *assign-stmt*
- A WHERE statement or construct

  The WHERE statement and construct use a mask to make the array assignments (see "WHERE Statement and Construct").
- A FORALL statement or construct

Rules and Behavior

If a construct name is specified in the FORALL statement, the same name must appear in the corresponding END FORALL statement.

A FORALL statement is executed by first evaluating all bounds and stride expressions in the triplet specifications, giving a set of values for each subscript name. The FORALL assignment statement is executed for all combinations of subscript name values for which the mask expression is true.

The FORALL assignment statement is executed as if all expressions (on both sides of the assignment) are completely evaluated before any part of the left side is changed. Valid values are assigned to corresponding elements of the array being assigned to. No element of an array can be assigned a value more than once.

A FORALL construct is executed as if it were multiple FORALL statements, with the same triplet specifications and mask expressions. Each statement in the FORALL body is executed completely before execution begins on the next FORALL body statement.

Any procedure referenced in the mask expression or FORALL assignment statement must be pure.

Pure functions can be used in the mask expression or called directly in a FORALL statement. Pure subroutines cannot be called directly in a FORALL statement, but can be called from other pure procedures.

Examples

Consider the following:

```
FORALL(I = 1:N, J = 1:N, A(I, J) .NE. 0.0) B(I, J) = 1.0 / A(I, J)
```

This statement takes the reciprocal of each nonzero element of array A(1:N, 1:N) and assigns it to the corresponding element of array B. Elements of A that are zero do not have their reciprocal taken, and no assignments are made to corresponding elements of B.

Every array assignment statement and WHERE statement can be written as a FORALL statement, but some FORALL statements cannot be written using just array syntax. For example, the preceding FORALL statement is equivalent to the following:

```
WHERE(A /= 0.0) B = 1.0 / A
```

It is also equivalent to:

```
FORALL (I = 1:N, J = 1:N)
  WHERE(A(I, J) .NE. 0.0) B(I, J) = 1.0/A(I, J)
END FORALL
```

However, the following FORALL example cannot be written using just array syntax:

```
FORALL(I = 1:N, J = 1:N) H(I, J) = 1.0/REAL(I + J - 1)
```

This statement sets array element H(I, J) to the value `1.0 /REAL(I + J - 1)` for values of I and J between 1 and N.

Consider the following:

```
TYPE MONARCH
   INTEGER, POINTER :: P
END TYPE MONARCH

TYPE(MONARCH), DIMENSION(8)   :: PATTERN
INTEGER, DIMENSION(8), TARGET :: OBJECT
FORALL(J=1:8)  PATTERN(J)%P => OBJECT(1+IEOR(J-1,2))
```

This FORALL statement causes elements 1 through 8 of array PATTERN to point to elements 3, 4, 1, 2, 7, 8, 5, and 6, respectively, of OBJECT. IEOR can be referenced here because it is pure.

The following example shows a FORALL construct:

```
FORALL(I = 3:N + 1, J = 3:N + 1)
   C(I, J) = C(I, J + 2) + C(I, J - 2) + C(I + 2, J) + C(I - 2, J)
   D(I, J) = C(I, J)
END FORALL
```

The assignment to array D uses the values of C computed in the first statement in the construct, not the values before the construct began execution.

See Also

- "Subscript Triplets"
- "Pointer Assignments"
- "WHERE Statement and Construct"
- "Pure Procedures"

Specification Statements

5

A *specification statement* is a nonexecutable statement that declares the attributes of data objects. In Fortran 95/90, many of the attributes that can be defined in specification statements can also be optionally specified in type declaration statements.

This chapter contains information on the following topics:

- "Type Declaration Statements"
  Explicitly specifies the properties (for example: data type, rank, and extent) of data objects.
- "ALLOCATABLE Attribute and Statement"
  Specifies a list of array names that are allocatable (have a deferred-shape).
- "AUTOMATIC and STATIC Attributes and Statements"
  Control the storage allocation of variables in subprograms.
- "COMMON Statement"
  Defines one or more contiguous areas, or blocks, of physical storage (called common blocks).
- "DATA Statement"
  Assigns initial values to variables before program execution.
- "DIMENSION Attribute and Statement"
  Specifies that an object is an array, and defines the shape of the array.
- "EQUIVALENCE Statement"
  Specifies that a storage area is shared by two or more objects in a program unit.
- "EXTERNAL Attribute and Statement"
  Allows external (user-supplied) procedures to be used as arguments to other subprograms.
- "IMPLICIT Statement"
  Overrides the implicit data type of names.
- "INTENT Attribute and Statement"
  Specifies the intended use of a dummy argument.

- "INTRINSIC Attribute and Statement"
  Allows intrinsic procedures to be used as arguments to subprograms.
- "NAMELIST Statement"
  Associates a name with a list of variables. This group name can be referenced in some input/output operations.
- "OPTIONAL Attribute and Statement"
  Allows a procedure reference to omit arguments.
- "PARAMETER Attribute and Statement"
  Defines a named constant.
- "POINTER Attribute and Statement"
  Specifies that an object is a pointer.
- "PRIVATE and PUBLIC Attributes and Statements"
  Declare the accessibility of entities in a module.
- "SAVE Attribute and Statement"
  Causes the definition and status of objects to be retained after the subprogram in which they are declared completes execution.
- "TARGET Attribute and Statement"
  Specifies a pointer target.
- "VOLATILE Attribute and Statement"
  Prevents optimizations from being performed on specified objects.

See Also

Chapter 8, "Program Units and Procedures", for details on BLOCK DATA and PROGRAM statements

Type Declaration Statements

A type declaration statement explicitly specifies the properties of data objects or functions.

The general form of a type declaration statement follows:

type [[, *att*]... ::] *v* [/*c-list*/] [, v [/*c-list*/]]...

type

Is one of the following data type specifiers:

BYTE	DOUBLE COMPLEX
INTEGER[([KIND=]k)]	CHARACTER[([LEN=]n)[,[KIND=]k]]

REAL[([KIND=]k)]	LOGICAL[([KIND=]k)]
DOUBLE PRECISION	TYPE (derived-type-name)
COMPLEX[([KIND=]k)]	

In the optional kind selector "([KIND=]*k*) ", *k* is the kind parameter. It must be an acceptable kind parameter for that data type. If the kind selector is not present, entities declared are of default type. (For a list of the valid noncharacter data types, see Table 5-2.)

Kind parameters for intrinsic numeric and logical data types can also be specified using the **n* format, where *n* is the length (in bytes) of the entity; for example, INTEGER*4.

att

Is one of the following attribute specifiers:

ALLOCATABLE	POINTER
AUTOMATIC	PRIVATE[1]
DIMENSION	PUBLIC[1]
EXTERNAL	SAVE
INTENT	STATIC
INTRINSIC	TARGET
OPTIONAL	VOLATILE
PARAMETER	

1. These are access specifiers.

v

Is the name of a data object or function. It can optionally be followed by:

- An array specification, if the object is an array.

  In a function declaration, an array must be a deferred-shape array if it has the POINTER attribute; otherwise, it must be an explicit-shape array.
- A character length, if the object is of type character.
- An initialization expression or, for pointer objects, => NULL( ).

A function name must be the name of an intrinsic function, external function, function dummy procedure, or statement function.

c-list

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

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

Rules and Behavior

Type declaration statements must precede all executable statements.

In most cases, a type declaration statement overrides (or confirms) the implicit type of an entity. However, a variable that appears in a DATA statement and is typed implicitly can appear in a subsequent type declaration only if that declaration confirms the implicit typing.

The double colon separator (::) is required only if the declaration contains an attribute specifier or initialization; otherwise it is optional.

If *att* appears, *c-list* cannot be specified; for example:

```
INTEGER I /2/                 ! Valid
INTEGER, SAVE :: I /2/        ! Invalid
```

The same attribute must not appear more than once in a given type declaration statement, and an entity cannot be given the same attribute more than once in a scoping unit.

If the PARAMETER attribute is specified, the declaration must contain an initialization expression.

If => NULL( ) is specified for a pointer, its initial association status is disassociated.

A variable (or variable subobject) can only be initialized once in an executable program.

If a declaration contains an initialization expression, but no PARAMETER attribute is specified, the object is a variable whose value is initially defined. The object becomes defined with the value determined from the initialization expression according to the rules of intrinsic assignment.

The presence of initialization implies that the name of the object is saved, except for objects in named common blocks or objects with the PARAMETER attribute.

The following objects cannot be initialized in a type declaration statement:

- A dummy argument
- A function result
- An object in a named common block (unless the type declaration is in a block data program unit)
- An object in blank common
- An allocatable array
- An external name
- An intrinsic name
- An automatic object
- An object that has the AUTOMATIC attribute

An object can have more than one attribute. Table 5-1 lists the compatible attributes.

Table 5-1 Compatible Attributes

Attribute	Compatible with:
ALLOCATABLE	AUTOMATIC, DIMENSION[1], PRIVATE, PUBLIC, SAVE, STATIC, TARGET, VOLATILE
AUTOMATIC	ALLOCATABLE, DIMENSION, POINTER, TARGET, VOLATILE
DIMENSION	ALLOCATABLE, AUTOMATIC, INTENT, OPTIONAL, PARAMETER, POINTER, PRIVATE, PUBLIC, SAVE, STATIC, TARGET, VOLATILE
EXTERNAL	OPTIONAL, PRIVATE, PUBLIC
INTENT	DIMENSION, OPTIONAL, TARGET, VOLATILE
INTRINSIC	PRIVATE, PUBLIC
OPTIONAL	DIMENSION, EXTERNAL, INTENT, POINTER, TARGET, VOLATILE
PARAMETER	DIMENSION, PRIVATE, PUBLIC
POINTER	AUTOMATIC, DIMENSION[1], OPTIONAL, PRIVATE, PUBLIC, SAVE, STATIC, VOLATILE
PRIVATE	ALLOCATABLE, DIMENSION, EXTERNAL, INTRINSIC, PARAMETER, POINTER, SAVE, STATIC, TARGET, VOLATILE
PUBLIC	ALLOCATABLE, DIMENSION, EXTERNAL, INTRINSIC, PARAMETER, POINTER, SAVE, STATIC, TARGET, VOLATILE
SAVE	ALLOCATABLE, DIMENSION, POINTER, PRIVATE, PUBLIC, SAVE, TARGET, VOLATILE
STATIC	ALLOCATABLE, DIMENSION, POINTER, PRIVATE, PUBLIC, SAVE, TARGET, VOLATILE
TARGET	ALLOCATABLE, AUTOMATIC, DIMENSION, INTENT, OPTIONAL, PRIVATE, PUBLIC, SAVE, STATIC, VOLATILE
VOLATILE	ALLOCATABLE, AUTOMATIC, DIMENSION, INTENT, OPTIONAL, POINTER, PRIVATE, PUBLIC, SAVE, STATIC, TARGET

1. With deferred shape

Examples

The following show valid type declaration statements:

```
DOUBLE PRECISION B(6)
INTEGER(KIND=2) I
REAL(KIND=4) X, Y
REAL(4) X, Y
LOGICAL, DIMENSION(10,10) :: ARRAY_A, ARRAY_B
```

```
INTEGER, PARAMETER :: SMALLEST = SELECTED_REAL_KIND(6, 70)
REAL(KIND (0.0)) M
COMPLEX(KIND=8) :: D
TYPE(EMPLOYEE) :: MANAGER
REAL, INTRINSIC :: COS CHARACTER(15) PROMPT
CHARACTER*12, SAVE :: HELLO_MSG
INTEGER COUNT, MATRIX(4,4), SUM
LOGICAL*2 SWITCH
REAL :: X = 2.0
TYPE (NUM), POINTER :: FIRST => NULL()
```

See Also

- "Derived Data Types"
- "Implicit Typing Rules"
- "DATA Statement"
- "Initialization Expressions"
- "Intrinsic Data Types" for details on specific kind parameters of intrinsic data types

Declaration Statements for Noncharacter Types

Table 5-2 shows the data types that can appear in noncharacter type declaration statements.

Table 5-2 Noncharacter Data Types

BYTE[1]
LOGICAL[2]
LOGICAL(1) (or LOGICAL*1)
LOGICAL(2) (or LOGICAL*2)
LOGICAL(4) (or LOGICAL*4)
LOGICAL(8) (or LOGICAL*8)
INTEGER[3]
INTEGER(1) (or INTEGER*1)
INTEGER(2) (or INTEGER*2)
INTEGER(4) (or INTEGER*4)
INTEGER(8) (or INTEGER*8)
REAL[4]
REAL(4) (or REAL*4)

Table 5-2 Noncharacter Data Types

DOUBLE PRECISION (REAL(8) or REAL*8)
REAL(16) (or REAL*16)
COMPLEX[5]
COMPLEX(4) (or COMPLEX*8)
DOUBLE COMPLEX (COMPLEX(8) or COMPLEX*16)
COMPLEX(16) (or COMPLEX*32)

1. Same as INTEGER(1).
2. This is treated as default logical.
3. This is treated as default integer.
4. This is treated as default real.
5. This is treated as default complex.

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

The value for *n* must be a valid length for the type of *v* (see Table 15-2). The type specifiers BYTE, DOUBLE PRECISION, and DOUBLE COMPLEX have one valid length, so the *n* specifier is invalid for them.

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

Examples

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

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

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

See Also

- "Type Declaration Statements" for details on the general form and rules for type declaration statements

- Your user's guide for details on compiler options that can affect the defaults for numeric and logical data types

Declaration Statements for Character Types

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

Keyword Forms

CHARACTER [([LEN=]*len*)]

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

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

Nonkeyword Form

CHARACTER***len*[,]

len

Is one of the following:

- In keyword forms

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

  If the length evaluates to a negative value, the length of the character entity is zero.
- In nonkeyword form

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

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

The largest valid value for *len* in both forms is 2**31–1 on IA-32 processors; 2**63–1 on Intel® Itanium® processors. Negative values are treated as zero.

n

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

Rules and Behavior

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

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

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

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

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

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

Examples

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

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

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

```
INTEGER, PARAMETER :: LENGTH=4
CHARACTER*(4+LENGTH) LAST, FIRST
```

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

```
SUBROUTINE S1(BUBBLE)
CHARACTER LETTER(26), BUBBLE*(*)
```

In the following example, NAME2 is an automatic object:

```
SUBROUTINE AUTO_NAME(NAME1)
  CHARACTER(LEN = *)          NAME1
  CHARACTER(LEN = LEN(NAME1)) NAME2
```

See Also

- "Type Declaration Statements" for details on the general form and rules for type declaration statements
- "Specification of Data Type" and "Assumed-Length Character Arguments" for details on asterisk length specifications

- Appendix A, "Deleted and Obsolescent Language Features", for details on obsolescent features in Fortran 95

Declaration Statements for Derived Types

The derived-type (TYPE) declaration statement specifies the properties of objects and functions of derived (user-defined) type.

The derived type must be defined before you can specify objects of that type in a TYPE type declaration statement.

An object of derived type must not have the PUBLIC attribute if its type is PRIVATE.

A structure constructor specifies values for derived-type objects.

Examples

The following are examples of derived-type declaration statements:

```
TYPE(EMPLOYEE) CONTRACT
...
TYPE(SETS), DIMENSION(:,:), ALLOCATABLE :: SUBSET_1
```

The following example shows a public type with private components:

```
TYPE LIST_ITEMS
  PRIVATE
  ...
  TYPE(LIST_ITEMS), POINTER :: NEXT, PREVIOUS
END TYPE LIST_ITEMS
```

See Also

- "Derived Data Types"
- "Use and Host Association"
- "PRIVATE and PUBLIC Attributes and Statements"
- "Structure Constructors"
- "Type Declaration Statements" for details on the general form and rules for type declaration statements

Declaration Statements for Arrays

An array declaration (or array declarator) declares the shape of an array. It takes the following form:

(*a-spec*)

a-spec

Is one of the following array specifications:

- "Explicit-Shape Specifications"
- "Assumed-Shape Specifications"
- "Assumed-Size Specifications"
- "Deferred-Shape Specifications"

The array specification can be appended to the name of the array when the array is declared.

Examples

The following examples show array declarations:

```
SUBROUTINE SUB(N, C, D, Z)
   REAL, DIMENSION(N, 15) :: IARRY       ! An explicit-shape array
   REAL C(:), D(0:)                      ! An assumed-shape array
   REAL, POINTER :: B(:,:)               ! A deferred-shape array pointer
   REAL, ALLOCATABLE, DIMENSION(:) :: K  ! A deferred-shape allocatable array
   REAL :: Z(N,*)                        ! An assumed-size array
```

See Also

"Type Declaration Statements" for details on the general form and rules for type declaration statements

Explicit-Shape Specifications

An *explicit-shape array* is declared with explicit values for the bounds in each dimension of the array. An explicit-shape specification takes the following form:

([*dl*:] *du*[, [*dl*:] *du*]...)

dl

Is a specification expression indicating the lower bound of the dimension. The expression can have a positive, negative, or zero value. If necessary, the value is converted to integer type.

If the lower bound is not specified, it is assumed to be 1.

du

Is a specification expression indicating the upper bound of the dimension. The expression can have a positive, negative, or zero value. If necessary, the value is converted to integer type.

The bounds can be specified as constant or nonconstant expressions, as follows:

- If the bounds are constant expressions, the subscript range of the array in a dimension is the set of integer values between and including the lower and upper bounds. If the lower bound is greater than the upper bound, the range is empty, the extent in that dimension is zero, and the array has a size of zero.
- If the bounds are nonconstant expressions, the array must be declared in a procedure. The bounds can have different values each time the procedure is executed, since they are determined when the procedure is entered.

  The bounds are not affected by any redefinition or undefinition of the variables in the specification expression that occurs while the procedure is executing.

  The following explicit-shape arrays can specify nonconstant bounds:

  — An automatic array (the array is a local variable)

  — An adjustable array (the array is a dummy argument to a subprogram)

The following are examples of explicit-shape specifications:

```
INTEGER I(3:8, -2:5)         ! Rank-two array; range of dimension one is
...                         !  3 to 8, range of dimension two is -2 to 5
SUBROUTINE SUB(A, B, C)
  INTEGER :: B, C
  REAL, DIMENSION(B:C) :: A  ! Rank-one array; range is B to C
```

Automatic Arrays

An *automatic array* is an explicit-shape array that is a local variable. Automatic arrays are only allowed in function and subroutine subprograms, and are declared in the specification part of the subprogram. At least one bound of an automatic array must be a nonconstant specification expression. The bounds are determined when the subprogram is called.

The following example shows automatic arrays:

```
SUBROUTINE SUB1 (A, B)
  INTEGER A, B, LOWER
   COMMON /BOUND/ LOWER
  ...
  INTEGER AUTO_ARRAY1(B)
  ...
  INTEGER AUTO_ARRAY2(LOWER:B)
  ..
   INTEGER AUTO_ARRAY3(20, B*A/2)
END SUBROUTINE
```

Adjustable Arrays

An *adjustable array* is an explicit-shape array that is a dummy argument to a subprogram. At least one bound of an adjustable array must be a nonconstant specification expression. The bounds are determined when the subprogram is called.

The array specification can contain integer variables that are either dummy arguments or variables in a common block.

When the subprogram is entered, each dummy argument specified in the bounds must be associated with an actual argument. If the specification includes a variable in a common block, the variable must have a defined value. The array specification is evaluated using the values of the actual arguments, as well as any constants or common block variables that appear in the specification.

The size of the adjustable array must be less than or equal to the size of the array that is its corresponding actual argument.

To avoid possible errors in subscript evaluation, make sure that the bounds expressions used to declare multidimensional adjustable arrays match the bounds as declared by the caller.

In the following example, the function computes the sum of the elements of a rank-two array. Notice how the dummy arguments M and N control the iteration:

```
FUNCTION THE_SUM(A, M, N)
   DIMENSION A(M, N)
   SUMX = 0.0
   DO J = 1, N
     DO I = 1, M
       SUMX = SUMX + A(I, J)
     END DO
   END DO
   THE_SUM = SUMX
END FUNCTION
```

The following are examples of calls on THE_SUM:

```
DIMENSION A1(10,35), A2(3,56)
SUM1 = THE_SUM(A1,10,35)
SUM2 = THE_SUM(A2,3,56)
```

The following example shows how the array bounds determined when the procedure is entered do not change during execution:

```
DIMENSION ARRAY(9,5)
L = 9
M = 5
```

```
CALL SUB(ARRAY,L,M)
END

SUBROUTINE SUB(X,I,J)
  DIMENSION X(-I/2:I/2,J)
  X(I/2,J) = 999
  J = 1
  I = 2
END
```

The assignments to I and J do not affect the declaration of adjustable array X as X(–4:4,5) on entry to subroutine SUB.

See Also

"Specification Expressions"

Assumed-Shape Specifications

An *assumed-shape array* is a dummy argument array that assumes the shape of its associated actual argument array. An assumed-shape specification takes the following form:

([*dl*]:[, [*dl*]:]...)

dl

Is a specification expression indicating the lower bound of the dimension. The expression can have a positive, negative, or zero value. If necessary, the value is converted to integer type.

If the lower bound is not specified, it is assumed to be 1.

The rank of the array is the number of colons (:) specified.

The value of the upper bound is the extent of the corresponding dimension of the associated actual argument array + *lower-bound* – 1.

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

```
INTERFACE
  SUBROUTINE SUB(M)
    INTEGER M(:, 1:, 5:)
   END SUBROUTINE
END INTERFACE
INTEGER L(20, 5:25, 10)
CALL SUB(L)
```

```
SUBROUTINE SUB(M)
  INTEGER M(:, 1:, 5:)
END SUBROUTINE
```

Array M has the same extents as array L, but array M has bounds (1:20, 1:21, 5:14).

Note that an explicit interface is required when calling a routine that expects an assumed-shape or pointer array.

Assumed-Size Specifications

An *assumed-size array* is a dummy argument array that assumes the size (only) of its associated actual argument array; the rank and extents can differ for the actual and dummy arrays. An assumed-size specification takes the following form:

([*expli-shape-spec*,] [*expli-shape-spec*,]... [*dl*:] *)

expli-shape-spec

Is an explicit-shape specification (see "Explicit-Shape Specifications").

dl

Is a specification expression indicating the lower bound of the dimension. The expression can have a positive, negative, or zero value. If necessary, the value is converted to integer type. <p> If the lower bound is not specified, it is assumed to be 1.

*

Is the upper bound of the last dimension.

The rank of the array is the number of explicit-shape specifications plus 1.

The size of the array is assumed from the actual argument associated with the assumed-size dummy array as follows:

- If the actual argument is an array of type other than default character, the size of the dummy array is the size of the actual array.
- If the actual argument is an array element of type other than default character, the size of the dummy array is `a + 1 - s`, where `s` is the subscript order value and `a` is the size of the actual array.
- If the actual argument is a default character array, array element, or array element substring, and it begins at character storage unit *b* of an array with *n* character storage units, the size of the dummy array is as follows:

  ```
  MAX(INT((n + 1 - b)/y), 0)
  ```

  The *y* is the length of an element of the dummy array.

An assumed-size array can only be used as a whole array reference in the following cases:

- When it is an actual argument in a procedure reference that does not require the shape

- In the intrinsic function LBOUND

Because the actual size of an assumed-size array is unknown, an assumed-size array cannot be used as any of the following in an I/O statement:

- An array name in the I/O list
- A unit identifier for an internal file
- A run-time format specifier

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

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

See Also

"Array Elements" for details on array element order

Deferred-Shape Specifications

A *deferred-shape array* is an array pointer or an allocatable array.

The array specification contains a colon (:) for each dimension of the array. No bounds are specified. The bounds (and shape) of allocatable arrays and array pointers are determined when space is allocated for the array during program execution.

An *array pointer* is an array declared with the POINTER attribute. Its bounds and shape are determined when it is associated with a target by pointer assignment, or when the pointer is allocated by execution of an ALLOCATE statement.

In pointer assignment, the lower bound of each dimension of the array pointer is the result of the LBOUND intrinsic function applied to the corresponding dimension of the target. The upper bound of each dimension is the result of the UBOUND intrinsic function applied to the corresponding dimension of the target.

A pointer dummy argument can be associated only with a pointer actual argument. An actual argument that is a pointer can be associated with a nonpointer dummy argument.

A function result can be declared to have the pointer attribute.

An *allocatable array* is declared with the ALLOCATABLE attribute. Its bounds and shape are determined when the array is allocated by execution of an ALLOCATE statement.

The following are examples of deferred-shape specifications:

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

See Also

- "POINTER Attribute and Statement"
- "ALLOCATABLE Attribute and Statement"
- "ALLOCATE Statement"
- "Pointer Assignments"
- "LBOUND"
- "UBOUND"

ALLOCATABLE Attribute and Statement

The ALLOCATABLE attribute specifies that an array is an allocatable array with a deferred shape. The shape of an allocatable array is determined when an ALLOCATE statement is executed, dynamically allocating space for the array.

The ALLOCATABLE attribute can be specified in a type declaration statement or an ALLOCATABLE statement, and takes one of the following forms:

Type Declaration Statement:

type, [*att-ls*,] ALLOCATABLE [, *att-ls*] :: *a*[(*d-spec*)] [, *a*[(*d-spec*)]]...

Statement:

ALLOCATABLE [::] *a*[(*d-spec*)] [, *a*[(*d-spec*)]]...

type

Is a data type specifier.

att-ls

Is an optional list of attribute specifiers.

a

Is the name of the allocatable array; it must not be a dummy argument or function result.

d-spec

Is a deferred-shape specification (: [, :]...). Each colon represents a dimension of the array.

Rules and Behavior

If the array is given the DIMENSION attribute elsewhere in the program, it must be declared as a deferred-shape array.

When the allocatable array is no longer needed, it can be deallocated by execution of a DEALLOCATE statement.

An allocatable array cannot be specified in a COMMON, EQUIVALENCE, DATA, or NAMELIST statement.

Allocatable arrays are not saved by default. If you want to retain the values of an allocatable array across procedure calls, you must specify the SAVE attribute for the array.

Examples

The following example shows a type declaration statement specifying the ALLOCATABLE attribute:

```
REAL, ALLOCATABLE :: Z(:, :, :)
```

The following is an example of the ALLOCATABLE statement:

```
REAL A, B(:) ALLOCATABLE :: A(:,:), B
```

See Also

- "Type Declaration Statements"
- "ALLOCATE Statement"
- "DEALLOCATE Statement"
- "Allocation of Allocatable Arrays" for details on allocation status
- Table 5-1 for details on compatible attributes

AUTOMATIC and STATIC Attributes and Statements

The AUTOMATIC and STATIC attributes control the storage allocation of variables in subprograms.

The AUTOMATIC and STATIC attributes can be specified in a type declaration statement or an AUTOMATIC or STATIC statement, and take one of the following forms:

Type Declaration Statement:

type, [*att-ls*,] AUTOMATIC [, *att-ls*] :: *v* [, *v*]...
type, [*att-ls*,] STATIC [, *att-ls*] :: *v* [, *v*]...

Statement:

AUTOMATIC *v* [, *v*]...
STATIC *v* [, *v*]...

type

Is a data type specifier.

att-ls

Is an optional list of attribute specifiers.

v

Is the name of a variable or an array specification. It can be of any type.

Rules and Behavior

AUTOMATIC and STATIC declarations only affect how data is allocated in storage, as follows:

- A variable declared as AUTOMATIC and allocated in memory resides in the stack storage area.
- A variable declared as STATIC and allocated in memory resides in the static storage area.

If you want to retain definitions of variables upon reentry to subprograms, you must use the SAVE attribute.

Automatic variables can reduce memory use because only the variables currently being used are allocated to memory.

Automatic variables allow possible recursion. With recursion, a subprogram can call itself (directly or indirectly), and resulting values are available upon a subsequent call or return to the subprogram. For recursion to occur, RECURSIVE must be specified in one of the following ways:

- As a keyword in a FUNCTION or SUBROUTINE statement
- As a compiler option
- As an option in an OPTIONS statement

By default, the compiler allocates local variables of non-recursive subprograms, except for allocatable arrays, in the static storage area. The compiler may choose to allocate a variable in temporary (stack or register) storage if it notices that the variable is always defined before use. Appropriate use of the SAVE attribute can prevent compiler warnings if a variable is used before it is defined.

To change the default for variables, specify them as AUTOMATIC or specify RECURSIVE (in one of the ways mentioned above).

To override any compiler option that may affect variables, explicitly specify the variables as AUTOMATIC or STATIC.

NOTE. *Variables that are data-initialized, and variables in* COMMON *and* SAVE *statements are always static. This is regardless of whether a compiler option specifies recursion.*

A variable cannot be specified as AUTOMATIC or STATIC more than once in the same scoping unit.

If the variable is a pointer, AUTOMATIC or STATIC apply only to the pointer itself, not to any associated target.

Some variables cannot be specified as AUTOMATIC or STATIC. The following table shows these restrictions:

Variable	AUTOMATIC	STATIC
Dummy argument	No	No
Automatic object	No	No
Common block item	No	Yes
Use-associated item	No	No
Function result	No	No
Component of a derived type	No	No

A variable can be specified with both the STATIC and SAVE attributes.

If a variable is in a module's outer scope, it can be specified as STATIC, but not as AUTOMATIC.

Examples

The following examples show type declaration statements specifying the AUTOMATIC and STATIC attributes:

```
REAL, AUTOMATIC :: A, B, C
INTEGER, STATIC :: ARRAY_A
```

The following example shows an AUTOMATIC and a STATIC statement:

```
...
CONTAINS
  INTEGER FUNCTION REDO_FUNC
    INTEGER I, J(10), K
    REAL C, D, E(30)
    AUTOMATIC I, J, K(20)
    STATIC C, D, E
   ...
  END FUNCTION
...
```

See Also

- "Type Declaration Statements"
- "OPTIONS Statement"

- "SAVE Attribute and Statement"
- "Functions, Subroutines, and Statement Functions" for details on subprograms
- "Recursive Procedures" for details on specifying recursive subprograms
- Table 5-1 for details on compatible attributes
- "POINTER Attribute and Statement" for details on pointers
- "Modules and Module Procedures" for details on modules
- Your user's guide for details on compiler options

COMMON Statement

A COMMON statement defines one or more contiguous areas, or blocks, of physical storage (called common blocks) that can be accessed by any of the scoping units in an executable program. COMMON statements also define the order in which variables and arrays are stored in each common block, which can prevent misaligned data items.

Common blocks can be named or unnamed (a *blank common*).

The COMMON statement takes the following form:

COMMON [/[*cname*]/] *var-list* [[,] /[*cname*]/ *var-list*]...

cname

Is the name of the common block. The name can be omitted for blank common (//).

var-list

Is a list of variable names, separated by commas.

The variable must not be a dummy argument, allocatable array, automatic object, function, function result, or entry to a procedure. It must not have the PARAMETER attribute. If an object of derived type is specified, it must be a sequence type.

Rules and Behavior

A common block is a global entity, and must not have the same name as any other global entity in the program, such as a subroutine or function.

Any common block name (or blank common) can appear more than once in one or more COMMON statements in a program unit. The list following each successive appearance of the same common block name is treated as a continuation of the list for the block associated with that name.

A variable can appear in only one common block within a scoping unit.

If an array is specified, it can be followed by an explicit-shape array specification, each bound of which must be a constant specification expression. Such an array must not have the POINTER attribute.

A pointer can only be associated with pointers of the same type and kind parameters, and rank.

An object with the TARGET attribute can only be associated with another object with the TARGET attribute and the same type and kind parameters.

A nonpointer can only be associated with another nonpointer, but association depends on their types, as follows:

Type of Variable	Type of Associated Variable
Intrinsic numeric[1] or numeric sequence[2]	Can be of any of these types
Default character or character sequence[2]	Can be of either of these types
Any other intrinsic type	Must have the same type and kind parameters
Any other sequence type	Must have the same type

1. Default integer, default real, double precision real, default complex, double complex, or default logical.
2. If an object of numeric sequence or character sequence type appears in a common block, it is as if the individual components were enumerated directly in the common list.

So, variables can be associated if they are of different numeric type. For example, the following is valid:

```
INTEGER A(20)
REAL Y(20)
COMMON /QUANTA/ A, Y
```

When common blocks from different program units have the same name, they share the same storage area when the units are combined into an executable program.

Entities are assigned storage in common blocks on a one-for-one basis. So, the data type of entities assigned by a COMMON statement in one program unit should agree with the data type of entities placed in a common block by another program unit. For example:

Program Unit A	Program Unit B
COMMON CENTS	INTEGER(2) MONEY
. . .	COMMON MONEY
	. . .

When these program units are combined into an executable program, incorrect results can occur if the 2-byte integer variable MONEY is made to correspond to the lower-addressed two bytes of the real variable CENTS.

Named common blocks must be declared to have the same size in each program unit. Blank common can have different lengths in different program units.

NOTE. *If a common block is initialized by a* DATA *statement, the module containing the initialization must declare the common block to be its maximum defined length. This limitation does not apply if you compile all source modules together.*

Examples

In the following example, the COMMON statement in the main program puts HEAT and X in blank common, and KILO and Q in a named common block, BLK1:

Main Program	Subprogram
COMMON HEAT, X /BLK1/KILO, Q	SUBROUTINE FIGURE
. . .	COMMON /BLK1/LIMA, R / /ALFA,BET
	. . .
CALL FIGURE	RETURN
. . .	END

The COMMON statement in the subroutine makes ALFA and BET share the same storage location as HEAT and X in blank common. It makes LIMA and R share the same storage location as KILO and Q in BLK1.

The following example shows how a COMMON statement can be used to declare arrays:

```
COMMON / MIXED / SPOTTED(100), STRIPED(50,50)
```

See Also

- "Specification Expressions"
- "Storage Association"
- "Derived Data Types"
- "EQUIVALENCE Statement"
- "EQUIVALENCE and COMMON Interaction"
- Your user's guide for details on alignment of data items in common blocks

DATA Statement

The DATA statement assigns initial values to variables before program execution. It takes the following form:

DATA *var-list* /*c-list*/[[,] *var-list* /*c-list*/]...

var-list

Is a list of variables or implied-DO lists, separated by commas.

Subscript expressions and expressions in substring references must be initialization expressions.

An implied-DO list in a DATA statement takes the following form:

(*do-list*, *var* = *expr*1, *expr*2 [, *expr*3])

do-list

Is a list of one or more array elements, substrings, scalar structure components, or implied-DO lists, separated by commas. Any array elements or scalar structure components must not have a constant parent.

var

Is the name of a scalar integer variable (the implied-DO variable).

expr

Are scalar integer expressions. The expressions can contain variables of other implied-DO lists that have this implied-DO list within their ranges.

c-list

Is a list of constants (or names of constants), or for pointer objects, NULL( ); constants must be separated by commas. If the constant is a structure constructor, each component must be an initialization expression. If the constant is in binary, octal, or hexadecimal form, the corresponding object must be of type integer.

A constant can be specified in the form r*constant, where *r* is a repeat specification. It is a nonnegative scalar integer constant (with no kind parameter). If it is a named constant, it must have been declared previously in the scoping unit or made accessible through use or host association. If *r* is omitted, it is assumed to be 1.

Rules and Behavior

A variable can be initialized only once in an executable program. A variable that appears in a DATA statement and is typed implicitly can appear in a subsequent type declaration only if that declaration confirms the implicit typing.

The number of constants in *c-list* must equal the number of variables in *var-list*. The constants are assigned to the variables in the order in which they appear (from left to right).

The following objects cannot be initialized in a DATA statement:

- A dummy argument
- A function
- A function result
- An automatic object
- An allocatable array
- A variable that is accessible by use or host association
- A variable in a named common block (unless the DATA statement is in a block data program unit)
- A variable in blank common

Except for variables in named COMMON blocks, a named variable has the SAVE attribute if any part of it is initialized in a DATA statement. You can confirm this property by specifying the variable in a SAVE statement or a type declaration statement containing the SAVE attribute.

When an unsubscripted array name appears in a DATA statement, values are assigned to every element of that array in the order of subscript progression. The associated constant list must contain enough values to fill the array.

Array element values can be initialized in three ways: by name, by element, or by an implied-DO list (interpreted in the same way as a DO construct).

The following conversion rules and restrictions apply to variable and constant list items:

- If the constant and the variable are both of numeric type, the following conversion occurs:
  - The constant value is converted to the data type of the variable being initialized, if necessary.
  - When a binary, octal, or hexadecimal constant is assigned to a variable or array element, the number of digits that can be assigned depends on the data type of the data item. If the constant contains fewer digits than the capacity of the variable or array element, the constant is extended on the left with zeros. If the constant contains more digits than can be stored, the constant is truncated on the left.
- If the constant and the variable are both of character type, the following conversion occurs:
  - If the length of the constant is less than the length of the variable, the rightmost character positions of the variable are initialized with blank characters.
  - If the length of the constant is greater than the length of the variable, the character constant is truncated on the right.
- If the constant is of numeric type and the variable is of character type, the following restrictions apply:
  - The character variable must have a length of one character.

— The constant must be an integer, binary, octal, or hexadecimal constant, and must have a value in the range 0 through 255.

When the constant and variable conform to these restrictions, the variable is initialized with the character that has the ASCII code specified by the constant. (This lets you initialize a character object to any 8-bit ASCII code.)

- If the constant is a Hollerith or character constant, and the variable is a numeric variable or numeric array element, the number of characters that can be assigned depends on the data type of the data item.

  If the Hollerith or character constant contains fewer characters than the capacity of the variable or array element, the constant is extended on the right with blank characters. If the constant contains more characters than can be stored, the constant is truncated on the right.

Examples

The following example shows the three ways that DATA statements can initialize array element values:

```
DIMENSION A(10,10)
DATA A/100*1.0/    ! initialization by name
DATA A(1,1), A(10,1), A(3,3) /2*2.5, 2.0/ ! initialization by element
DATA ((A(I,J), I=1,5,2), J=1,5) /15*1.0/  ! initialization by implied-DO list
```

The following example shows DATA statements containing structure components:

```
TYPE EMPLOYEE
   INTEGER ID
   CHARACTER(LEN=40) NAME
END TYPE EMPLOYEE
TYPE(EMPLOYEE) MAN_NAME, CON_NAME
DATA MAN_NAME / EMPLOYEE(417, 'Henry Adams') /
DATA CON_NAME%ID, CON_NAME%NAME /891,  "David James "/
```

In the following example, the first DATA statement assigns zero to all 10 elements of array A, and four asterisks followed by two blanks to the character variable STARS:

```
INTEGER A(10), B(10)
CHARACTER BELL, TAB, LF, FF, STARS*6
DATA A,STARS /10*0,'****'/
DATA BELL,TAB,LF,FF /7,9,10,12/
DATA (B(I), I=1,10,2) /5*1/
```

In this case, the second DATA statement assigns ASCII control character codes to the character variables BELL, TAB, LF, and FF. The last DATA statement uses an implied-DO list to assign the value 1 to the odd-numbered elements in the array B.

As a Fortran 95 feature, a pointer can be initialized as disassociated by using a DATA statement. For example:

```
INTEGER, POINTER :: P
DATA P/NULL( )/
END
```

See Also

- "Initialization and Specification Expressions"
- "Type Declaration Statements"
- "I/O Lists" for details on implied-DO lists

DIMENSION Attribute and Statement

The DIMENSION attribute specifies that an object is an array, and defines the shape of the array.

The DIMENSION attribute can be specified in a type declaration statement or a DIMENSION statement, and takes one of the following forms:

Type Declaration Statement:

type, [*att-ls*,] DIMENSION (*a-spec*) [, *att-ls*] :: *a*[(*a-spec*)] [, *a*[(*a-spec*)]]...

Statement:

DIMENSION [::] *a*(*a-spec*) [, *a*(*a-spec*)]...

type

Is a data type specifier.

att-ls

Is an optional list of attribute specifiers.

a-spec

Is an array specification.

In a type declaration statement, any array specification following an array overrides any array specification following DIMENSION.

a

Is the name of the array being declared.

Rules and Behavior

The DIMENSION attribute allocates a number of storage elements to each array named, one storage element to each array element in each dimension. The size of each storage element is determined by the data type of the array.

The total number of storage elements assigned to an array is equal to the number produced by multiplying together the number of elements in each dimension in the array specification. For example, the following statement defines ARRAY as having 16 real elements of 4 bytes each and defines MATRIX as having 125 integer elements of 4 bytes each:

```
DIMENSION ARRAY(4,4), MATRIX(5,5,5)
```

An array can also be declared in the following statements: ALLOCATABLE, POINTER, TARGET, and COMMON.

Examples

The following examples show type declaration statements specifying the DIMENSION attribute:

```
REAL, DIMENSION(10, 10) :: A, B, C(10, 15)  ! Specification following C
                                            !   overrides the one following
                                            !   DIMENSION
REAL, ALLOCATABLE, DIMENSION(:) :: E
```

The following are examples of the DIMENSION statement:

```
DIMENSION BOTTOM(12,24,10)
DIMENSION X(5,5,5), Y(4,85), Z(100)
DIMENSION MARK(4,4,4,4)
SUBROUTINE APROC(A1,A2,N1,N2,N3)
DIMENSION A1(N1:N2), A2(N3:*)

CHARACTER(LEN = 20) D
DIMENSION A(15), B(15, 40), C(-5:8, 7), D(15)
```

See Also

- "Type Declaration Statements"
- "Arrays"
- "ALLOCATABLE Attribute and Statement"
- "COMMON Statement"
- "POINTER Attribute and Statement"
- "TARGET Attribute and Statement"
- "Declaration Statements for Arrays" for details on array specifications
- Table 5-1 for details on compatible attributes

EQUIVALENCE Statement

The EQUIVALENCE statement specifies that a storage area is shared by two or more objects in a program unit. This causes total or partial storage association of the objects that share the storage area.

The EQUIVALENCE statement takes the following form:

EQUIVALENCE (*equiv-list*) [, (*equiv-list*)]...

equiv-list

Is a list of two or more variables, array elements, or substrings, separated by commas (also called an equivalence set). If an object of derived type is specified, it must be a sequence type. Objects cannot have the TARGET attribute.

Each expression in a subscript or a substring reference must be an integer initialization expression. A substring must not have a length of zero.

Rules and Behavior

The following objects cannot be specified in EQUIVALENCE statements:

- A dummy argument
- An allocatable array
- A pointer
- An object of nonsequence derived type
- An object of sequence derived type containing a pointer in the structure
- A function, entry, or result name
- A named constant
- A structure component
- A subobject of any of the above objects

The EQUIVALENCE statement causes all of the entities in one parenthesized list to be allocated storage beginning at the same storage location.

Association of objects depends on their types, as follows:

Type of Object	Type of Associated Object
Intrinsic numeric[1] or numeric sequence	Can be of any of these types
Default character or character sequence	Can be of either of these types[2]
Any other intrinsic type	Must have the same type and kind parameters

Type of Object	Type of Associated Object
Any other sequence type	Must have the same type

1. Default integer, default real, double precision real, default complex, double complex, or default logical.
2. The lengths do not have to be equal.

So, objects can be associated if they are of different numeric type. For example, the following is valid:

```
INTEGER A(20)
REAL Y(20)
EQUIVALENCE(A, Y)
```

Objects of default character do not need to have the same length. The following example associates character variable D with the last 4 (of the 6) characters of character array F:

```
CHARACTER(LEN=4) D
CHARACTER(LEN=3) F(2)
EQUIVALENCE(D, F(1)(3:))
```

Entities having different data types can be associated because multiple components of one data type can share storage with a single component of a higher-ranked data type. For example, if you make an integer variable equivalent to a complex variable, the integer variable shares storage with the real part of the complex variable.

The same storage unit cannot occur more than once in a storage sequence, and consecutive storage units cannot be specified in a way that would make them nonconsecutive.

Examples

The following EQUIVALENCE statement is invalid because it specifies the same storage unit for X(1) and X(2):

```
REAL, DIMENSION(2), :: X
REAL :: Y
EQUIVALENCE(X(1), Y), (X(2), Y)
```

The following EQUIVALENCE statement is invalid because A(1) and A(2) will not be consecutive:

```
REAL A(2)
DOUBLE PRECISION D(2)
EQUIVALENCE(A(1), D(1)), (A(2), D(2))
```

In the following example, the EQUIVALENCE statement causes the four elements of the integer array IARR to share the same storage as that of the double-precision variable DVAR.

```
DOUBLE PRECISION DVAR
```

```
INTEGER(KIND=2) IARR(4)
EQUIVALENCE(DVAR, IARR(1))
```

In the following example, the EQUIVALENCE statement causes the first character of the character variables KEY and STAR to share the same storage location. The character variable STAR is equivalent to the substring KEY(1:10).

```
CHARACTER KEY*16, STAR*10
EQUIVALENCE(KEY, STAR)
```

See Also

- "Initialization Expressions"
- "Derived Data Types"
- "Storage Association" for details on storage units, sequence, and association

Making Arrays Equivalent

When you make an element of one array equivalent to an element of another array, the EQUIVALENCE statement also sets equivalences between the other elements of the two arrays. Thus, if the first elements of two equal-sized arrays are made equivalent, both arrays share the same storage. If the third element of a 7-element array is made equivalent to the first element of another array, the last five elements of the first array overlap the first five elements of the second array.

Two or more elements of the same array should not be associated with each other in one or more EQUIVALENCE statements. For example, you cannot use an EQUIVALENCE statement to associate the first element of one array with the first element of another array, and then attempt to associate the fourth element of the first array with the seventh element of the other array.

Consider the following valid example:

```
DIMENSION TABLE (2,2), TRIPLE (2,2,2)
EQUIVALENCE(TABLE(2,2), TRIPLE(1,2,2))
```

These statements cause the entire array TABLE to share part of the storage allocated to TRIPLE. Table 5-3 shows how these statements align the arrays:

Table 5-3 Equivalence of Array Storage

Array TRIPLE		Array TABLE	
Array Elements	**Element Number**	**Array Element**	**Element Number**
TRIPLE(1,1,1)	1		
TRIPLE(2,1,1)	2		
TRIPLE(1,2,1)	3		

Table 5-3 Equivalence of Array Storage

Array TRIPLE		Array TABLE	
Array Elements	**Element Number**	**Array Element**	**Element Number**
TRIPLE(2,2,1)	4	TABLE(1,1)	1
TRIPLE(1,1,2)	5	TABLE(2,1)	2
TRIPLE(2,1,2)	6	TABLE(1,2)	3
TRIPLE(1,2,2)	7	TABLE(2,2)	4
TRIPLE(2,2,2)	8		

Each of the following statements also aligns the two arrays as shown in Table 5-3:

```
EQUIVALENCE(TABLE, TRIPLE(2,2,1))
EQUIVALENCE(TRIPLE(1,1,2), TABLE(2,1))
```

You can also make arrays equivalent with nonunity lower bounds. For example, an array defined as A(2:3,4) is a sequence of eight values. A reference to A(2,2) refers to the third element in the sequence. To make array A(2:3,4) share storage with array B(2:4,4), you can use the following statement:

```
EQUIVALENCE(A(3,4), B(2,4))
```

The entire array A shares part of the storage allocated to array B. Table 5-4 shows how these statements align the arrays. The arrays can also be aligned by the following statements:

```
EQUIVALENCE(A, B(4,1)) EQUIVALENCE(B(3,2), A(2,2))
```

Table 5-4 Equivalence of Arrays with Nonunity Lower Bounds

Array B		Array A	
Array Element	**Element Number**	**Array Element**	**Element Number**
B(2,1)	1		
B(3,1)	2		
B(4,1)	3	A(2,1)	1
B(2,2)	4	A(3,1)	2
B(3,2)	5	A(2,2)	3
B(4,2)	6	A(3,2)	4
B(2,3)	7	A(2,3)	5
B(3,3)	8	A(3,3)	6
B(4,3)	9	A(2,4)	7
B(2,4)	10	A(3,4)	8
B(3,4)	11		

Table 5-4 Equivalence of Arrays with Nonunity Lower Bounds

Array B		Array A	
Array Element	**Element Number**	**Array Element**	**Element Number**
B(4,4)	12		

Only in the EQUIVALENCE statement can you identify an array element with a single subscript (the linear element number), even though the array was defined as multidimensional. For example, the following statements align the two arrays as shown in Table 5-4:

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

Making Substrings Equivalent

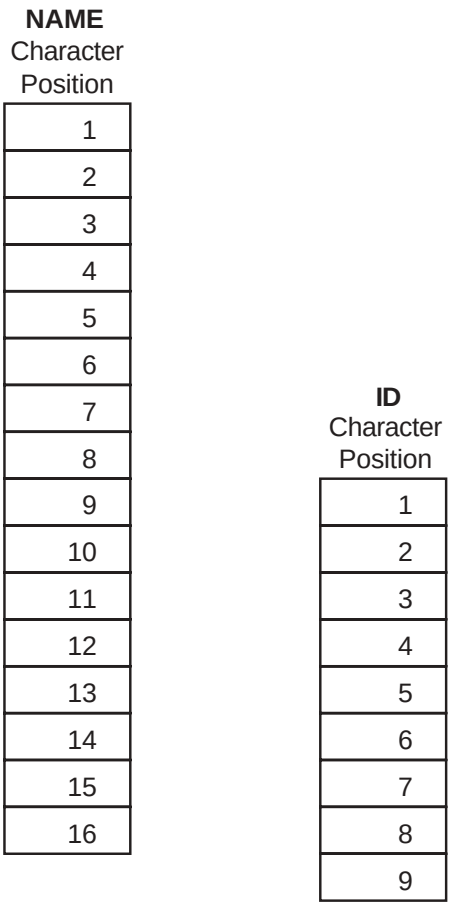
When you make one character substring equivalent to another character substring, the EQUIVALENCE statement also sets associations between the other corresponding characters in the character entities; for example:

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

These statements cause character variables NAME and ID to share space (see Figure 5-1). The arrays can also be aligned by the following statement:

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

Figure 5-1 Equivalence of Substrings



ZK−0618−GE

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

Character elements of arrays can overlap at any character position. For example, the following statements cause character arrays FIELDS and STAR to share storage (see Figure 5-2).

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

The EQUIVALENCE statement cannot assign the same storage location to two or more substrings that start at different character positions in the same character variable or character array. The EQUIVALENCE statement also cannot assign memory locations in a way that is inconsistent with the normal linear storage of character variables and arrays.

EQUIVALENCE and COMMON Interaction

A common block can extend beyond its original boundaries if variables or arrays are associated with entities stored in the common block. However, a common block can only extend beyond its last element; the extended portion cannot precede the first element in the block.

Examples

Figure 5-3 and Figure 5-4 show valid and invalid extensions of the common block, respectively.

Figure 5-2 Equivalence of Character Arrays

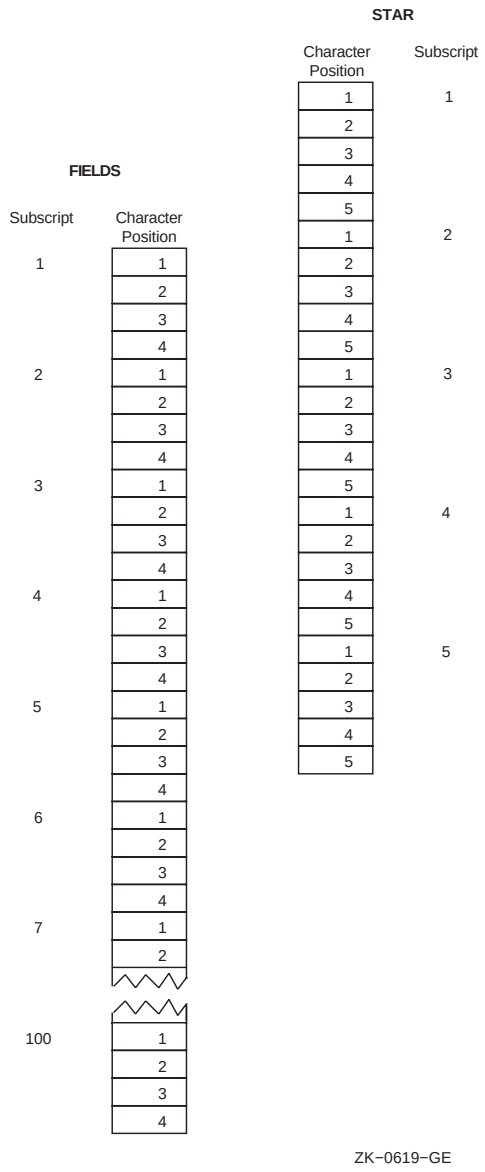
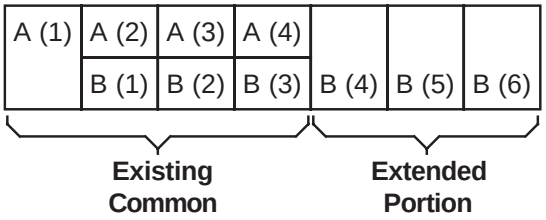


Figure 5-3 A Valid Extension of a Common Block

Valid

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

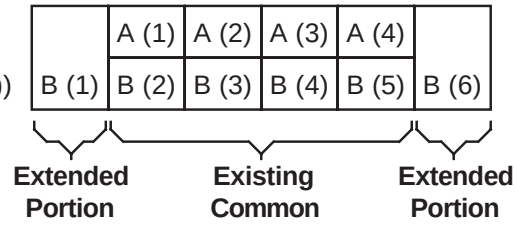


ZK−1944−GE

Figure 5-4 An Invalid Extension of a Common Block

Invalid

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



ZK−1945−GE

The second example is invalid because the extended portion, B(1), precedes the first element of the common block.

The following example shows a valid EQUIVALENCE statement and an invalid EQUIVALENCE statement in the context of a common block.

```
COMMON A, B, C
DIMENSION D(3)
EQUIVALENCE(B, D(1))       ! Valid, because common block is extended
                           !   from the end.
```

```
COMMON A, B, C
DIMENSION D(3)
EQUIVALENCE(B, D(3))      ! Invalid, because D(1) would extend common
                          !   block to precede A's location.
```

EXTERNAL Attribute and Statement

The EXTERNAL attribute allows an external or dummy procedure to be used as an actual argument. (To specify intrinsic procedures as actual arguments, use the INTRINSIC attribute.)

The EXTERNAL attribute can be specified in a type declaration statement or an EXTERNAL statement, and takes one of the following forms:

Type Declaration Statement:

type, [*att-ls*,] EXTERNAL [, *att-ls*] :: *ex-pro* [, *ex-pro*]...

Statement:

EXTERNAL *ex-pro* [, *ex-pro*]...

type

Is a data type specifier.

att-ls

Is an optional list of attribute specifiers.

ex-pro

Is the name of an external (user-supplied) procedure or dummy procedure.

Rules and Behavior

In a type declaration statement, only *functions* can be declared EXTERNAL. However, you can use the EXTERNAL *statement* to declare subroutines and block data program units, as well as functions, to be external.

The name declared EXTERNAL is assumed to be the name of an external procedure, even if the name is the same as that of an intrinsic procedure. For example, if SIN is declared with the EXTERNAL attribute, all subsequent references to SIN are to a user-supplied function named SIN, not to the intrinsic function of the same name.

You can include the name of a block data program unit in the EXTERNAL statement to force a search of the object module libraries for the block data program unit at link time. However, the name of the block data program unit must not be used in a type declaration statement.

Examples

The following example shows type declaration statements specifying the EXTERNAL attribute:

```
PROGRAM TEST
...
INTEGER, EXTERNAL :: BETA
LOGICAL, EXTERNAL :: COS
 ...
CALL SUB(BETA)       ! External function BETA is an actual argument
```

You can use a name specified in an EXTERNAL statement as an actual argument to a subprogram, and the subprogram can then use the corresponding dummy argument in a function reference or a CALL statement; for example:

```
EXTERNAL FACET
CALL BAR(FACET)

SUBROUTINE BAR(F)
EXTERNAL F
CALL F(2)
```

Used as an argument, a complete function reference represents a value, not a subprogram; for example, FUNC(B) represents a value in the following statement:

```
CALL SUBR(A, FUNC(B), C)
```

See Also

- "Type Declaration Statements"
- Chapter 9, "Intrinsic Procedures"
- "INTRINSIC Attribute and Statement"
- Table 5-1 for details on compatible attributes

IMPLICIT Statement

The IMPLICIT statement overrides the default implicit typing rules for names. (The default data type is INTEGER for names beginning with the letters I through N, and REAL for names beginning with any other letter.)

The IMPLICIT statement takes one of the following forms:

IMPLICIT *type* (*a*[, *a*]...)[, *type* (*a*[, *a*]...)]...

IMPLICIT NONE

type

Is a data type specifier (CHARACTER*(*) is not allowed).

a

Is a single letter, a dollar sign ($), or a range of letters in alphabetical order. The form for a range of letters is a_1-a_2, where the second letter follows the first alphabetically (for example, A-C).

The dollar sign can be used at the end of a range of letters, since IMPLICIT interprets the dollar sign to alphabetically follow the letter Z. For example, a range of X–$ would apply to identifiers beginning with the letters X, Y, Z, or $.

Rules and Behavior

The IMPLICIT statement assigns the specified data type (and kind parameter) to all names that have no explicit data type and begin with the specified letter or range of letters. It has no effect on the default types of intrinsic procedures.

When the data type is CHARACTER*len, *len* is the length for character type. The *len* is an unsigned integer constant or an integer initialization expression enclosed in parentheses. The range for *len* is 1 to 2**31–1 on IA-32 processors; 1 to 2**63–1 on Intel Itanium processors.

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

The IMPLICIT NONE statement disables all implicit typing defaults. When IMPLICIT NONE is used, all names in a program unit must be explicitly declared. An IMPLICIT NONE statement must precede any PARAMETER statements, and there must be no other IMPLICIT statements in the scoping unit.

NOTE. *To receive diagnostic messages when variables are used but not declared, you can specify a compiler option instead of using* IMPLICIT NONE.

The following IMPLICIT statement represents the default typing for names when they are not explicitly typed:

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

Examples

The following are examples of the IMPLICIT statement:

```
IMPLICIT DOUBLE PRECISION (D)
IMPLICIT COMPLEX (S,Y), LOGICAL(1) (L,A-C)
IMPLICIT CHARACTER*32 (T-V)
IMPLICIT CHARACTER*2 (W)
```

```
IMPLICIT TYPE(COLORS) (E-F), INTEGER (G-H)
```

See Also

Your user's guide for details on compiler options

INTENT Attribute and Statement

The INTENT attribute specifies the intended use of one or more dummy arguments.

The INTENT attribute can be specified in a type declaration statement or an INTENT statement, and takes one of the following forms:

Type Declaration Statement:

type, [*att-ls*,] INTENT (*intent-spec*) [, *att-ls*] :: *d-arg* [, *d-arg*]...

Statement:

INTENT (*intent-spec*) [::] *d-arg* [, *d-arg*]...

type

Is a data type specifier.

att-ls

Is an optional list of attribute specifiers.

intent-spec

Is one of the following specifiers:

- IN

  Specifies that the dummy argument will be used only to provide data to the procedure. The dummy argument must not be redefined (or become undefined) during execution of the procedure.

  Any associated actual argument must be an expression.
- OUT

  Specifies that the dummy argument will be used to pass data from the procedure back to the calling program. The dummy argument is undefined on entry and must be defined before it is referenced in the procedure.

  Any associated actual argument must be definable.
- INOUT

  Specifies that the dummy argument can both provide data to the procedure and return data to the calling program.

  Any associated actual argument must be definable.

d-arg

Is the name of a dummy argument. It cannot be a dummy procedure or dummy pointer.

Rules and Behavior

The INTENT statement can only appear in the specification part of a subprogram or interface body.

If no INTENT attribute is specified for a dummy argument, its use is subject to the limitations of the associated actual argument.

If a function specifies a defined operator, the dummy arguments must have intent IN.

If a subroutine specifies defined assignment, the first argument must have intent OUT or INOUT, and the second argument must have intent IN.

A dummy argument with intent IN (or a subobject of such a dummy argument) must not appear as any of the following:

- A DO variable or implied-DO variable
- The variable of an assignment statement
- The *pointer-object* of a pointer assignment statement
- An *object* or STAT variable in an ALLOCATE or DEALLOCATE statement
- An input item in a READ statement
- A variable name in a NAMELIST statement if the namelist group name appears in a NML specifier in a READ statement
- An internal file unit in a WRITE statement
- A definable variable in an INQUIRE statement
- An IOSTAT or SIZE specifier in an I/O statement
- An actual argument in a reference to a procedure with an explicit interface if the associated dummy argument has intent OUT or INOUT

If an actual argument is an array section with a vector subscript, it cannot be associated with a dummy array that is defined or redefined (has intent OUT or INOUT).

Examples

The following example shows type declaration statements specifying the INTENT attribute:

```
SUBROUTINE TEST(I, J)
   INTEGER, INTENT(IN) :: I
   INTEGER, INTENT(OUT), DIMENSION(I) :: J
```

The following are examples of the INTENT statement:

```
SUBROUTINE TEST(A, B, X)
```

```
   INTENT(INOUT) :: A, B
   ...
SUBROUTINE CHANGE(FROM, TO)
   USE EMPLOYEE_MODULE
   TYPE(EMPLOYEE) FROM, TO
   INTENT(IN) FROM
   INTENT(OUT) TO
   ...
```

See Also

- "Type Declaration Statements"
- "Argument Association"
- Table 5-1 for details on compatible attributes

INTRINSIC Attribute and Statement

The INTRINSIC attribute allows the specific name of an intrinsic procedure to be used as an actual argument. (Not all specific names can be used as actual arguments. For more information, see Table 9-1.)

The INTRINSIC attribute can be specified in a type declaration statement or an INTRINSIC statement, and takes one of the following forms:

Type Declaration Statement:

type, [*att-ls*,] INTRINSIC [, *att-ls*] :: *in-pro* [, *in-pro*]...

Statement:

INTRINSIC *in-pro* [, *in-pro*]...

type

Is a data type specifier.

att-ls

Is an optional list of attribute specifiers.

in-pro

Is the name of an intrinsic procedure.

Rules and Behavior

In a type declaration statement, only *functions* can be declared INTRINSIC. However, you can use the INTRINSIC *statement* to declare subroutines, as well as functions, to be intrinsic.

The name declared INTRINSIC is assumed to be the name of an intrinsic procedure. If a generic intrinsic function name is given the INTRINSIC attribute, the name retains its generic properties.

Examples

The following example shows a type declaration statement specifying the INTRINSIC attribute:

```
PROGRAM EXAMPLE
...
REAL(8), INTRINSIC :: DACOS
...
CALL TEST(X, DACOS)     ! Intrinsic function DACOS is an actual argument
```

The following example shows an INTRINSIC statement:

Main Program	Subprogram
EXTERNAL CTN	SUBROUTINE TRIG(X,F,Y)
INTRINSIC SIN, COS	Y = F(X)
. . .	RETURN
	END
CALL TRIG(ANGLE,SIN,SINE)	
. . .	FUNCTION CTN(X)
	CTN = COS(X)/SIN(X)
CALL TRIG(ANGLE,COS, COSINE)	RETURN
. . .	END
CALL TRIG(ANGLE,CTN,COTANGENT)	

Note that when TRIG is called with a second argument of SIN or COS, the function reference F(X) references the Fortran 95/90 library functions SIN and COS; but when TRIG is called with a second argument of CTN, F(X) references the user function CTN.

See Also

- "Type Declaration Statements"
- "References to Generic Intrinsic Functions"
- "References to Elemental Intrinsic Procedures"
- Chapter 9, "Intrinsic Procedures", for details on specific intrinsic procedures
- Table 5-1 for details on compatible attributes

NAMELIST Statement

The NAMELIST statement associates a name with a list of variables. This group name can be referenced in some input/output operations.

A NAMELIST statement takes the following form:

NAMELIST /*group*/*var-list* [[,] /*group*/*var-list*]...

group

Is the name of the group.

var-list

Is a list of variables (separated by commas) that are to be associated with the preceding group name. The variables can be of any data type.

Rules and Behavior

The namelist group name is used by namelist I/O statements instead of an I/O list. The unique group name identifies a list whose entities can be modified or transferred.

A variable can appear in more than one namelist group.

Each variable in *var-list* must be accessed by use or host association, or it must have its type, type parameters, and shape explicitly or implicitly specified in the same scoping unit. If the variable is implicitly typed, it can appear in a subsequent type declaration only if that declaration confirms the implicit typing.

The following variables cannot be specified in a namelist group:

- An array dummy argument with nonconstant bounds
- A variable with assumed character length
- An allocatable array
- An automatic object
- A pointer
- A variable of a type that has a pointer as an ultimate component
- A subobject of any of the above objects

Only the variables specified in the namelist can be read or written in namelist I/O. It is not necessary for the input records in a namelist input statement to define every variable in the associated namelist.

The order of variables in the namelist controls the order in which the values appear on namelist output. Input of namelist values can be in any order.

If the group name has the PUBLIC attribute, no item in the variable list can have the PRIVATE attribute.

The group name can be specified in more than one NAMELIST statement in a scoping unit. The variable list following each successive appearance of the group name is treated as a continuation of the list for that group name.

Examples

In the following example, D and E are added to the variables A, B, and C for group name LIST:

```
NAMELIST /LIST/ A, B, C
NAMELIST /LIST/ D, E
```

In the following example, two group names are defined:

```
CHARACTER*30 NAME(25)
NAMELIST /INPUT/ NAME, GRADE, DATE /OUTPUT/ TOTAL, NAME
```

Group name INPUT contains variables NAME, GRADE, and DATE. Group name OUTPUT contains variables TOTAL and NAME.

See Also

- "Rules for Namelist Sequential READ Statements" for details on namelist input
- "Rules for Namelist Sequential WRITE Statements" for details on namelist output

OPTIONAL Attribute and Statement

The OPTIONAL attribute permits dummy arguments to be omitted in a procedure reference.

The OPTIONAL attribute can be specified in a type declaration statement or an OPTIONAL statement, and takes one of the following forms:

Type Declaration Statement:

type, [*att-ls*,] OPTIONAL [, *att-ls*] :: *d-arg* [, *d-arg*]...

Statement:

OPTIONAL [::] *d-arg* [, *d-arg*]...

type

Is a data type specifier.

att-ls

Is an optional list of attribute specifiers.

d-arg

Is the name of a dummy argument.

Rules and Behavior

The OPTIONAL attribute can only appear in the scoping unit of a subprogram or an interface body, and can only be specified for dummy arguments.

A dummy argument is "present " if it associated with an actual argument. A dummy argument that is not optional must be present. You can use the PRESENT intrinsic function to determine whether an optional dummy argument is associated with an actual argument.

To call a procedure that has an optional argument, you must use an explicit interface.

Examples

The following example shows a type declaration statement specifying the OPTIONAL attribute:

```
SUBROUTINE TEST(A)
  REAL, OPTIONAL, DIMENSION(-10:2) :: A
END SUBROUTINE
```

The following is an example of the OPTIONAL statement:

```
SUBROUTINE TEST(A, B, L, X)
  OPTIONAL :: B
  INTEGER A, B, L, X
  IF (PRESENT(B)) THEN        ! Printing of B is conditional
    PRINT *, A, B, L, X       !   on its presence
  ELSE
    PRINT *, A, L, X
  ENDIF
END SUBROUTINE
INTERFACE
  SUBROUTINE TEST(ONE, TWO, THREE, FOUR)
    INTEGER ONE, TWO, THREE, FOUR
    OPTIONAL :: TWO
  END SUBROUTINE
END INTERFACE

INTEGER I, J, K, L
I = 1
J = 2
K = 3
L = 4

CALL TEST(I, J, K, L)            ! Prints:  1  2  3  4
CALL TEST(I, THREE=K, FOUR=L)    ! Prints:  1  3  4 END
```

Note that in the second call to subroutine TEST, the second positional (optional) argument is omitted. In this case, all following arguments must be keyword arguments.

See Also

- "Type Declaration Statements"
- "PRESENT"
- "Optional Arguments"
- Table 5-1 for details on compatible attributes

PARAMETER Attribute and Statement

The PARAMETER attribute defines a named constant.

The PARAMETER attribute can be specified in a type declaration statement or a PARAMETER statement, and takes one of the following forms:

Type Declaration Statement:

type, [*att-ls*,] PARAMETER [, *att-ls*] :: *c* = *expr* [, *c* = *expr*]...

Statement:

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

type

Is a data type specifier.

att-ls

Is an optional list of attribute specifiers.

c

Is the name of the constant.

expr

Is an initialization expression. It can be of any data type.

Rules and Behavior

The type, type parameters, and shape of the named constant are determined in one of the following ways:

- By an explicit type declaration statement in the same scoping unit.
- By the implicit typing rules in effect for the scoping unit. If the named constant is implicitly typed, it can appear in a subsequent type declaration only if that declaration confirms the implicit typing.

For example, consider the following statement:

```
PARAMETER (MU=1.23)
```

According to implicit typing, MU is of integer type, so MU=1. For MU to equal 1.23, it should previously be declared REAL in a type declaration or be declared in an IMPLICIT statement.

A named constant must not appear in a format specification or as the character count for Hollerith constants. For compilation purposes, writing the name is the same as writing the value.

If the named constant is used as the length specifier in a CHARACTER declaration, it must be enclosed in parentheses.

The name of a constant cannot appear as part of another constant, although it can appear as either the real or imaginary part of a complex constant.

You can only use the named constant within the scoping unit containing the defining PARAMETER statement.

Any named constant that appears in the initialization expression must have been defined previously in the same type declaration statement (or in a previous type declaration statement or PARAMETER statement), or made accessible by use or host association.

Examples

The following example shows a type declaration statement specifying the PARAMETER attribute:

```
REAL, PARAMETER :: C = 2.9979251, Y = (4.1 / 3.0)
```

The following is an example of the PARAMETER statement:

```
REAL(4) PI, PIOV2
REAL(8) DPI, DPIOV2
LOGICAL FLAG
CHARACTER*(*) LONGNAME

PARAMETER (PI=3.1415927, DPI=3.141592653589793238D0)
PARAMETER (PIOV2=PI/2, DPIOV2=DPI/2)
PARAMETER (FLAG=.TRUE., LONGNAME='A STRING OF 25 CHARACTERS')
```

See Also

- "Type Declaration Statements"
- "Initialization Expressions"
- "IMPLICIT Statement"
- "Alternative Syntax for the PARAMETER Statement"
- Table 5-1 for details on compatible attributes

POINTER Attribute and Statement

The POINTER attribute specifies that an object is a pointer (a dynamic variable). A pointer does not contain data, but *points* to a scalar or array variable where data is stored. A pointer has no initial storage set aside for it; memory storage is created for the pointer as a program runs.

The POINTER attribute can be specified in a type declaration statement or a POINTER statement, and takes one of the following forms:

Type Declaration Statement:

type, [*att-ls*,] POINTER [, *att-ls*] :: *ptr* [(*d-spec*)] [, *ptr* [(*d-spec*)]]...

Statement:

POINTER [::] *ptr* [(*d-spec*)] [, *ptr* [(*d-spec*)]]...

type

Is a data type specifier.

att-ls

Is an optional list of attribute specifiers.

ptr

Is the name of the pointer. The pointer cannot be declared with the INTENT or PARAMETER attributes.

d-spec

Is a deferred-shape specification (: [,:]...). Each colon represents a dimension of the array.

Rules and Behavior

No storage space is created for a pointer until it is allocated with an ALLOCATE statement or until it is assigned to a allocated target. A pointer must not be referenced or defined until memory is associated with it.

Each pointer has an association status, which tells whether the pointer is currently associated with a target object. When a pointer is initially declared, its status is undefined. You can use the ASSOCIATED intrinsic function to find the association status of a pointer.

If the pointer is an array, and it is given the DIMENSION attribute elsewhere in the program, it must be declared as a deferred-shape array.

A pointer cannot be specified in a DATA, EQUIVALENCE, or NAMELIST statement.

Fortran 95/90 pointers are not the same as integer pointers. For more information, see the "Integer POINTER Statement".

Examples

The following example shows type declaration statements specifying the POINTER attribute:

```
TYPE(SYSTEM), POINTER :: CURRENT, LAST
REAL, DIMENSION(:,:), POINTER :: I, J, REVERSE
```

The following is an example of the POINTER statement:

```
TYPE(SYSTEM) :: TODAYS
POINTER :: TODAYS, A(:,:)
```

See Also

- "Type Declaration Statements"
- "Pointer Assignments"
- "ALLOCATE Statement"
- "Pointer Association"
- "Pointer Arguments"
- "ASSOCIATED"
- "Deferred-Shape Specifications" for details on deferred-shape arrays
- "NULL", which can be used to disassociate a pointer
- Table 5-1 for details on compatible attributes

PRIVATE and PUBLIC Attributes and Statements

The PRIVATE and PUBLIC attributes specify the accessibility of entities in a module. (These attributes are also called accessibility attributes.)

The PRIVATE and PUBLIC attributes can be specified in a type declaration statement or a PRIVATE or PUBLIC statement, and take one of the following forms:

Type Declaration Statement:

type, [*att-ls*,] PRIVATE [, *att-ls*] :: *entity* [, *entity*]...
type, [*att-ls*,] PUBLIC [, *att-ls*] :: *entity* [, *entity*]...

Statement:

PRIVATE [[::] *entity* [, *entity*]...]
PUBLIC [[::] *entity* [, *entity*]...]

type

Is a data type specifier.

att-ls

Is an optional list of attribute specifiers.

entity

Is one of the following:

- A variable name
- A procedure name
- A derived type name
- A named constant
- A namelist group name

In statement form, an entity can also be a generic identifier (a generic name, defined operator, or defined assignment).

Rules and Behavior

The PRIVATE and PUBLIC attributes can only appear in the scoping unit of a module.

Only one PRIVATE or PUBLIC statement without an entity list is permitted in the scoping unit of a module; it sets the default accessibility of all entities in the module.

If no PUBLIC or PRIVATE statements are specified in a module, the default is PUBLIC accessibility. Entities with PUBLIC accessibility can be accessed from outside the module by means of a USE statement.

If a derived type is declared PRIVATE in a module, its components are also PRIVATE. The derived type and its components are accessible to any subprograms within the defining module through host association, but they are not accessible from outside the module.

If the derived type is declared PUBLIC in a module, but its components are declared PRIVATE, any scoping unit accessing the module though use association (or host association) can access the derived-type definition, but not its components.

If a module procedure has a dummy argument or a function result of a type that has PRIVATE accessibility, the module procedure must have PRIVATE accessibility. If the module has a generic identifier, it must also be declared PRIVATE.

If a procedure has a generic identifier, the accessibility of the procedure's specific name is independent of the accessibility of its generic identifier. One can be declared PRIVATE and the other PUBLIC.

Examples

The following examples show type declaration statements specifying the PUBLIC and PRIVATE attributes:

```
REAL,  PRIVATE  :: A, B, C
INTEGER, PUBLIC :: LOCAL_SUMS
```

The following is an example of the PUBLIC and PRIVATE statements:

```
MODULE SOME_DATA
  REAL ALL_B
  PUBLIC ALL_B
  TYPE RESTRICTED_DATA
    REAL LOCAL_C
    DIMENSION LOCAL_C(50)
  END TYPE RESTRICTED_DATA
  PRIVATE RESTRICTED_DATA
END MODULE
```

The following derived-type declaration statement indicates that the type is restricted to the module:

```
TYPE, PRIVATE  :: DATA
  ...
END TYPE DATA
```

The following example shows a PUBLIC type with PRIVATE components:

```
MODULE MATTER
  TYPE ELEMENTS
    PRIVATE
    INTEGER C, D
  END TYPE
...
END MODULE MATTER
```

In this case, components C and D are private to type ELEMENTS, but type ELEMENTS is not private to MODULE MATTER. Any program unit that uses the module MATTER, can declare variables of type ELEMENTS, and pass as arguments values of type ELEMENTS.

See Also

- "Type Declaration Statements"
- "Derived Data Types"
- "USE Statement"
- "Use and Host Association"
- "Defining Generic Names for Procedures" for details on generic identifiers
- "Modules and Module Procedures" for details on modules
- Table 5-1 for details on compatible attributes

SAVE Attribute and Statement

The SAVE attribute causes the values and definition of objects to be retained after execution of a RETURN or END statement in a subprogram.

The SAVE attribute can be specified in a type declaration statement or a SAVE statement, and takes one of the following forms:

Type Declaration Statement:

type, [*att-ls*,] SAVE [, *att-ls*] :: [*object* [, *object*]...]

Statement:

SAVE [*object* [, *object*]...]

type

Is a data type specifier.

att-ls

Is an optional list of attribute specifiers.

object

Is the name of an object, or the name of a common block enclosed in slashes (/*common-block-name*/).

Rules and Behavior

In Intel® Fortran, certain variables are given the SAVE attribute, or not, by default:

- The following variables are saved by default:
  - COMMON variables
  - Local variables of non-recursive subprograms
  - Data initialized by DATA statements
- The following variables are not saved by default:
  - Variables that are declared AUTOMATIC
  - Local variables that are allocatable arrays
  - Derived-type variables that are data initialized by default initialization of any of their components
  - RECORD variables that are data initialized by default initialization specified in its STRUCTURE declaration
- Local variables that are not described in the preceding two lists are saved by default.

To enhance portability and avoid possible compiler warning messages, Intel® recommends that you use the SAVE statement to name variables whose values you want to preserve between subprogram invocations.

When a SAVE statement does not explicitly contain a list, all allowable items in the scoping unit are saved.

A SAVE statement cannot specify the following (their values cannot be saved):

- A blank common
- An object in a common block
- A procedure
- A dummy argument
- A function result
- An automatic object
- A PARAMETER (named) constant

Even though a common block can be included in a SAVE statement, individual variables within the common block can become undefined (or redefined) in another scoping unit.

If a common block is saved in any scoping unit of a program (other than the main program), it must be saved in every scoping unit in which the common block appears.

A SAVE statement has no effect in a main program.

Examples

The following example shows a type declaration statement specifying the SAVE attribute:

```
SUBROUTINE TEST()
  REAL, SAVE :: X, Y
```

The following is an example of the SAVE statement:

```
SAVE A, /BLOCK_B/, C, /BLOCK_D/, E
```

See Also

- "Type Declaration Statements"
- "DATA Statement"
- "COMMON Statement" for details on common blocks
- "Recursive Procedures" for details on recursive program units
- "Modules and Module Procedures" for details on modules.
- Table 5-1 for details on compatible attributes

TARGET Attribute and Statement

The TARGET attribute specifies that an object can become the target of a pointer (it can be pointed to).

The TARGET attribute can be specified in a type declaration statement or a TARGET statement, and takes one of the following forms:

Type Declaration Statement:

type, [*att-ls*,] TARGET [, *att-ls*] :: *object* [(*a-spec*)] [, *object* [(*a-spec*)]]...

Statement:

TARGET [::] *object* [(*a-spec*)] [, *object* [(*a-spec*)]]...

type

Is a data type specifier.

att-ls

Is an optional list of attribute specifiers.

object

Is the name of the object. The object must not be declared with the PARAMETER attribute.

a-spec

Is an array specification.

Rules and Behavior

A pointer is associated with a target by pointer assignment or by an ALLOCATE statement.

If an object does not have the TARGET attribute or has not been allocated (using an ALLOCATE statement), no part of it can be accessed by a pointer.

Examples

The following example shows type declaration statements specifying the TARGET attribute:

```
TYPE(SYSTEM), TARGET :: FIRST
REAL, DIMENSION(20, 20), TARGET :: C, D
```

The following is an example of a TARGET statement:

```
TARGET :: C(50, 50), D
```

See Also

- "Type Declaration Statements"
- "ALLOCATE Statement"
- "Pointer Assignments"
- "Pointer Association"
- Table 5-1 for details on compatible attributes

VOLATILE Attribute and Statement

The VOLATILE attribute specifies that the value of an object is entirely unpredictable, based on information local to the current program unit. It prevents objects from being optimized during compilation.

The VOLATILE attribute can be specified in a type declaration statement or a VOLATILE statement, and takes one of the following forms:

Type Declaration Statement:

type, [*att-ls*,] VOLATILE [, *att-ls*] :: *object* [, *object*]...

Statement:

VOLATILE *object* [, *object*]...

type

Is a data type specifier.

att-ls

Is an optional list of attribute specifiers.

object

Is the name of an object, or the name of a common block enclosed in slashes.

Rules and Behavior

A variable or COMMON block must be declared VOLATILE if it can be read or written in a way that is not visible to the compiler. For example:

- If an operating system feature is used to place a variable in shared memory (so that it can be accessed by other programs), the variable must be declared VOLATILE.
- If a variable is accessed or modified by a routine called by the operating system when an asynchronous event occurs, the variable must be declared VOLATILE.

If an array is declared VOLATILE, each element in the array becomes volatile. If a common block is declared VOLATILE, each variable in the common block becomes volatile.

If an object of derived type is declared VOLATILE, its components become volatile.

If a pointer is declared VOLATILE, the pointer itself becomes volatile.

A VOLATILE statement cannot specify the following:

- A procedure
- A function result
- A namelist group

Examples

The following example shows a type declaration statement specifying the VOLATILE attribute:

```
INTEGER, VOLATILE :: D, E
```

The following example shows a VOLATILE statement:

```
PROGRAM TEST
LOGICAL(1) IPI(4)
INTEGER(4) A, B, C, D, E, ILOOK
INTEGER(4) P1, P2, P3, P4
COMMON /BLK1/A, B, C

VOLATILE /BLK1/, D, E
EQUIVALENCE(ILOOK, IPI)
EQUIVALENCE(A, P1)
EQUIVALENCE(P1, P4)
```

The named common block, BLK1, and the variables D and E are volatile. Variables P1 and P4 become volatile because of the direct equivalence of P1 and the indirect equivalence of P4.

See Also

- "Type Declaration Statements"
- Table 5-1 for details on compatible attributes
- Your user's guide for details on optimizations performed by the compiler

Dynamic Allocation

6

Data objects can be static or dynamic. If a data object is static, a fixed amount of memory storage is created for it at compile time and is not freed until the program exits. If a data object is dynamic, memory storage for the object can be created (allocated), altered, or freed (deallocated) as a program executes.

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

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

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

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

This chapter contains information on the following topics:

- The "ALLOCATE Statement"
- The "DEALLOCATE Statement"
- The "NULLIFY Statement"

See Also

- "Pointer Assignments"
- "Explicit-Shape Specifications" for details on automatic arrays
- "NULL", which can also be used to disassociate a pointer

ALLOCATE Statement

The ALLOCATE statement dynamically creates storage for allocatable arrays and pointer targets. The storage space allocated is uninitialized.

The ALLOCATE statement takes the following form:

ALLOCATE (*object* [(*s-spec*[, *s-spec*...])] [, *object*[(*s-spec*[, *s-spec*...])]]...[, STAT=*sv*])

object

Is the object to be allocated. It is a variable name or structure component, and must be a pointer or allocatable array. The object can be of type character with zero length.

s-spec

Is a shape specification in the form [lower-bound:]upper-bound. Each bound must be a scalar integer expression. The number of shape specifications must be the same as the rank of the *object*.

sv

Is a scalar integer variable in which the status of the allocation is stored.

Rules and Behavior

A bound in *s-spec* must not be an expression containing an array inquiry function whose argument is any allocatable object in the same ALLOCATE statement; for example, the following is not permitted:

```
INTEGER ERR
INTEGER, ALLOCATABLE :: A(:), B(:)
...
ALLOCATE(A(10:25), B(SIZE(A)), STAT=ERR) ! A is invalid as an argument
                                         !  to function SIZE
```

If a STAT variable is specified, it must not be allocated in the ALLOCATE statement in which it appears. If the allocation is successful, the variable is set to zero. If the allocation is not successful, an error condition occurs, and the variable is set to a positive integer value (representing the run-time error). If no STAT variable is specified and an error condition occurs, program execution terminates.

Examples

The following is an example of the ALLOCATE statement:

```
INTEGER J, N, ALLOC_ERR
REAL, ALLOCATABLE :: A(:), B(:,:)
...
ALLOCATE(A(0:80), B(-3:J+1, N), STAT = ALLOC_ERR)
```

See Also

- "ALLOCATABLE Attribute and Statement" for details on allocatable arrays
- "POINTER Attribute and Statement" for details on pointers
- Your user's guide or online documentation for details on run-time error messages

Allocation of Allocatable Arrays

The bounds (and shape) of an allocatable array are determined when it is allocated. Subsequent redefinition or undefinition of any entities in the bound expressions does not affect the array specification.

If the lower bound is greater than the upper bound, that dimension has an extent of zero, and the array has a size of zero. If the lower bound is omitted, it is assumed to be 1.

When an array is allocated, it is definable. If you try to allocate a currently allocated allocatable array, an error occurs.

The intrinsic function ALLOCATED can be used to determine whether an allocatable array is currently allocated; for example:

```
REAL, ALLOCATABLE :: E(:,:)
...
IF (.NOT. ALLOCATED(E)) ALLOCATE(E(2:4,7))
```

Allocation Status

During program execution, the allocation status of an allocatable array is one of the following:

- Not currently allocated

  The array was never allocated or the last operation on it was a deallocation. Such an array must not be referenced or defined.
- Currently allocated

  The array was allocated by an ALLOCATE statement. Such an array can be referenced, defined, or deallocated.

If an allocatable array has the SAVE attribute, it has an initial status of "not currently allocated". If the array is then allocated, its status changes to "currently allocated". It keeps that status until the array is deallocated.

If an allocatable array *does not* have the SAVE attribute, it has the status of "not currently allocated" at the beginning of each invocation of the procedure. If the array's status changes to "currently allocated", it is deallocated if the procedure is terminated by execution of a RETURN or END statement.

Examples

Example 6-1 shows a program that performs virtual memory allocation. This program uses Fortran 95/90 standard-conforming statements instead of calling an operating system memory allocation routine.

Example 6-1 Allocating Virtual Memory

```
! Program accepts an integer and displays square root values

  INTEGER(4) :: N
  READ (5,*) N                          ! Reads an integer value
  CALL MAT(N)
  END

! Subroutine MAT uses the typed integer value to display the square
!  root values of numbers from 1 to N (the number read)

  SUBROUTINE MAT(N)
  REAL(4), ALLOCATABLE :: SQR(:)        ! Declares SQR as a one-dimensional
                                        !  allocatable array
  ALLOCATE (SQR(N))                     ! Allocates array SQR
  DO J=1,N
    SQR(J) = SQRT(FLOATJ(J))            ! FLOATJ converts integer to REAL
  ENDDO

  WRITE (6,*) SQR                       ! Displays calculated values
  DEALLOCATE (SQR)                      ! Deallocates array SQR
  END SUBROUTINE MAT
```

See Also

"ALLOCATED"

Allocation of Pointer Targets

When a pointer is allocated, the pointer is associated with a target and can be used to reference or define the target. (The target can be an array or a scalar, depending on how the pointer was declared.)

Other pointers can become associated with the pointer target (or part of the pointer target) by pointer assignment.

In contrast to allocatable arrays, a pointer can be allocated a new target even if it is currently associated with a target. The previous association is broken and the pointer is then associated with the new target.

If the previous target was created by allocation, it becomes inaccessible unless it can still be referred to by other pointers that are currently associated with it.

The intrinsic function ASSOCIATED can be used to determine whether a pointer is currently associated with a target. (The association status of the pointer must be *defined*.) For example:

```
REAL, TARGET  :: TAR(0:50)
REAL, POINTER :: PTR(:)
PTR => TAR
...
IF (ASSOCIATED(PTR,TAR))...
```

See Also

- "Pointer Assignments"
- "ASSOCIATED"
- "POINTER Attribute and Statement" for details on pointers

DEALLOCATE Statement

The DEALLOCATE statement frees the storage allocated for allocatable arrays and pointer targets (and causes the pointers to become disassociated). It takes the following form:

DEALLOCATE (*object* [, *object*]...[, STAT=*sv*])

object

Is a structure component or the name of a variable, and must be a pointer or allocatable array.

sv

Is a scalar integer variable in which the status of the deallocation is stored.

Rules and Behavior

If a STAT variable is specified, it must not be deallocated in the DEALLOCATE statement in which it appears. If the deallocation is successful, the variable is set to zero. If the deallocation is not successful, an error condition occurs, and the variable is set to a positive integer value (representing the run-time error). If no STAT variable is specified and an error condition occurs, program execution terminates.

It is recommended that all explicitly allocated storage be explicitly deallocated when it is no longer needed.

Examples

The following example shows deallocation of an allocatable array:

```
INTEGER ALLOC_ERR
REAL, ALLOCATABLE :: A(:), B(:,:)
...
ALLOCATE (A(10), B(-2:8,1:5))
...
DEALLOCATE(A, B, STAT = ALLOC_ERR)
```

See Also

Your user's guide or online documentation for details on run-time error messages

Deallocation of Allocatable Arrays

If the DEALLOCATE statement specifies an array that is not currently allocated, an error occurs.

If an allocatable array with the TARGET attribute is deallocated, the association status of any pointer associated with it becomes undefined.

If a RETURN or END statement terminates a procedure, an allocatable array has one of the following allocation statuses:

- It keeps its previous allocation and association status if the following is true:
  - It has the SAVE attribute.
  - It is in the scoping unit of a module that is accessed by another scoping unit which is currently executing.
  - It is accessible by host association.
- It remains allocated if it is accessed by use association.
- Otherwise, its allocation status is deallocated.

The intrinsic function ALLOCATED can be used to determine whether an allocatable array is currently allocated; for example:

```
SUBROUTINE TEST
  REAL, ALLOCATABLE, SAVE :: F(:,:)
  REAL, ALLOCATABLE :: E(:,:,:)
  ...
  IF (.NOT. ALLOCATED(E)) ALLOCATE(E(2:4,7,14))
END SUBROUTINE TEST
```

Note that when subroutine TEST is exited, the allocation status of F is maintained because F has the SAVE attribute. Since E does not have the SAVE attribute, it is deallocated. On the next invocation of TEST, E will have the status of "not currently allocated".

See Also

- "Use and Host Association"
- "TARGET Attribute and Statement"
- "RETURN Statement"
- "END Statement"
- "SAVE Attribute and Statement"

Deallocation of Pointer Targets

A pointer must not be deallocated unless it has a defined association status. If the DEALLOCATE statement specifies a pointer that has undefined association status, or a pointer whose target was not created by allocation, an error occurs.

A pointer must not be deallocated if it is associated with an allocatable array, or it is associated with a portion of an object (such as an array element or an array section).

If a pointer is deallocated, the association status of any other pointer associated with the target (or portion of the target) becomes undefined.

Execution of a RETURN or END statement in a subprogram causes the pointer association status of any pointer declared (or accessed) in the procedure to become undefined, unless any of the following applies to the pointer:

- It has the SAVE attribute.
- It is in the scoping unit of a module that is accessed by another scoping unit which is currently executing.
- It is accessible by host association.
- It is in blank common.
- It is in a named common block that appears in another scoping unit that is currently executing.
- It is the return value of a function declared with the POINTER attribute.

If the association status of a pointer becomes undefined, it cannot subsequently be referenced or defined.

Examples

The following example shows deallocation of a pointer:

```
INTEGER ERR
REAL, POINTER :: PTR_A(:)
...
ALLOCATE (PTR_A(10), STAT=ERR)
...
DEALLOCATE(PTR_A)
```

See Also

- "Use and Host Association"
- "RETURN Statement"
- "END Statement"
- "SAVE Attribute and Statement"
- "POINTER Attribute and Statement" for details on pointers
- "COMMON Statement" for details on common blocks
- "NULL", which can be used to disassociate a pointer

NULLIFY Statement

The NULLIFY statement disassociates a pointer from its target. It takes the following form:

NULLIFY (*pointer-object* [, *pointer-object*]...)

pointer-object

Is a structure component or the name of a variable; it must be a pointer (have the POINTER attribute).

Rules and Behavior

The initial association status of a pointer is undefined. You can use NULLIFY to initialize an undefined pointer, giving it disassociated status. Then the pointer can be tested using the intrinsic function ASSOCIATED.

Examples

The following is an example of the NULLIFY statement:

```
REAL, TARGET  :: TAR(0:50)
REAL, POINTER :: PTR_A(:), PTR_B(:)
PTR_A => TAR
PTR_B => TAR
...
NULLIFY(PTR_A)
```

After these statements are executed, PTR_A will have disassociated status, while PTR_B will continue to be associated with variable TAR.

See Also

- "POINTER Attribute and Statement"
- "Pointer Assignments"
- "ASSOCIATED"
- "NULL", which can be used to disassociate a pointer

Execution Control

7

A program normally executes statements in the order in which they are written. Executable control constructs and statements modify this normal execution by transferring control to another statement in the program, or by selecting blocks (groups) of constructs and statements for execution or repetition.

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

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

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

This chapter contains information on the following topics:

- The "Branch Statements"
- The "CALL Statement"
- The "CASE Constructs"
- The "CONTINUE Statement"
- The "DO Constructs"
- The "END Statement"
- The "IF Construct and Statement"
- The "PAUSE Statement"
- The "RETURN Statement"
- The "STOP Statement"

Branch Statements

Branching affects the normal execution sequence by transferring control to a labeled statement in the same scoping unit. The transfer statement is called the *branch statement*, while the statement to which the transfer is made is called the *branch target statement*.

Any executable statement can be a branch target statement, except for the following:

- CASE statement
- ELSE statement
- ELSE IF statement

Certain restrictions apply to the following statements:

Statement	Restriction
DO terminal statement	The branch must be taken from within its nonblock DO construct.[1]
END DO	The branch must be taken from within its block DO construct.
END IF	The branch should be taken from within its IF construct.[2]
END SELECT	The branch must be taken from within its CASE construct.

1. If the terminal statement is shared by more than one nonblock DO construct, the branch can only be taken from within the innermost DO construct.
2. You can branch to an END IF statement from outside the IF construct; this is a deleted feature in Fortran 95. Intel® Fortran fully supports features deleted in Fortran 95.

The following branch statements are described in this section:

- "Unconditional GO TO Statement"
- "Computed GO TO Statement"
- "The ASSIGN and Assigned GO TO Statements"
- "Arithmetic IF Statement"

See Also

- "IF Construct and Statement"
- "CASE Constructs"
- "DO Constructs"

Unconditional GO TO Statement

The unconditional GO TO statement transfers control to the same branch target statement every time it executes. It takes the following form:

GO TO *label*

label

Is the label of a valid branch target statement in the same scoping unit as the GO TO statement.

The unconditional GO TO statement transfers control to the branch target statement identified by the specified label.

The following are examples of GO TO statements:

```
GO TO 7734
GO TO 99999
```

Computed GO TO Statement

The computed GO TO statement transfers control to one of a set of labeled branch target statements based on the value of an expression. It is an obsolescent feature in Fortran 95.

The computed GO TO statement takes the following form:

GO TO (*label-list*)[,] *expr*

label-list

Is a list of labels (separated by commas) of valid branch target statements in the same scoping unit as the computed GO TO statement. (Also called the transfer list.) The same label can appear more than once in this list.

expr

Is a scalar numeric expression in the range 1 to n, where *n* is the number of statement labels in *label-list*. If necessary, it is converted to integer data type.

Rules and Behavior

When the computed GO TO statement is executed, the expression is evaluated first. The value of the expression represents the ordinal position of a label in the associated list of labels. Control is transferred to the statement identified by the label. For example, if the list contains (30,20,30,40) and the value of the expression is 2, control is transferred to the statement identified with label 20.

If the value of the expression is less than 1 or greater than the number of labels in the list, control is transferred to the next executable statement or construct following the computed GO TO statement.

Examples

The following example shows valid computed GO TO statements:

```
GO TO (12,24,36), INDEX
GO TO (320,330,340,350,360), SITU(J,K) + 1
```

See Also

Appendix A, "Deleted and Obsolescent Language Features", for details on obsolescent features in Fortran 95

The ASSIGN and Assigned GO TO Statements

The ASSIGN statement assigns a label to an integer variable. Subsequently, this variable can be used as a branch target statement by an assigned GO TO statement or as a format specifier in a formatted input/output statement.

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

See Also

Appendix A, "Deleted and Obsolescent Language Features", for details on obsolescent features in Fortran 95 and Fortran 90, as well as features deleted in Fortran 95

ASSIGN Statement

The ASSIGN statement assigns a statement label value to an integer variable. It takes the following form:

ASSIGN *label* TO *var*

label

Is the label of a branch target or FORMAT statement in the same scoping unit as the ASSIGN statement.

var

Is a scalar integer variable.

Rules and Behavior

When an ASSIGN statement is executed, the statement label is assigned to the integer variable. The variable is then undefined as an integer variable and can only be used as a label (unless it is later redefined with an integer value).

The ASSIGN statement must be executed before the statements in which the assigned variable is used.

Examples

The following example shows ASSIGN statements:

```
INTEGER ERROR
```

```
...
ASSIGN 10 TO NSTART
ASSIGN 99999 TO KSTOP
ASSIGN 250 TO ERROR
```

Note that NSTART and KSTOP are integer variables implicitly, but ERROR must be previously declared as an integer variable.

The following statement associates the variable NUMBER with the statement label 100:

```
ASSIGN 100 TO NUMBER
```

If an arithmetic operation is subsequently performed on variable NUMBER (such as follows), the run-time behavior is unpredictable:

```
NUMBER = NUMBER + 1
```

To return NUMBER to the status of an integer variable, you can use the following statement:

```
NUMBER = 10
```

This statement dissociates NUMBER from statement 100 and assigns it an integer value of 10. Once NUMBER is returned to its integer variable status, it can no longer be used in an assigned GO TO statement.

Assigned GO TO Statement

The assigned GO TO statement transfers control to the statement whose label was most recently assigned to a variable. The assigned GO TO statement takes the following form:

GO TO *var* [[,] (*label-list*)]

var

Is a scalar integer variable.

label-list

Is a list of labels (separated by commas) of valid branch target statements in the same scoping unit as the assigned GO TO statement. The same label can appear more than once in this list.

Rules and Behavior

The variable must have a statement label value assigned to it by an ASSIGN statement (not an arithmetic assignment statement) before the GO TO statement is executed.

If a list of labels appears, the statement label assigned to the variable must be one of the labels in the list.

Both the assigned GO TO statement and its associated ASSIGN statement must be in the same scoping unit.

Examples

The following example is equivalent to `GO TO 200`:

```
ASSIGN 200 TO IGO
GO TO IGO
```

The following example is equivalent to `GO TO 450`:

```
ASSIGN 450 TO IBEG
GO TO IBEG, (300,450,1000,25)
```

The following example shows an invalid use of an assigned variable:

```
ASSIGN 10 TO I
J = I
GO TO J
```

In this case, variable J is not the variable assigned to, so it cannot be used in the assigned GO TO statement.

Arithmetic IF Statement

The arithmetic IF statement conditionally transfers control to one of three statements, based on the value of an arithmetic expression. It is an obsolescent feature in Fortran 95 and Fortran 90.

The arithmetic IF statement takes the following form:

IF (*expr*) *label1*, *label2*, *label3*

expr

Is a scalar numeric expression of type integer or real (enclosed in parentheses).

label1, *label2*, *label3*

Are the labels of valid branch target statements that are in the same scoping unit as the arithmetic IF statement.

Rules and Behavior

All three labels are required, but they do not need to refer to three different statements. The same label can appear more than once in the same arithmetic IF statement.

During execution, the expression is evaluated first. Depending on the value of the expression, control is then transferred as follows:

If the Value of *expr* is:	Control Transfers To:
Less than 0	Statement *label1*
Equal to 0	Statement *label2*

If the Value of *expr* is:	Control Transfers To:
Greater than 0	Statement *label3*

Examples

The following example transfers control to statement 50 if the real variable THETA is less than or equal to the real variable CHI. Control passes to statement 100 only if THETA is greater than CHI.

```
IF (THETA-CHI) 50,50,100
```

The following example transfers control to statement 40 if the value of the integer variable NUMBER is even. It transfers control to statement 20 if the value is odd.

```
IF (NUMBER / 2*2 - NUMBER) 20,40,20
```

See Also

Appendix A, "Deleted and Obsolescent Language Features", for details on obsolescent features in Fortran 95 and Fortran 90

CALL Statement

The CALL statement transfers control to a subroutine subprogram. It takes the following form:

CALL *sub* [([*a-arg* [, *a-arg*]...])]

sub

Is the name of the subroutine subprogram or other external procedure, or a dummy argument associated with a subroutine subprogram or other external procedure.

a-arg

Is an actual argument optionally preceded by [keyword=], where *keyword* is the name of a dummy argument in the explicit interface for the subroutine. The keyword is assigned a value when the procedure is invoked.

Each actual argument must be a variable, an expression, the name of a procedure, or an alternate return specifier. (It must not be the name of an internal procedure, statement function, or the generic name of a procedure.)

An alternate return specifier is an asterisk (*), or ampersand (&), followed by the label of an executable branch target statement in the same scoping unit as the CALL statement. (An alternate return is an obsolescent feature in Fortran 95 and Fortran 90.)

Rules and Behavior

When the CALL statement is executed, any expressions in the actual argument list are evaluated, then control is passed to the first executable statement or construct in the subroutine. When the subroutine finishes executing, control returns to the next executable statement following the CALL statement, or to a statement identified by an alternate return label (if any).

If an argument list appears, each actual argument is associated with the corresponding dummy argument by its position in the argument list or by the name of its keyword. The arguments must agree in type and kind parameters.

If positional arguments and argument keywords are specified, the argument keywords must appear last in the actual argument list.

If a dummy argument is optional, the actual argument can be omitted.

An actual argument associated with a dummy procedure must be the specific name of a procedure, or be another dummy procedure. Certain specific intrinsic function names must not be used as actual arguments (see Table 9-1).

You can use a CALL statement to invoke a function as long as the function is not one of the following types:

- REAL(8)
- REAL(16)
- COMPLEX(8)
- COMPLEX(16)
- CHARACTER

Examples

The following example shows valid CALL statements:

```
CALL CURVE(BASE,3.14159+X,Y,LIMIT,R(LT+2))

CALL PNTOUT(A,N,'ABCD')

CALL EXIT

CALL MULT(A,B,*10,*20,C)       ! The asterisks and ampersands denote
CALL SUBA(X,&30,&50,Y)         !  alternate returns
```

The following example shows a subroutine with argument keywords:

```
PROGRAM KEYWORD_EXAMPLE
  INTERFACE
    SUBROUTINE TEST_C(I, L, J, KYWD2, D, F, KYWD1)
      INTEGER I, L(20), J, KYWD1
```

```
      REAL, OPTIONAL :: D, F
      COMPLEX KYWD2
      ...
    END SUBROUTINE TEST_C
  END INTERFACE
INTEGER I, J, K
INTEGER L(20)
COMPLEX Z1
CALL TEST_C(I, L, J, KYWD1 = K, KYWD2 = Z1)
...
```

The first three actual arguments are associated with their corresponding dummy arguments by position. The argument keywords are associated by keyword name, so they can appear in any order.

Note that the interface to subroutine TEST has two optional arguments that have been omitted in the CALL statement.

The following is another example of a subroutine call with argument keywords:

```
CALL TEST(X, Y, N, EQUALITIES = Q, XSTART = X0)
```

The first three arguments are associated by position.

See Also

- "Subroutines"
- "Argument Association" for details on procedure arguments
- "OPTIONAL Attribute and Statement" for details on optional arguments
- "Dummy Procedure Arguments" for details on dummy arguments
- Appendix A, "Deleted and Obsolescent Language Features", for details on obsolescent features in Fortran 95 and Fortran 90

CASE Constructs

The CASE construct conditionally executes one block of constructs or statements depending on the value of a scalar expression in a SELECT CASE statement.

The CASE construct takes the following form:

[*name*:] SELECT CASE (*expr*)
[CASE (*case-value* [, *case-value*]...) [*name*]
block]...
[CASE DEFAULT [*name*]
block]
END SELECT [*name*]

name

Is the name of the CASE construct.

expr

Is a scalar expression of type integer, logical, or character (enclosed in parentheses). Evaluation of this expression results in a value called the *case index*.

case-value

Is one or more scalar integer, logical, or character initialization expressions enclosed in parentheses. Each *expr* must be of the same type and kind parameter as *expr*. If the type is character, *case-value* and *expr* can be of different lengths, but their kind parameter must be the same.

Integer and character expressions can be expressed as a range of case values, taking one of the following forms:

```
low:high
low:
:high
```

Case values must not overlap.

block

Is a sequence of zero or more statements or constructs.

Rules and Behavior

If a construct name is specified in a SELECT CASE statement, the same name must appear in the corresponding END SELECT statement. The same construct name can optionally appear in any CASE statement in the construct. The same construct name must not be used for different named constructs in the same scoping unit.

The case expression (*expr*) is evaluated first. The resulting case index is compared to the case values to find a matching value (there can only be one). When a match occurs, the block following the matching case value is executed and the construct terminates.

The following rules determine whether a match occurs:

- When the case value is a single value (no colon appears), a match occurs as follows:

Data Type	A Match Occurs If:
Logical	case-index .EQV. case- value
Integer or character	case-index = = case-value

- When the case value is a range of values (a colon appears), a match depends on the range specified, as follows:

Range	A Match Occurs If:
low:	case-index >= low
:high	case-index <= high
low:high	low <= case-index <= high

The following are all valid case values:

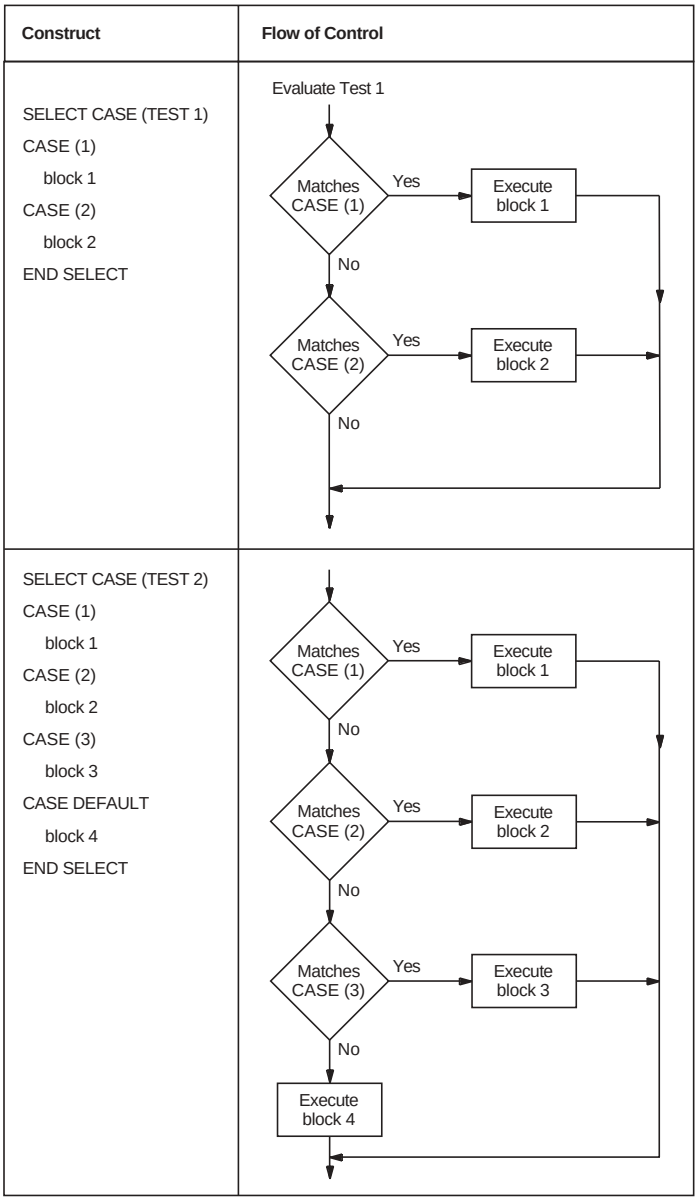
```
CASE (1, 4, 7, 11:14, 22)      ! Individual values as specified:
                               !     1, 4, 7, 11, 12, 13, 14, 22
CASE (:-1)                     ! All values less than zero
CASE (0)                       ! Only zero
CASE (1:)                      ! All values above zero
```

If no match occurs but a CASE DEFAULT statement is present, the block following that statement is executed and the construct terminates.

If no match occurs and no CASE DEFAULT statement is present, no block is executed, the construct terminates, and control passes to the next executable statement or construct following the END SELECT statement.

Figure 7-1 shows the flow of control in a CASE construct.

Figure 7-1 Flow of Control in CASE Constructs



ZK-6515A-GE

You cannot use branching statements to transfer control to a CASE statement. However, branching to a SELECT CASE statement is allowed. Branching to the END SELECT statement is allowed only from within the CASE construct.

Examples

The following are examples of CASE constructs:

```
INTEGER FUNCTION STATUS_CODE (I)
   INTEGER I
   CHECK_STATUS: SELECT CASE (I)
   CASE (:-1)
     STATUS_CODE = -1
   CASE (0)
     STATUS_CODE = 0
   CASE (1:)
     STATUS_CODE = 1
   END SELECT CHECK_STATUS
END FUNCTION STATUS_CODE
```

```
SELECT CASE (J)
CASE (1, 3:7, 9)    ! Values: 1, 3, 4, 5, 6, 7, 9
  CALL SUB_A
CASE DEFAULT
  CALL SUB_B
END SELECT
```

The following three examples are equivalent:

```
1. SELECT CASE (ITEST .EQ. 1)
   CASE (.TRUE.)
      CALL SUB1 ()
   CASE (.FALSE.)
      CALL SUB2 ()
   END SELECT
2. SELECT CASE (ITEST)
   CASE DEFAULT
     CALL SUB2 ()
   CASE (1)
     CALL SUB1 ()
   END SELECT
```

```
3. IF (ITEST .EQ. 1) THEN
    CALL SUB1 ()
   ELSE
    CALL SUB2 ()
   END IF
```

CONTINUE Statement

The CONTINUE statement is primarily used to terminate a labeled DO construct when the construct would otherwise end improperly with either a GO TO, arithmetic IF, or other prohibited control statement.

The CONTINUE statement takes the following form:

CONTINUE

The statement by itself does nothing and has no effect on program results or execution sequence.

The following example shows a CONTINUE statement:

```
    DO 150 I = 1,40
40  Y = Y + 1
    Z = COS(Y)
    PRINT *, Z
    IF (Y .LT. 30) GO TO 150
    GO TO 40
150 CONTINUE
```

DO Constructs

The DO construct controls the repeated execution of a block of statements or constructs. (This repeated execution is called a *loop*.)

The number of iterations of a loop can be specified in the initial DO statement in the construct, or the number of iterations can be left indefinite by a simple DO ("DO forever") construct or DO WHILE statement.

The EXIT and CYCLE statements modify the execution of a loop. An EXIT statement terminates execution of a loop, while a CYCLE statement terminates execution of the current iteration of a loop. For example:

```
DO
  READ (EUNIT, IOSTAT=IOS) Y
  IF (IOS /= 0) EXIT
  IF (Y <0) CYCLE
```

```
  CALL SUB_A(Y)
END DO
```

If an error or end-of-file occurs, the DO construct terminates. If a negative value for Y is read, the program skips to the next READ statement.

See Also

- "CYCLE Statement"
- "EXIT Statement"
- "FORALL Statement and Construct" for details on DO loops in FORALL constructs

Forms for DO Constructs

A DO construct takes one of the following forms:

Block Form:

[*name*:] DO [*label*][,] [*loop-control*]
  block
[*label*] *term-stmt*

Nonblock Form:

DO *label*[,] [*loop-control*]
  block
[*label*] *ex-term-stmt*

name

Is the name of the DO construct.

label

Is a statement label identifying the terminal statement.

loop-control

Is a DO iteration (see "Iteration Loop Control") or a (DO) WHILE statement (see "DO WHILE Statement").

block

Is a sequence of zero or more statements or constructs.

term-stmt

Is the terminal statement for the block form of the construct.

ex-term-stmt

Is the terminal statement for the nonblock form of the construct.

Rules and Behavior

The terminal statement (*term-stmt*) for a block DO construct is an END DO or CONTINUE statement. If the block DO statement contains a label, the terminal statement must be identified with the same label. If no label appears, the terminal statement must be an END DO statement.

If a construct name is specified in a block DO statement, the same name must appear in the terminal END DO statement. If no construct name is specified in the block DO statement, no name can appear in the terminal END DO statement.

The terminal statement (*ex-term-stmt*) for a nonblock DO construct is an executable statement (or construct) that is identified by the label specified in the nonblock DO statement. A nonblock DO construct can share a terminal statement with another nonblock DO construct. A block DO construct cannot share a terminal statement.

The following cannot be terminal statements for nonblock DO constructs:

- CONTINUE (allowed if it is a shared terminal statement)
- CYCLE
- END (for a program or subprogram)
- EXIT
- GO TO (unconditional or assigned)
- Arithmetic IF
- RETURN
- STOP

The nonblock DO construct is an obsolescent feature in Fortran 95 and Fortran 90.

Examples

The following example shows equivalent block DO and nonblock DO constructs:

```
   DO I = 1, N                 ! Block DO
     TOTAL = TOTAL + B(I)
   END DO

   DO 20 I = 1, N              ! Nonblock DO
20 TOTAL = TOTAL + B(I)
```

The following example shows a simple block DO construct (contains no iteration count or DO WHILE statement):

```
DO
  READ *, N
  IF (N == 0) STOP
  CALL SUBN
```

```
END DO
```

The DO block executes repeatedly until the value of zero is read. Then the DO construct terminates.

The following example shows a named block DO construct:

```
LOOP_1: DO I = 1, N
          A(I) = C * B(I)
        END DO LOOP_1
```

The following example shows a nonblock DO construct with a shared terminal statement:

```
    DO 20 I = 1, N
    DO 20 J = 1 + I, N
 20 RESULT(I,J) = 1.0 / REAL(I + J)
```

See Also

Appendix A, "Deleted and Obsolescent Language Features", for details on obsolescent features in Fortran 95 and Fortran 90

Execution of DO Constructs

The range of a DO construct includes all the statements and constructs that follow the DO statement, up to and including the terminal statement. If the DO construct contains another construct, the inner (nested) construct must be entirely contained within the DO construct.

Execution of a DO construct differs depending on how the loop is controlled, as follows:

- For simple DO constructs, there is no loop control. Statements in the DO range are repeated until the DO statement is terminated explicitly by a statement within the range.
- For iterative DO statements, loop control is specified as `do-var = expr1, expr2 [, expr3]`. An iteration count specifies the number of times the DO range is executed. (For more information on iteration loop control, see "Iteration Loop Control".)
- For DO WHILE statements, loop control is specified as a DO range. The DO range is repeated as long as a specified condition remains true. Once the condition is evaluated as false, the DO construct terminates. (For more information on the DO WHILE statement, see "DO WHILE Statement".)

Iteration Loop Control

DO iteration loop control takes the following form:

do-var = *expr*1, *expr*2 [, *expr*3]

do-var

Is the name of a scalar variable of type integer or real. It cannot be the name of an array element or structure component.

expr

Is a scalar numeric expression of type integer or real. If it is not the same type as *do-var*, it is converted to that type.

Rules and Behavior

A DO variable or expression of type real is a deleted feature in Fortran 95; it was obsolescent in Fortran 90. Intel Fortran fully supports features deleted in Fortran 95.

The following steps are performed in iteration loop control:

1. The expressions *expr*1, *expr*2, and *expr*3 are evaluated to respectively determine the initial, terminal, and increment parameters.

   The increment parameter (*expr*3) is optional and must not be zero. If an increment parameter is not specified, it is assumed to be of type default integer with a value of 1.
2. The DO variable (*do-var*) becomes defined with the value of the initial parameter (*expr*1).
3. The iteration count is determined as follows:

   ```
   MAX(INT((expr2 - expr1 + expr3)/expr3), 0)
   ```

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

   ```
   expr1 > expr2 and expr3 > 0
   expr1 < expr2 and expr3 < 0
   ```
4. The iteration count is tested. If the iteration count is zero, the loop terminates and the DO construct becomes inactive. (A compiler option can affect this, see your user's guide for more information.) If the iteration count is nonzero, the range of the loop is executed.
5. The iteration count is decremented by one, and the DO variable is incremented by the value of the increment parameter, if any.

After termination, the DO variable retains its last value (the one it had when the iteration count was tested and found to be zero).

The DO variable must not be redefined or become undefined during execution of the DO range.

If you change variables in the initial, terminal, or increment expressions during execution of the DO construct, it does not affect the iteration count. The iteration count is fixed each time the DO construct is entered.

Examples

The following example specifies 25 iterations:

```
DO 100 K=1,50,2
```

K = 49 during the final iteration, K = 51 after the loop.

The following example specifies 27 iterations:

```
DO 350 J=50,-2,-2
```

J = –2 during the final iteration, J = –4 after the loop.

The following example specifies 9 iterations:

```
DO NUMBER=5,40,4
```

NUMBER = 37 during the final iteration, NUMBER = 41 after the loop. The terminating statement of this DO loop must be END DO.

See Also

Appendix A, "Deleted and Obsolescent Language Features", for details on obsolescent features in Fortran 95 and Fortran 90, as well as features deleted in Fortran 95

Nested DO Constructs

A DO construct can contain one or more complete DO constructs (loops). The range of an inner nested DO construct must lie completely within the range of the next outer DO construct. Nested nonblock DO constructs can share a labeled terminal statement.

Figure 7-2 shows correctly and incorrectly nested DO constructs.

Figure 7-2 Nested DO Constructs

Correctly Nested DO Loops	Incorrectly Nested DO loops
DO 45 K=1,10	DO 15 K=1,10
.	.
.	.
DO 35 L=2,50,2	DO 25 L=1,20
.	.
.	.
35 CONTINUE	15 CONTINUE
.	.
.	.
DO 45 M=1,20	DO 30 M=1,15
.	.
.	.
45 CONTINUE	25 CONTINUE
.	.
.	.
DO 10 I=1,20	30 CONTINUE
.	.
.	.
DO J=1,5	DO 10 I=1,5
.	.
.	.
DO K=1,10	DO J=1,10
.	.
.	.
END DO	10 CONTINUE
.	.
.	.
END DO	END DO
.	
.	
10 CONTINUE	

ZK−7969−GE

In a nested DO construct, you can transfer control from an inner construct to an outer construct. However, you cannot transfer control from an outer construct to an inner construct.

If two or more nested DO constructs share the same terminal statement, you can transfer control to that statement only from within the range of the innermost construct. Any other transfer to that statement constitutes a transfer from an outer construct to an inner construct, because the shared statement is part of the range of the innermost construct.

Extended Range

A DO construct has an extended range if both of the following are true:

- The DO construct contains a control statement that transfers control out of the construct.
- Another control statement returns control back into the construct after execution of one or more statements.

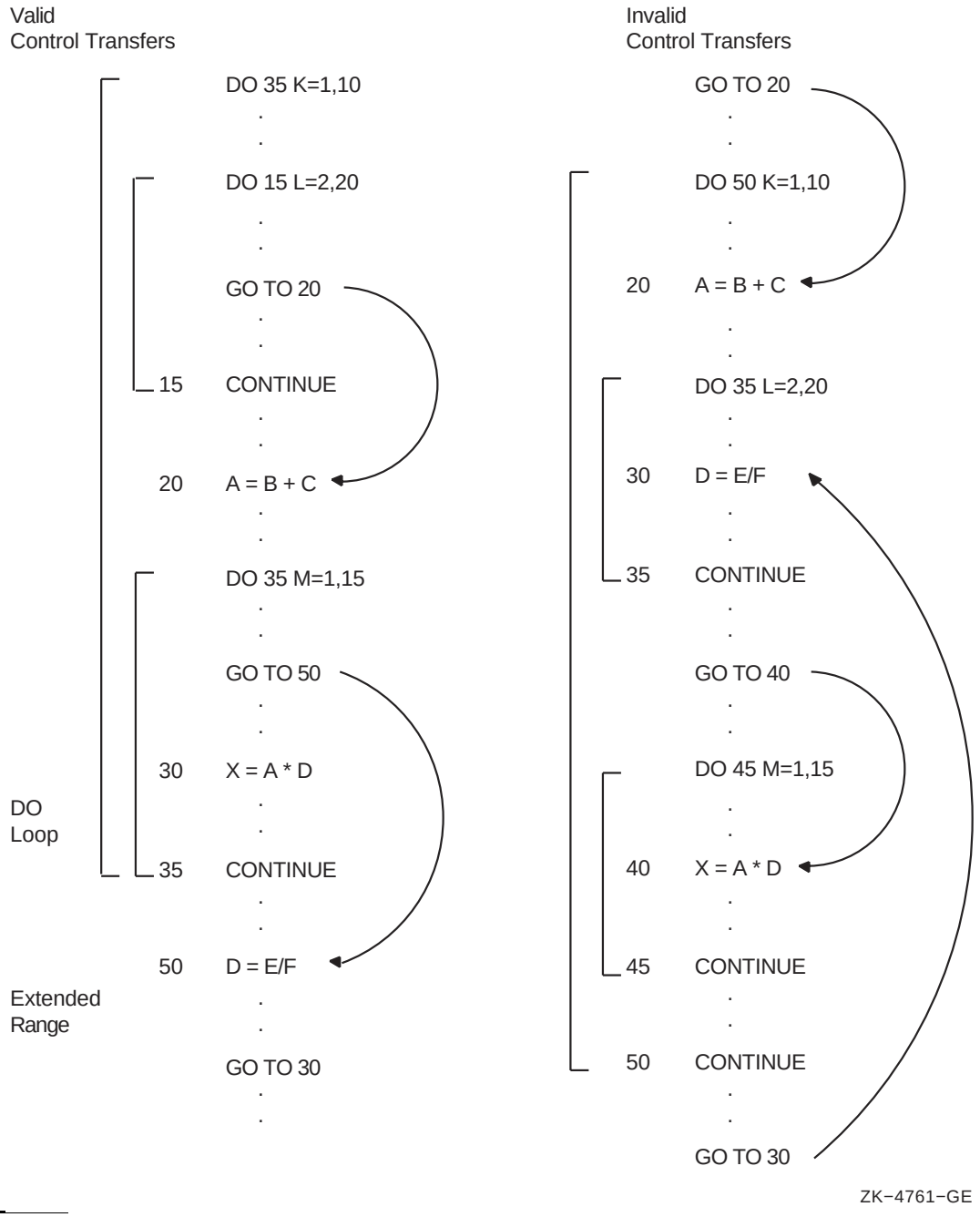
The range of the construct is extended to include all executable statements between the destination statement of the first transfer and the statement that returns control to the construct.

The following rules apply to a DO construct with extended range:

- A transfer into the range of a DO statement is permitted only if the transfer is made from the extended range of that DO statement.
- The extended range of a DO statement must not change the control variable of the DO statement.

Figure 7-3 illustrates valid and invalid extended range control transfers.

Figure 7-3 Control Transfers and Extended Range



DO WHILE Statement

The DO WHILE statement executes the range of a DO construct while a specified condition remains true. The statement takes the following form:

DO [*label*][,] WHILE (*expr*)

label

Is a label specifying an executable statement in the same program unit.

expr

Is a scalar logical expression enclosed in parentheses.

Rules and Behavior

Before each execution of the DO range, the logical expression is evaluated. If it is true, the statements in the body of the loop are executed. If it is false, the DO construct terminates and control transfers to the statement following the loop.

If no label appears in a DO WHILE statement, the DO WHILE loop must be terminated with an END DO statement.

You can transfer control out of a DO WHILE loop but not into a loop from elsewhere in the program.

Examples

The following example shows a DO WHILE statement:

```
CHARACTER*132 LINE
...
I = 1
DO WHILE (LINE(I:I) .EQ. ' ')
   I = I + 1
END DO
```

The following examples show required and optional END DO statements:

```
Required                  Optional
DO WHILE (I .GT. J)          DO 10 WHILE (I .GT. J)
  ARRAY(I,J) = 1.0             ARRAY(I,J) = 1.0
  I = I - 1                    I = I - 1
END DO                    10 END DO
```

CYCLE Statement

The CYCLE statement interrupts the current execution cycle of the innermost (or named) DO construct.

The CYCLE statement takes the following form:

CYCLE [*name*]

name

Is the name of the DO construct.

Rules and Behavior

When a CYCLE statement is executed, the following occurs:

1. The current execution cycle of the named (or innermost) DO construct is terminated.

   If a DO construct name is specified, the CYCLE statement must be within the range of that construct.
2. The iteration count (if any) is decremented by 1.
3. The DO variable (if any) is incremented by the value of the increment parameter (if any).
4. A new iteration cycle of the DO construct begins.

Any executable statements following the CYCLE statement (including a labeled terminal statement) are not executed.

A CYCLE statement can be labeled, but it cannot be used to terminate a DO construct.

Examples

The following example shows a CYCLE statement:

```
DO I =1, 10
  A(I) = C + D(I)
  IF (D(I) < 0) CYCLE    ! If true, the next statement is omitted
  A(I) = 0               !  from the loop and the loop is tested again.
END DO
```

EXIT Statement

The EXIT statement terminates execution of a DO construct. It takes the following form:

EXIT [*name*]

name

Is the name of the DO construct.

Rules and Behavior

The EXIT statement causes execution of the named (or innermost) DO construct to be terminated.

If a DO construct name is specified, the EXIT statement must be within the range of that construct.

Any DO variable present retains its last defined value.

An EXIT statement can be labeled, but it cannot be used to terminate a DO construct.

Examples

The following example shows an EXIT statement:

```
LOOP_A : DO I = 1, 15
  N = N + 1
  IF (N > I) EXIT LOOP_A
END DO LOOP_A
```

END Statement

The END statement marks the end of a program unit. It takes one of the following forms:

END [PROGRAM [*program-name*]]
END [FUNCTION [*function-name*]]
END [SUBROUTINE [*subroutine-name*]]
END [MODULE [*module-name*]]
END [BLOCK DATA [*block-data-name*]]

For internal procedures and module procedures, you must specify the FUNCTION and SUBROUTINE keywords in the END statement; otherwise, the keywords are optional.

In main programs, function subprograms, and subroutine subprograms, END statements are executable and can be branch target statements. If control reaches the END statement in these program units, the following occurs:

- In a main program, execution of the program terminates.
- In a function or subroutine subprogram, a RETURN statement is implicitly executed.

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

The END statements in a module or block data program unit are nonexecutable.

See Also

- Chapter 8, “Program Units and Procedures”
- “Branch Statements” for details on branch target statements

IF Construct and Statement

The IF construct conditionally executes one block of statements or constructs.

The IF statement conditionally executes one statement.

The decision to transfer control or to execute the statement or block is based on the evaluation of a logical expression within the IF statement or construct.

See Also

"Arithmetic IF Statement"

IF Construct

The IF construct conditionally executes one block of constructs or statements depending on the evaluation of a logical expression. (This construct was called a block IF statement in FORTRAN 77.)

The IF construct takes the following form:

```
[name:] IF (expr) THEN
    block
[ELSE IF (expr) THEN [name]
    block]...
[ELSE [name]
    block]
END IF [name]
```

name

Is the name of the IF construct.

expr

Is a scalar logical expression enclosed in parentheses.

block

Is a sequence of zero or more statements or constructs.

Rules and Behavior

If a construct name is specified at the beginning of an IF THEN statement, the same name must appear in the corresponding END IF statement. The same construct name must not be used for different named constructs in the same scoping unit.

Depending on the evaluation of the logical expression, one block or no block is executed. The logical expressions are evaluated in the order in which they appear, until a true value is found or an ELSE or END IF statement is encountered.

Once a true value is found or an ELSE statement is encountered, the block immediately following it is executed and the construct execution terminates.

If none of the logical expressions evaluate to true and no ELSE statement appears in the construct, no block in the construct is executed and the construct execution terminates.

NOTE. *No additional statement can be placed after the* IF THEN *statement in a block* IF *construct. For example, the following statement is invalid in the block* IF *construct:*

```
IF (e) THEN I = J
```

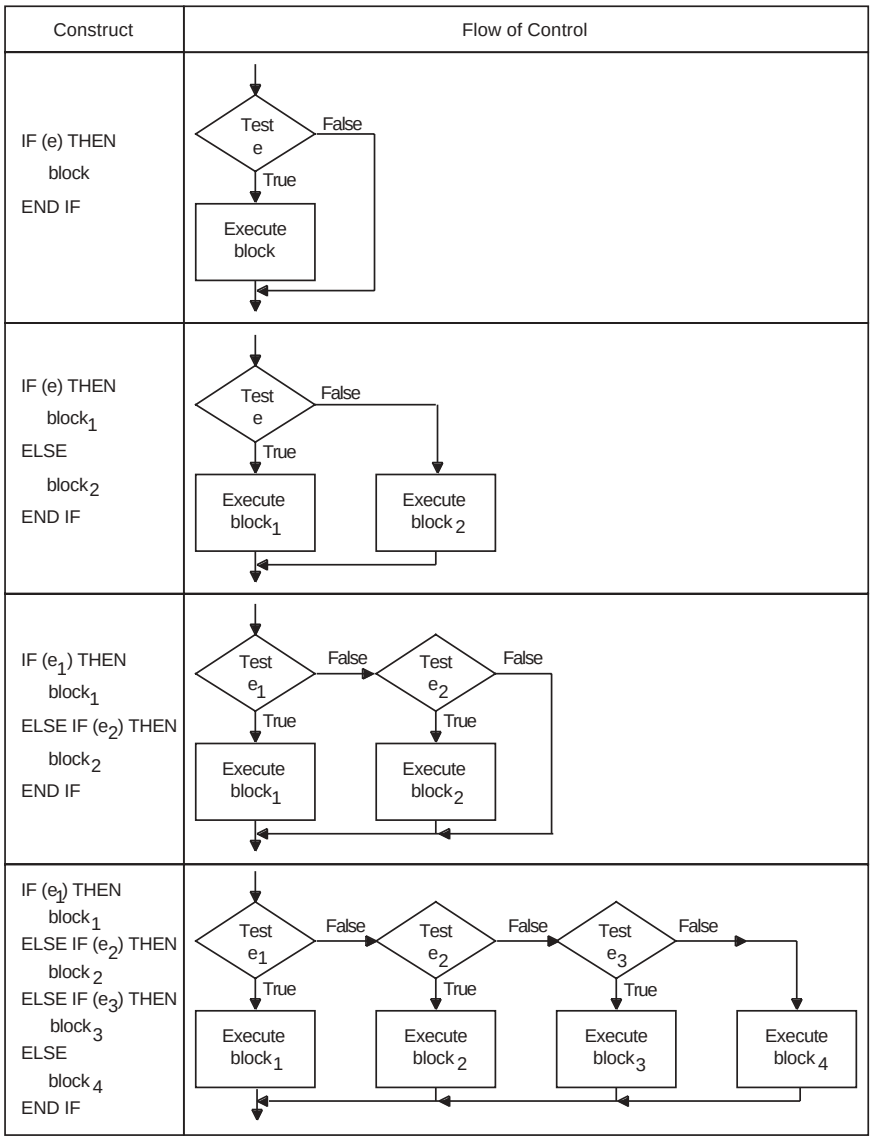
This statement is translated as the following logical IF *statement:*

```
IF (e) THENI = J
```

You cannot use branching statements to transfer control to an ELSE IF statement or ELSE statement. However, you can branch to an END IF statement from within the IF construct.

Figure 7-4 shows the flow of control in IF constructs.

Figure 7-4 Flow of Control in IF Constructs



ZK−0617−GE

You can include an IF construct in the statement block of another IF construct, if the nested IF construct is completely contained within a statement block. It cannot overlap statement blocks.

Examples

The following example shows the simplest form of an IF construct:

Form	Example
IF (expr) THEN	IF (ABS(ADJU) .GE. 1.0E-6) THEN
block	TOTERR = TOTERR + ABS(ADJU)
	QUEST = ADJU/FNDVAL
END IF	END IF

This construct conditionally executes the block of statements between the IF THEN and the END IF statements.

The following example shows an IF construct containing an ELSE statement:

Form	Example
IF (expr) THEN	IF (NAME .LT. 'N') THEN
block 1	IFRONT = IFRONT + 1
	FRLET(IFRONT) = NAME(1:2)
ELSE	ELSE
block 2	IBACK = IBACK + 1
END IF	END IF

Block1 consists of all the statements between the IF THEN and ELSE statements. Block2 consists of all the statements between the ELSE and the END IF statements.

If the value of the character variable NAME is less than 'N', block1 is executed. If the value of NAME is greater than or equal to 'N', block2 is executed.

The following example shows an IF construct containing an ELSE IF THEN statement:

Form	Example
IF (expr) THEN	IF (A .GT. B) THEN
block 1	D = B
	F = A – B
ELSE IF (expr) THEN	ELSE IF (A .GT. B/2.) THEN
block 2	D = B/2.

Form	Example
	F = A – B/2.
END IF	END IF

If A is greater than B, block1 is executed. If A is not greater than B, but A is greater than B/2, block2 is executed. If A is not greater than B and A is not greater than B/2, neither block1 nor block2 is executed. Control transfers directly to the next executable statement after the END IF statement.

The following example shows an IF construct containing several ELSE IF THEN statements and an ELSE statement:

Form	Example
IF (expr) THEN	IF (A .GT. B) THEN
block1	D = B
	F = A – B
ELSE IF (expr) THEN	ELSE IF (A .GT. C) THEN
block2	D = C
	F = A – C
ELSE IF (expr) THEN	ELSE IF (A .GT. Z) THEN
block3	D = Z
	F = A – Z
ELSE	ELSE
block4	D = 0.0
	F = A
END IF	END IF

If A is greater than B, block1 is executed. If A is not greater than B but is greater than C, block2 is executed. If A is not greater than B or C but is greater than Z, block3 is executed. If A is not greater than B, C, or Z, block4 is executed.

The following example shows a nested IF construct:

Form	Example
IF (expr) THEN	IF (A .LT. 100) THEN
block1	INRAN = INRAN + 1
IF (expr2) THEN	IF (ABS(A – AVG) .LE. 5.) THEN

Form	Example
block1a	INAVG = INAVG + 1
ELSE	ELSE
block1b	OUTAVG = OUTAVG + 1
END IF	END IF
ELSE	ELSE
block2	OUTRAN = OUTRAN + 1
END IF	END IF

If A is less than 100, the code immediately following the IF is executed. This code contains a nested IF construct. If the absolute value of A minus AVG is less than or equal to 5, block1a is executed. If the absolute value of A minus AVG is greater than 5, block1b is executed.

If A is greater than or equal to 100, block2 is executed, and the nested IF construct (in block1) is not executed.

The following example shows a named IF construct:

```
BLOCK_A: IF (D > 0.0) THEN       ! Initial statement for named construct
  RADIANS = ACOS(D)              ! These two statements
  DEGREES = ACOSD(D)             !   form a block
END IF BLOCK_A                  ! Terminal statement for named construct
```

IF Statement

The IF statement conditionally executes one statement based on the value of a logical expression. (This statement was called a logical IF statement in FORTRAN 77.)

The IF statement takes the following form:

IF (*expr*) *stmt*

expr

Is a scalar logical expression enclosed in parentheses.

stmt

Is any complete, unlabeled, executable Fortran statement, except for the following:

- A CASE, DO, IF, FORALL, or WHERE construct
- Another IF statement
- The END statement for a program, function, or subroutine

When an IF statement is executed, the logical expression is evaluated first. If the value is true, the statement is executed. If the value is false, the statement is not executed and control transfers to the next statement in the program.

Examples

The following examples show valid IF statements:

```
IF (J.GT.4 .OR. J.LT.1) GO TO 250
IF (REF(J,K) .NE. HOLD) REF(J,K) = REF(J,K) * (-1.5D0)
IF (ENDRUN) CALL EXIT
```

PAUSE Statement

The PAUSE statement temporarily suspends program execution until the user or system resumes execution. The PAUSE statement is a deleted feature in Fortran 95; it was obsolescent in Fortran 90. Intel Fortran fully supports features deleted in Fortran 95.

The PAUSE statement takes the following form:

PAUSE [*pause-code*]

pause-code

Is an optional message. It can be either of the following:

- A scalar character constant of type default character.
- A string of up to six digits; leading zeros are ignored. (Fortran 90 and FORTRAN 77 limit digits to five.)

Rules and Behavior

If you specify *pause-code*, the PAUSE statement displays the specified message and then displays the default prompt.

If you do not specify *pause-code*, the system displays the following default message:

```
FORTRAN PAUSE
```

The following prompt is then displayed:

- On Linux* systems:
  ```
  PAUSE prompt>
  ```
- On Windows* systems:
  ```
  Fortran Pause - Enter command<CR> or <CR> to continue.
  ```

Effect on Linux* Systems

The effect of PAUSE differs depending on whether the program is a foreground or background process, as follows:

- If a program is a foreground process, the program is suspended until you enter the CONTINUE command. Execution then resumes at the next executable statement.

  Any other command terminates execution.
- If a program is a background process, the behavior depends on `stdin`, as follows:
  - If `stdin` is redirected from a file, the system displays the following (after the pause code and prompt):

    ```
    To continue from background, execute 'kill -15 n'
    ```

    In this message, `n` is the process id of the program.
  - If `stdin` is not redirected from a file, the program becomes a suspended background job, and you must specify `fg` to bring the job into the foreground. You can then enter a command to resume or terminate processing.

Effect on Windows* Systems

The program waits for input on `stdin`. If you enter a blank line, execution resumes at the next executable statement.

Anything else is treated as a DOS command and is executed by a `system()`call. The program loops, letting you execute multiple DOS commands, until a blank line is entered. Execution then resumes at the next executable statement.

Examples

The following examples show valid PAUSE statements:

```
PAUSE 701
PAUSE 'ERRONEOUS RESULT DETECTED'
```

See Also

- Your user's guide for details on `stdin`
- Appendix A, "Deleted and Obsolescent Language Features", for details on obsolescent features in Fortran 95 and Fortran 90, as well as features deleted in Fortran 95

RETURN Statement

The RETURN statement transfers control from a subprogram to the calling program unit.

The RETURN statement takes the following form:

RETURN [*expr*]

expr

Is a scalar expression that is converted to an integer value if necessary.

The *expr* is only allowed in subroutines; it indicates an alternate return. (An alternate return is an obsolescent feature in Fortran 95 and Fortran 90.)

Rules and Behavior

When a RETURN statement is executed in a function subprogram, control is transferred to the referencing statement in the calling program unit.

When a RETURN statement is executed in a subroutine subprogram, control is transferred to the first executable statement following the CALL statement that invoked the subroutine, or to the alternate return (if one is specified).

Examples

The following shows how alternate returns can be used in a subroutine:

```
   CALL CHECK(A, B, *10, *20, C)
   ...
10 ...
20 ...
   SUBROUTINE CHECK(X, Y, *, *, C)
   ...
50   IF (X) 60, 70, 80
60   RETURN
70   RETURN 1
80   RETURN 2
   END
```

The value of X determines the return, as follows:

- If X < 0, a normal return occurs and control is transferred to the first executable statement following CALL CHECK in the calling program.
- If X = = 0, the first alternate return (RETURN 1) occurs and control is transferred to the statement identified with label 10.
- If X > 0, the second alternate return (RETURN 2) occurs and control is transferred to the statement identified with label 20.

Note that an asterisk (*) specifies the alternate return. An ampersand (&) can also specify an alternate return in a CALL statement, but not in a subroutine's dummy argument list.

See Also

- "CALL Statement"

- Appendix A, "Deleted and Obsolescent Language Features", for details on obsolescent features in Fortran 95 and Fortran 90

STOP Statement

The STOP statement terminates program execution before the end of the program unit. It takes the following form:

STOP [*stop-code*]

stop-code

Is an optional message. It can be either of the following:

- A scalar character constant of type default character.
- A string of up to six digits; leading zeros are ignored. (Fortran 95/90 and FORTRAN 77 limit digits to five.)

Effect on Linux* Systems

If you specify *stop-code*, the STOP statement writes the specified message to the standard error device and terminates program execution. The program returns a status of zero to the operating system.

If you do not specify *stop-code*, no message is output.

Effect on Windows* Systems

If you specify *stop-code*, the effect differs depending on its form, as follows:

- If *stop-code* is specified as a character constant, the STOP statement writes the specified message to the standard error device and terminates program execution. The program returns a status of zero to the operating system.
- If *stop-code* is specified as a string of digits, the STOP statement writes the following to the standard error device and terminates program execution:

  `Return code stop-code`

  In QuickWin programs, the following is displayed in a message box:

  `Program terminated with Exit Code stop-code`

  In both cases, the program returns a status of *stop-code* to the operating system as an integer.

If you do not specify *stop-code*, the STOP statement writes the following default message to the standard error device and terminates program execution:

```
Stop - Program terminated.
```

The program returns a status of zero to the operating system.

Examples

The following examples show valid STOP statements:

```
STOP 98
STOP 'END OF RUN'
DO
   READ *, X, Y
  IF (X > Y) STOP 5555
END DO
```

Program Units and Procedures

8

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

- Main program

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

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

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

  Program units that provide initial values for variables in named common blocks.

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

A procedure can be invoked during program execution to perform a specific task.

There are several kinds of procedures, as follows:

Kind of Procedure	Description
External procedure	A procedure that is not part of any other program unit
Module procedure	A procedure defined within a module
Internal procedure[1]	A procedure (other than a statement function) contained within a main program, function, or subroutine
Intrinsic procedure	A procedure defined by the Fortran language
Dummy procedure	An actual argument specified as a procedure or appearing in a procedure reference

Kind of Procedure	Description
Statement function	A computing procedure defined by a single statement

1. The program unit that contains an internal procedure is called its **host**.

A *function* is invoked in an expression using the name of the function or a defined operator. It returns a single value (function result) that is used to evaluate the expression.

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

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

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

This chapter contains information on the following topics:

- "Main Program"
- "Modules and Module Procedures"
- "Block Data Program Units"
- "Functions, Subroutines, and Statement Functions"
- "External Procedures"
- "Internal Procedures"
- "Argument Association"
- "Procedure Interfaces"
- The "CONTAINS Statement"
- The "ENTRY Statement"

See Also

- Chapter 9, "Intrinsic Procedures"
- "Program Structure" for an overview of program structure
- "Scope" for details on the scope of program entities
- "Recursive Procedures" for details on recursion

Main Program

A Fortran program must include one main program. It takes the following form:

[PROGRAM *name*]
[*specification-part*]
[*execution-part*]
[CONTAINS
internal-subprogram-part]
END [PROGRAM [*name*]]

name

Is the name of the program.

specification-part

Is one or more specification statements, except for the following:

- INTENT (or its equivalent attribute)
- OPTIONAL (or its equivalent attribute)
- PUBLIC and PRIVATE (or their equivalent attributes)

An automatic object must not appear in a specification statement. If a SAVE statement is specified, it has no effect.

execution-part

Is one or more executable constructs or statements, except for ENTRY or RETURN statements.

internal-subprogram-part

Is one or more internal subprograms (defining internal procedures). The *internal-subprogram-part* is preceded by a CONTAINS statement.

Rules and Behavior

The PROGRAM statement is optional. Within a program unit, a PROGRAM statement can be preceded only by comment lines or an OPTIONS statement.

The END statement is the only required part of a program. If a name follows the END statement, it must be the same as the name specified in the PROGRAM statement.

The program name is considered global and must be unique. It cannot be the same as any local name in the main program or the name of any other program unit, external procedure, or common block in the executable program.

A main program must not reference itself (either directly or indirectly).

Examples

The following is an example of a main program:

```
PROGRAM TEST
   INTEGER C, D, E(20,20)      ! Specification part
   CALL SUB_1                  ! Executable part
...
CONTAINS
   SUBROUTINE SUB_1            ! Internal subprogram
   ...
   END SUBROUTINE SUB_1
END PROGRAM TEST
```

See Also

Your user's guide for details on the default name for a main program

Modules and Module Procedures

A module contains specifications and definitions that can be used by one or more program units. For the module to be accessible, the other program units must reference its name in a USE statement, and the module entities must be public.

A module takes the following form:

MODULE *name*
  [*specification-part*]
[CONTAINS
  module-subprogram
  [*module-subprogram*]...]
END [MODULE [*name*]]

name

Is the name of the module.

specification-part

Is one or more specification statements, except for the following:

- ENTRY
- FORMAT
- AUTOMATIC (or its equivalent attribute)
- INTENT (or its equivalent attribute)
- OPTIONAL (or its equivalent attribute)
- Statement functions

An automatic object must not appear in a specification statement.

module-subprogram

Is a function or subroutine subprogram that defines the module procedure. A function must end with END FUNCTION and a subroutine must end with END SUBROUTINE.

A module subprogram can contain internal procedures.

Rules and Behavior

If a name follows the END statement, it must be the same as the name specified in the MODULE statement.

The module name is considered global and must be unique. It cannot be the same as any local name in the main program or the name of any other program unit, external procedure, or common block in the executable program.

A module is host to any module procedures it contains, and entities in the module are accessible to the module procedures through host association.

A module must not reference itself (either directly or indirectly).

You can use the PRIVATE attribute to restrict access to procedures or variables within a module.

Although ENTRY statements, FORMAT statements, and statement functions are not allowed in the specification part of a module, they are allowed in the specification part of a module subprogram.

Any executable statements in a module can only be specified in a module subprogram.

A module can contain one or more procedure interface blocks, which let you specify an explicit interface for an external subprogram or dummy subprogram.

Examples

The following example shows a simple module that can be used to provide global data:

```
MODULE MOD_A
  INTEGER :: B, C
  REAL E(25,5)
END MODULE MOD_A
...
SUBROUTINE SUB_Z
  USE MOD_A               ! Makes scalar variables B and C, and array
  ...                     !   E available to this subroutine
END SUBROUTINE SUB_Z
```

The following example shows a module procedure:

```
MODULE RESULTS
...
CONTAINS
  FUNCTION MOD_RESULTS(X,Y)  ! A module procedure
  ...
  END FUNCTION MOD_RESULTS
END MODULE RESULTS
```

The following example shows a module containing a derived type:

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

The following example shows a module containing an interface block:

```
MODULE ARRAY_CALCULATOR
  INTERFACE
    FUNCTION CALC_AVERAGE(D)
      REAL :: CALC_AVERAGE
      REAL, INTENT(IN) :: D(:)
    END FUNCTION
  END INTERFACE
END MODULE ARRAY_CALCULATOR
```

The following example shows a derived-type definition that is public with components that are private:

```
MODULE MATTER
  TYPE ELEMENTS
    PRIVATE
    INTEGER C, D
  END TYPE
...
END MODULE MATTER
```

In this case, components C and D are private to type ELEMENTS, but type ELEMENTS is not private to MODULE MATTER. Any program unit that uses the module MATTER can declare variables of type ELEMENTS, and pass as arguments values of type ELEMENTS.

This design allows you to change components of a type without affecting other program units that use the module.

If a derived type is needed in more than one program unit, the definition should be placed in a module and accessed by a USE statement whenever it is needed, as follows:

```
MODULE STUDENTS
  TYPE STUDENT_RECORD
  ...
  END TYPE
CONTAINS
  SUBROUTINE COURSE_GRADE(...)
  TYPE(STUDENT_RECORD) NAME
  ...
  END SUBROUTINE
END MODULE STUDENTS
...
PROGRAM SENIOR_CLASS
  USE STUDENTS
  TYPE(STUDENT_RECORD) ID
  ...
END PROGRAM
```

Program SENIOR_CLASS has access to type STUDENT_RECORD, because it uses module STUDENTS. Module procedure COURSE_GRADE also has access to type STUDENT_RECORD, because the derived-type definition appears in its host.

See Also

- "Procedure Interfaces"
- "PRIVATE and PUBLIC Attributes and Statements"

Module References

A program unit references a module in a USE statement. This module reference lets the program unit access the public definitions, specifications, and procedures in the module.

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

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

See Also

- "USE Statement"

- "PRIVATE and PUBLIC Attributes and Statements"
- "Use and Host Association" for details on use association

USE Statement

The USE statement gives a program unit accessibility to public entities in a module. It takes one of the following forms:

USE *name* [, *rename-list*]

USE *name*, ONLY : [*only-list*]

name

Is the name of the module.

rename-list

Is one or more items having the following form:

local-name => *mod-name*

local-name

Is the name of the entity in the program unit using the module.

mod-name

Is the name of a public entity in the module.

only-list

Is the name of a public entity in the module or a generic identifier (a generic name, defined operator, or defined assignment).

An entity in the *only-list* can also take the form:

[*local-name* =>] *mod-name*

Rules and Behavior

If the USE statement is specified without the ONLY option, the program unit has access to all public entities in the named module.

If the USE statement is specified with the ONLY option, the program unit has access to only those entities following the option.

If more than one USE statement for a given module appears in a scoping unit, the following rules apply:

- If one USE statement does not have the ONLY option, all public entities in the module are accessible, and any *rename-list*s and *only-list*s are interpreted as a single, concatenated *rename-list*.

- If all the USE statements have ONLY options, all the *only-list*s are interpreted as a single, concatenated *only-list*. Only those entities named in one or more of the *only-list*s are accessible.

If two or more generic interfaces that are accessible in a scoping unit have the same name, the same operator, or are both assignments, they are interpreted as a single generic interface. Otherwise, multiple accessible entities can have the same name only if no reference to the name is made in the scoping unit.

The local names of entities made accessible by a USE statement must not be respecified with any attribute other than PUBLIC or PRIVATE. The local names can appear in namelist group lists, but not in a COMMON or EQUIVALENCE statement.

Examples

The following shows examples of the USE statement:

```
MODULE MOD_A
  INTEGER :: B, C
  REAL E(25,5), D(100)
END MODULE MOD_A
...
SUBROUTINE SUB_Y
  USE MOD_A, DX => D, EX => E   ! Array D has been renamed DX and array E
  ...                           !  has been renamed EX. Scalar variables B
END SUBROUTINE SUB_Y            !  and C are also available to this subrou-
...                             !  tine (using their module names).
SUBROUTINE SUB_Z
  USE MOD_A, ONLY: B, C         ! Only scalar variables B and C are
  ...                           !  available to this subroutine
END SUBROUTINE SUB_Z
...
```

The following example shows a module containing common blocks:

```
MODULE COLORS
  COMMON /BLOCKA/ C, D(15)
  COMMON /BLOCKB/ E, F
  ...
END MODULE COLORS
...
FUNCTION HUE(A, B)
  USE COLORS
```

```
  ...
END FUNCTION HUE
```

The USE statement makes all of the variables in the common blocks in module COLORS available to the function HUE.

To provide data abstraction, a user-defined data type and operations to be performed on values of this type can be packaged together in a module. The following example shows such a module:

```
MODULE CALCULATION
  TYPE ITEM
    REAL :: X, Y
  END TYPE ITEM
  INTERFACE OPERATOR (+)
    MODULE PROCEDURE ITEM_CALC
  END INTERFACE
CONTAINS
  FUNCTION ITEM_CALC (A1, A2)
    TYPE(ITEM) A1, A2, ITEM_CALC
    ...
  END FUNCTION ITEM_CALC
  ...
END MODULE CALCULATION

PROGRAM TOTALS
USE CALCULATION
TYPE(ITEM) X, Y, Z
  ...
  X = Y + Z
  ...
END
```

The USE statement allows program TOTALS access to both the type ITEM and the extended intrinsic operator + to perform calculations.

Block Data Program Units

A block data program unit provides initial values for nonpointer variables in named common blocks. It takes the following form:

BLOCK DATA [*name*]
[*specification-part*]
END [BLOCK DATA [*name*]]

name

Is the name of the block data program unit.

specification-part

Is one or more of the following statements:

COMMON	INTRINSIC	STATIC
DATA	PARAMETER	TARGET
Derived-type definition	POINTER	Type declaration[1]
DIMENSION	RECORD[2]	USE[3]
EQUIVALENCE	Record structure declaration[2]	
IMPLICIT	SAVE	

1. Can only contain attributes: DIMENSION, INTRINSIC, PARAMETER, POINTER, SAVE, STATIC, or TARGET.
2. For more information on the RECORD statement and record structure declarations, see "Record Structures".
3. Allows access to only named constants.

Rules and Behavior

A block data program unit need not be named, but there can only be one unnamed block data program unit in an executable program.

If a name follows the END statement, it must be the same as the name specified in the BLOCK DATA statement.

An interface block must not appear in a block data program unit and a block data program unit must not contain any executable statements.

If a DATA statement initializes any variable in a named common block, the block data program unit must have a complete set of specification statements establishing the common block. However, all of the variables in the block do not have to be initialized.

A block data program unit can establish and define initial values for more than one common block, but a given common block can appear in only one block data program unit in an executable program.

The name of a block data program unit can appear in the EXTERNAL statement of a different program unit to force a search of object libraries for the block data program unit at link time.

Examples

The following is an example of a block data program unit:

```
BLOCK DATA BLKDAT
  INTEGER S,X
  LOGICAL T,W
  DOUBLE PRECISION U
  DIMENSION R(3)
  COMMON /AREA1/R,S,U,T /AREA2/W,X,Y
  DATA R/1.0,2*2.0/, T/.FALSE./, U/0.214537D-7/, W/.TRUE./, Y/3.5/
END
```

See Also

- "DATA Statement"
- "EXTERNAL Attribute and Statement"
- "COMMON Statement" for details on common blocks

Functions, Subroutines, and Statement Functions

Functions, subroutines, and statement functions are user-written subprograms that perform computing procedures. The computing procedure can be either a series of arithmetic operations or a series of Fortran statements. A single subprogram can perform a computing procedure in several places in a program, to avoid duplicating a series of operations or statements in each place.

The following table shows the statements that define these subprograms, and how control is transferred to the subprogram:

Subprogram	Defining Statements	Control Transfer Method
Function	FUNCTION or ENTRY	Function reference[1]
Subroutine	SUBROUTINE or ENTRY	CALL statement[2]
Statement function	Statement function definition	Function reference

1. A function can also be invoked by a defined operation (see "Defining Generic Operators").
2. A subroutine can also be invoked by a defined assignment (see "Defining Generic Assignment").

A *function reference* is used in an expression to invoke a function; it consists of the function name and its actual arguments. The function reference returns a value to the calling expression that is used to evaluate the expression.

The following topics are described in this section:

- "General Rules for Function and Subroutine Subprograms"

- "Functions"
- "Subroutines"
- "Statement Functions"

See Also

- "ENTRY Statement"
- "CALL Statement"

General Rules for Function and Subroutine Subprograms

A subprogram can be an external, module, or internal subprogram. The END statement for an internal or module subprogram must be END SUBROUTINE [name] for a subroutine, or END FUNCTION [name] for a function. In an external subprogram, the SUBROUTINE and FUNCTION keywords are optional.

If a subprogram name appears after the END statement, it must be the same as the name specified in the SUBROUTINE or FUNCTION statement.

Function and subroutine subprograms can change the values of their arguments, and the calling program can use the changed values.

A SUBROUTINE or FUNCTION statement can be optionally preceded by an OPTIONS statement.

Dummy arguments (except for dummy pointers or dummy procedures) can be specified with an intent and can be made optional.

The following sections describe recursion, pure procedures, and user-defined elemental procedures.

See Also

- "Modules and Module Procedures"
- "Internal Procedures"
- "External Procedures"
- "Optional Arguments"
- "INTENT Attribute and Statement" for details on argument intent

Recursive Procedures

A recursive procedure can reference itself directly or indirectly. Recursion is permitted if the keyword RECURSIVE is specified in a FUNCTION or SUBROUTINE statement, or if RECURSIVE is specified as a compiler option or in an OPTIONS statement.

If a function is directly recursive and array valued, the keywords RECURSIVE and RESULT must both be specified in the FUNCTION statement.

The procedure interface is explicit within the subprogram in the following cases:

- When RECURSIVE is specified for a subroutine
- When RECURSIVE and RESULT are specified for a function

The keyword RECURSIVE must be specified if any of the following applies (directly or indirectly):

- The subprogram invokes itself.
- The subprogram invokes a subprogram defined by an ENTRY statement in the same subprogram.
- An ENTRY procedure in the same subprogram invokes one of the following:
  - Itself
  - Another ENTRY procedure in the same subprogram
  - The subprogram defined by the FUNCTION or SUBROUTINE statement

See Also

- "OPTIONS Statement"
- "Functions" for details on the FUNCTION statement
- "Subroutines" for details on the SUBROUTINE statement
- Your user's guide for details on compiler options

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 procedures are implicitly pure:

- All intrinsic functions
- The elemental intrinsic subroutine MVBITS

A statement function is pure only if all functions that it references are pure.

Rules and Behavior

Except for procedure arguments and pointer arguments, the following intent must be specified for all dummy arguments in the specification part of the procedure:

- For functions: INTENT(IN)
- For subroutines: any INTENT (IN, OUT, or INOUT)

A local variable declared in a pure procedure (including variables declared in any internal procedure) must not:

- Specify the SAVE attribute
- Be initialized in a type declaration statement or a DATA statement

The following variables have restricted use in pure procedures (and any internal procedures):

- Global variables
- Dummy arguments with INTENT(IN) (or no declared intent)
- Objects that are storage associated with any part of a global variable

They must not be used in any context that does either of the following:

- Causes their value to change. For example, they must not be used as:
  - The left side of an assignment statement or pointer assignment statement
  - An actual argument associated with a dummy argument with INTENT(OUT), INTENT(INOUT), or the POINTER attribute
  - An index variable in a DO or FORALL statement, or an implied-DO clause
  - The variable in an ASSIGN statement
  - An input item in a READ statement
  - An internal file unit in a WRITE statement
  - An object in an ALLOCATE, DEALLOCATE, or NULLIFY statement
  - An IOSTAT or SIZE specifier in an I/O statement, or the STAT specifier in a ALLOCATE or DEALLOCATE statement
- Creates a pointer to that variable. For example, they must not be used as:
  - The target in a pointer assignment statement
  - The right side of an assignment to a derived-type variable (including a pointer to a derived type) if the derived type has a pointer component at any level

A pure procedure must not contain the following:

- Any external I/O statement (including a READ or WRITE statement whose I/O unit is an external file unit number or *)
- A PAUSE statement
- A STOP statement

A pure procedure can be used in contexts where other procedures are restricted; for example:

- It can be called directly in a FORALL statement or be used in the mask expression of a FORALL statement.
- It can be called from a pure procedure. Pure procedures can only call other pure procedures.
- It can be passed as an actual argument to a pure procedure.

If a procedure is used in any of these contexts, its interface must be explicit and it must be declared pure in that interface.

Examples

The following shows a pure function:

```
PURE INTEGER FUNCTION MANDELBROT(X)
  COMPLEX, INTENT(IN) :: X
  COMPLEX  :: XTMP
  INTEGER  :: K
  ! Assume SHARED_DEFS includes the declaration
  !  INTEGER ITOL
  USE SHARED_DEFS

  K = 0
  XTMP = -X
  DO WHILE (ABS(XTMP)<2.0 .AND. K<ITOL)
    XTMP = XTMP**2 - X
    K = K + 1
  END DO
  ITER = K
END FUNCTION
```

The following shows the preceding function used in an interface block:

```
INTERFACE
  PURE INTEGER FUNCTION MANDELBROT(X)
    COMPLEX, INTENT(IN) :: X
  END FUNCTION MANDELBROT
END INTERFACE
```

The following shows a FORALL construct calling the MANDELBROT function to update all the elements of an array:

```
FORALL (I = 1:N, J = 1:M)
  A(I,J) = MANDELBROT(COMPLX((I-1)*1.0/(N-1), (J-1)*1.0/(M-1))
END FORALL
```

See Also

- "Elemental Procedures"
- "Functions" for details on the FUNCTION statement
- "Subroutines" for details on the SUBROUTINE statement
- "FORALL Statement and Construct" for details on pure procedures in FORALLs
- "Defining Explicit Interfaces" for details on pure procedures in interface blocks
- Your user's guide for details on how to use pure procedures

Elemental Procedures

An elemental procedure is a user-defined procedure that is a restricted form of pure procedure. An elemental procedure can be passed an array, which is acted upon one element at a time. Elemental procedures are a feature of Fortran 95.

To specify an elemental procedure, use the prefix ELEMENTAL in a FUNCTION or SUBROUTINE statement.

An explicit interface must be visible to the caller of an ELEMENTAL procedure.

For functions, the result must be scalar; it cannot have the POINTER or ALLOCATABLE attribute.

Dummy arguments have the following restrictions:

- They must be scalar.
- They cannot have the POINTER or ALLOCATABLE attribute.
- They (or their subobjects) cannot appear in a specification expression, except as an argument to one of the intrinsic functions BIT_SIZE, LEN, KIND, or the numeric inquiry functions.
- They cannot be *.
- They cannot be dummy procedures.

If the actual arguments are all scalar, the result is scalar. If the actual arguments are array-valued, the values of the elements (if any) of the result are the same as if the function or subroutine had been applied separately, in any order, to corresponding elements of each array actual argument.

Elemental procedures are pure procedures and all rules that apply to pure procedures also apply to elemental procedures.

Examples

Consider the following:

```
MIN (A, 0, B)          ! A and B are arrays of shape (S, T)
```

In this case, the elemental reference to the MIN intrinsic function is an array expression whose elements have the following values:

```
MIN (A(I,J), 0, B(I,J)), I = 1, 2, ..., S, J = 1, 2, ..., T
```

See Also

- "Determining When Procedures Require Explicit Interfaces"
- "Pure Procedures"
- "Optional Arguments"
- "Functions" for details on the FUNCTION statement
- "Subroutines" for details on the SUBROUTINE statement

Functions

A *function* subprogram is invoked in an expression and returns a single value (a function result) that is used to evaluate the expression.

The FUNCTION statement is the initial statement of a function subprogram. It takes the following form:

[*prefix*] FUNCTION name [([*d-arg-list*])] [RESULT (*r-name*)]

prefix

Is one of the following:

type [*keyword*]
keyword [*type*]

type

Is a data type specifier.

keyword

Is one of the following:

Keyword	Meaning
RECURSIVE	Permits direct recursion to occur. If a function is directly recursive and array valued, RESULT must also be specified (see "Recursive Procedures").
PURE	Asserts that the procedure has no side effects (see "Pure Procedures").
ELEMENTAL	A restricted form of pure procedure that acts on one array element at a time (see "Elemental Procedures").

name

Is the name of the function. If RESULT is specified, the function name must not appear in any specification statement in the scoping unit of the function subprogram.

The function name can be followed by the length of the data type. The length is specified by an asterisk (*) followed by any unsigned, nonzero integer that is a valid length for the function's type. For example, REAL FUNCTION LGFUNC*8 (Y, Z) specifies the function result as REAL(8) (or REAL*8).

This optional length specification is not permitted if the length has already been specified following the keyword CHARACTER.

d-arg-list

Is a list of one or more dummy arguments. If there are no dummy arguments and no RESULT variable, the parentheses can be omitted. For example, the following is valid:

```
FUNCTION F
```

r-name

Is the name of the function result. This name must not be the same as the function name. A function result can be declared with the POINTER or ALLOCATABLE attribute.

Rules and Behavior

The type and kind parameters (if any) of the function's result can be defined in the FUNCTION statement or in a type declaration statement within the function subprogram, but not both. If no type is specified, the type is determined by implicit typing rules in effect for the function subprogram.

Execution begins with the first executable construct or statement following the FUNCTION statement. Control returns to the calling program unit once the END statement (or a RETURN statement) is executed.

If you specify CHARACTER*(*), the function assumes the length declared for it in the program unit that invokes it. This type of character function can have different lengths when it is invoked by different program units; it is an obsolescent feature in Fortran 95.

If the length is specified as an integer constant, the value must agree with the length of the function specified in the program unit that invokes the function. If no length is specified, a length of 1 is assumed.

If the function is an array, allocatable, or a pointer, the declarations within the function must state these attributes for the function result name. The specification of the function result attributes, dummy argument attributes, and the information in the procedure heading collectively define the interface of the function.

The value of the result variable is returned by the function when it completes execution. Certain rules apply depending on whether the result is a pointer, as follows:

- If the result is a pointer, its allocation status must be determined before the function completes execution. (The function must associate a target with the pointer, or cause the pointer to be explicitly disassociated from a target.)

  The shape of the value returned by the function is determined by the shape of the result variable when the function completes execution.
- If the result is not a pointer, its value must be defined before the function completes execution. If the result is an array, all the elements must be defined; if the result is a derived-type structure, all the components must be defined.

A function subprogram cannot contain a SUBROUTINE statement, a BLOCK DATA statement, a PROGRAM statement, or another FUNCTION statement. ENTRY statements can be included to provide multiple entry points to the subprogram.

You can use a CALL statement to invoke a function as long as the function is not one of the following types:

- REAL(8)
- REAL(16)
- COMPLEX(8)
- COMPLEX(16)
- CHARACTER

Examples

The following example uses the Newton-Raphson iteration method `(F(X) = cosh(X) + cos(X) - A = 0)` to get the root of the function:

```
FUNCTION ROOT(A)
  X  = 1.0
  DO
    EX = EXP(X)
    EMINX = 1./EX
    ROOT  = X - ((EX+EMINX)*.5+COS(X)-A)/((EX-EMINX)*.5-SIN(X))
    IF (ABS((X-ROOT)/ROOT) .LT. 1E-6) RETURN
    X  = ROOT
  END DO
END
```

In the preceding example, the following formula is calculated repeatedly until the difference between X_i and X_{i+1} is less than 1.0E–6:

$$X_{i+1} = X_i - \frac{\cos(X_i) + \cos(X_i) - A}{\sinh(X_i) - \sin(X_i)}$$

The following example shows an assumed-length character function:

```
CHARACTER*(*) FUNCTION REDO(CARG)
  CHARACTER*1 CARG
  DO I=1,LEN(REDO)
    REDO(I:I) = CARG
  END DO
  RETURN
END FUNCTION
```

This function returns the value of its argument, repeated to fill the length of the function.

Within any given program unit, all references to an assumed-length character function must have the same length. In the following example, the REDO function has a length of 1000:

```
CHARACTER*1000 REDO, MANYAS, MANYZS
MANYAS = REDO('A')
MANYZS = REDO('Z')
```

Another program unit within the executable program can specify a different length. For example, the following REDO function has a length of 2:

```
CHARACTER HOLD*6, REDO*2
HOLD = REDO('A')//REDO('B')//REDO('C')
```

The following example shows a dynamic array-valued function:

```
FUNCTION SUB (N)
  REAL, DIMENSION(N) :: SUB
  ...
END FUNCTION
```

The following example shows an allocatable function with allocatable arguments:

```
MODULE AP
CONTAINS

 FUNCTION ADD_VEC(P1,P2)
  ! Function to add two allocatable arrays of possibly differing lengths.
  ! The arrays may be thought of as polynomials (coefficients)
   REAL, ALLOCATABLE :: ADD_VEC(:), P1(:), P2(:)

  ! This function returns an allocatable array whose length is set to
  ! the length of the larger input array.
  ALLOCATE(ADD_VEC(MAX(SIZE(P1), SIZE(P2))))
  M = MIN(SIZE(P1), SIZE(P2))
    ! Add up to the shorter input array size
```

```
  ADD_VEC(:M) = P1(:M) + P2(:M)
  ! Use the larger input array elements afterwards (from P1 or P2)
  IF(SIZE(P1) > M) THEN
    ADD_VEC(M+1:) = P1(M+1:)
  ELSE IF(SIZE(P2) > M) THEN
    ADD_VEC(M+1:) = P2(M+1:)
  ENDIF
 END FUNCTION
END MODULE

PROGRAM TEST
 USE AP
 REAL, ALLOCATABLE :: P(:), Q(:), R(:), S(:)
 ALLOCATE(P(3))
 ALLOCATE(Q(2))
 ALLOCATE(R(3))
 ALLOCATE(S(3))

 ! Notice that P and Q differ in length
 P = (/4,2,1/)  ! P = X**2 + 2X + 4
 Q = (/-1,1/)   ! Q =         X - 1
 PRINT *,' Result should be:    3.000000       3.000000       1.000000'
 PRINT *,' Coefficients are: ', ADD_VEC(P, Q)  ! X**2 + 3X + 3

 P = (/1,1,1/)  ! P =  X**2 +  X + 1
 R = (/2,2,2/)  ! R = 2X**2 + 2X + 2
 S = (/3,3,3/)  ! S = 3X**2 + 3X + 3
 PRINT *,' Result should be:    6.000000       6.000000       6.000000'
 PRINT *,' Coefficients are: ', ADD_VEC(ADD_VEC(P,R), S)
END
```

See Also

- "RESULT Keyword"
- "General Rules for Function and Subroutine Subprograms"
- "ENTRY Statement"
- "RETURN Statement"
- "Function References" for details on argument keywords in function references

RESULT Keyword

Normally, a function result is returned in the function's name, and all references to the function name are references to the function result.

However, if you use the RESULT keyword in a FUNCTION statement, you can specify a local variable name for the function result. In this case, all references to the function name are recursive calls, and the function name must not appear in specification statements.

The RESULT name must be different from the name of the function.

The following shows an example of a recursive function specifying a RESULT variable:

```
RECURSIVE FUNCTION FACTORIAL(P) RESULT(L)
  INTEGER, INTENT(IN) :: P
  INTEGER L
  IF (P == 1) THEN
    L = 1
  ELSE
    L = P * FACTORIAL(P - 1)
  END IF
END FUNCTION
```

Function References

Functions are invoked by a function reference in an expression or by a defined operation.

A function reference takes the following form:

fun ([*a-arg* [, *a-arg*]...])

fun

Is the name of the function subprogram.

a-arg

Is an actual argument optionally preceded by [keyword=], where *keyword* is the name of a dummy argument in the explicit interface for the function. The keyword is assigned a value when the procedure is invoked.

Each actual argument must be a variable, an expression, or the name of a procedure. (It must not be the name of an internal procedure, statement function, or the generic name of a procedure.)

Rules and Behavior

When a function is referenced, each actual argument is associated with the corresponding dummy argument by its position in the argument list or by the name of its keyword. The arguments must agree in type and kind parameters.

Execution of the function produces a result that is assigned to the function name or to the result name, depending on whether the RESULT keyword was specified.

The program unit uses the result value to complete the evaluation of the expression containing the function reference.

If positional arguments and argument keywords are specified, the argument keywords must appear last in the actual argument list.

If a dummy argument is optional, the actual argument can be omitted.

If a dummy argument is specified with the INTENT attribute, its use may be limited. A dummy argument whose intent is not specified is subject to the limitations of its associated actual argument.

An actual argument associated with a dummy procedure must be the specific name of a procedure, or be another dummy procedure. Certain specific intrinsic function names must not be used as actual arguments (see Table 9-1).

Examples

Consider the following example:

```
X = 2.0
NEW_COS = COS(X)         ! A function reference
```

Intrinsic function COS calculates the cosine of 2.0. The value –0.4161468 is returned (in place of COS(X)) and assigned to NEW_COS.

See Also

- "INTENT Attribute and Statement"
- "Optional Arguments"
- "Defining Generic Operators" for details on defined operations
- "Argument Association" for details on procedure arguments
- "Dummy Procedure Arguments" for details on dummy arguments
- Chapter 9, "Intrinsic Procedures", for details on intrinsic functions
- "RESULT Keyword" for details on using the keyword in FUNCTION statements
- "Functions" for details on the FUNCTION statement

Subroutines

A *subroutine* subprogram is invoked in a CALL statement or by a defined assignment statement, and does not return a particular value.

The SUBROUTINE statement is the initial statement of a subroutine subprogram. It takes the following form:

[*prefix*] SUBROUTINE *name* [([*d-arg-list*])]

prefix

Is one of the following:

Keyword	Meaning
RECURSIVE	Permits direct recursion to occur. If a function is directly recursive and array valued, RESULT must also be specified (see "Recursive Procedures").
PURE	Asserts that the procedure has no side effects (see "Pure Procedures").
ELEMENTAL	A restricted form of pure procedure that acts on one array element at a time (see "Elemental Procedures").

name

Is the name of the subroutine.

d-arg-list

Is a list of one or more dummy arguments or alternate return specifiers (*).

Rules and Behavior

A subroutine is invoked by a CALL statement or defined assignment. When a subroutine is invoked, dummy arguments (if present) become associated with the corresponding actual arguments specified in the call.

Execution begins with the first executable construct or statement following the SUBROUTINE statement. Control returns to the calling program unit once the END statement (or a RETURN statement) is executed.

A subroutine subprogram cannot contain a FUNCTION statement, a BLOCK DATA statement, a PROGRAM statement, or another SUBROUTINE statement. ENTRY statements can be included to provide multiple entry points to the subprogram.

Examples

The following example shows a subroutine:

Main Program	Subroutine
CALL HELLO_WORLD	SUBROUTINE HELLO_WORLD
. . .	PRINT *, "Hello World"

Main Program	Subroutine
END	END SUBROUTINE

The following example uses alternate return specifiers to determine where control transfers on completion of the subroutine:

Main Program	Subroutine
CALL CHECK(A,B,*10,*20,C)	SUBROUTINE CHECK(X,Y,*.*.Q)
TYPE *, 'VALUE LESS THAN ZERO'	. . .
GO TO 30	50 IF(Z) 60,70,80
10 TYPE*, 'VALUE EQUALS ZERO'	60 RETURN
GO TO 30	70 RETURN
20 TYPE*, 'VALUE MORE THAN ZERO'	80 RETURN
30 CONTINUE	END
. . .	

The SUBROUTINE statement argument list contains two dummy alternate return arguments corresponding to the actual arguments *10 and *20 in the CALL statement argument list.

The value of Z determines the return, as follows:

- If Z < zero, a normal return occurs and control is transferred to the first executable statement following CALL CHECK in the main program.
- If Z = = zero, the return is to statement label 10 in the main program.
- If Z > zero, the return is to statement label 20 in the main program.

An alternate return is an obsolescent feature in Fortran 90 and Fortran 95.

See Also

- "General Rules for Function and Subroutine Subprograms"
- "RETURN Statement"
- "ENTRY Statement"
- "CALL Statement" for details on argument keywords in subroutine references
- "Defining Generic Assignment" for details on defined assignment
- "Argument Association" for details on procedure arguments
- Chapter 9, "Intrinsic Procedures", for details on intrinsic subroutines
- Appendix A, "Deleted and Obsolescent Language Features", for details on obsolescent features in Fortran 90 and Fortran 95

Statement Functions

A statement function is a procedure defined by a single statement in the same program unit in which the procedure is referenced. It takes the following form:

fun ([*d-arg* [, *d-arg*]...]) = *expr*

fun

Is the name of the statement function.

d-arg

Is a dummy argument. A dummy argument can appear only once in any list of dummy arguments, and its scope is local to the statement function.

expr

Is a scalar expression defining the computation to be performed.

Named constants and variables used in the expression must have been declared previously in the specification part of the scoping unit or made accessible by use or host association.

If the expression contains a function reference, the function must have been defined previously in the same program unit.

A statement function reference takes the following form:

fun ([*a-arg* [, *a-arg*]...])

fun

Is the name of the statement function.

a-arg

Is an actual argument.

Rules and Behavior

When a statement function reference appears in an expression, the values of the actual arguments are associated with the dummy arguments in the statement function definition. The expression in the definition is then evaluated. The resulting value is used to complete the evaluation of the expression containing the function reference.

The data type of a statement function can be explicitly defined in a type declaration statement. If no type is specified, the type is determined by implicit typing rules in effect for the program unit.

Actual arguments must agree in number, order, and data type with their corresponding dummy arguments.

Except for the data type, declarative information associated with an entity is not associated with dummy arguments in the statement function; for example, declaring an entity to be an array or to be in a common block does not affect a dummy argument with the same name.

The name of the statement function cannot be the same as the name of any other entity within the same program unit.

Any reference to a statement function must appear in the same program unit as the definition of that function.

A statement function reference must appear as (or be part of) an expression. The reference cannot appear on the left side of an assignment statement.

A statement function must not be provided as a procedure argument.

Examples

The following are examples of statement functions:

```
REAL VOLUME, RADIUS
VOLUME(RADIUS) = 4.189*RADIUS**3

CHARACTER*10 CSF,A,B
CSF(A,B) = A(6:10)//B(1:5)
```

The following example shows a statement function and some references to it:

```
AVG(A,B,C) = (A+B+C)/3.
...
GRADE = AVG(TEST1,TEST2,XLAB) I
F (AVG(P,D,Q) .LT. AVG(X,Y,Z)) STOP
FINAL = AVG(TEST3,TEST4,LAB2)       ! Invalid reference; implicit
...                                 !  type of third argument does not
...                                 !  match implicit type of dummy argument
```

Implicit typing problems can be avoided if all arguments are explicitly typed.

The following statement function definition is invalid because it contains a constant, which cannot be used as a dummy argument:

```
REAL COMP, C, D, E
COMP(C,D,E,3.) = (C + D - E)/3.
```

See Also

- "Use and Host Association"
- "Argument Association" for details on procedure arguments

External Procedures

External procedures are user-written functions or subroutines. They are located outside of the main program and can't be part of any other program unit.

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

In Fortran 95/90, external procedures can include internal subprograms (defining internal procedures). An internal subprogram begins with a CONTAINS statement.

An external procedure can reference itself (directly or indirectly).

The interface of an external procedure is implicit unless an interface block is supplied for the procedure.

See Also

- "Procedure Interfaces"
- "Functions, Subroutines, and Statement Functions" for details on function and subroutine subprograms
- Your user's guide for details on passing arguments

Internal Procedures

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

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

An internal procedure takes the following form:

CONTAINS
internal-subprogram
[*internal-subprogram*]...

internal-subprogram

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

Rules and Behavior

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

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

- An internal procedure must not contain an ENTRY statement.

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

The interface of an internal procedure is always explicit.

Examples

The following example shows an internal procedure:

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

See Also

- "Use and Host Association"
- "Procedure Interfaces"
- "Functions, Subroutines, and Statement Functions" for details on function and subroutine subprograms

Argument Association

Procedure arguments provide a way for different program units to access the same data.

When a procedure is referenced in an executable program, the program unit invoking the procedure can use one or more *actual* arguments to pass values to the procedure's *dummy* arguments. The dummy arguments are associated with their corresponding actual arguments when control passes to the subprogram.

In general, when control is returned to the calling program unit, the last value assigned to a dummy argument is assigned to the corresponding actual argument.

An actual argument can be a variable, expression, or procedure name. The type and kind parameters, and rank of the actual argument must match those of its associated dummy argument.

A dummy argument is either a dummy data object, a dummy procedure, or an alternate return specifier (*). Except for alternate return specifiers, dummy arguments can be optional.

If argument keywords are not used, argument association is positional. The first dummy argument becomes associated with the first actual argument, and so on. If argument keywords are used, arguments are associated by the keyword name, so actual arguments can be in a different order than dummy arguments.

A keyword is required for an argument only if a preceding optional argument is omitted or if the argument sequence is changed.

A scalar dummy argument can be associated with only a scalar actual argument.

If a dummy argument is an array, it must be no larger than the array that is the actual argument. You can use adjustable arrays to process arrays of different sizes in a single subprogram.

An actual argument associated with a dummy argument that is allocatable or a pointer must have the same type parameters as the dummy argument.

A dummy argument referenced as a subprogram must be associated with an actual argument that has been declared EXTERNAL or INTRINSIC in the calling routine.

If a scalar dummy argument is of type character, its length must not be greater than the length of its associated actual argument.

If the character dummy argument's length is specified as *(*) (assumed length), it uses the length of the associated actual argument.

Once an actual argument has been associated with a dummy argument, no action can be taken that affects the value or availability of the actual argument, except indirectly through the dummy argument. For example, if the following statement is specified:

```
CALL SUB_A (B(2:6), B(4:10))
```

B(4:6) must not be defined, redefined, or become undefined through either dummy argument, since it is associated with both arguments. However, B(2:3) is definable through the first argument, and B(7:10) is definable through the second argument.

Similarly, if any part of the actual argument is defined through a dummy argument, the actual argument can only be referenced through that dummy argument during execution of the procedure. For example, if the following statements are specified:

```
MODULE MOD_A
  REAL :: A, B, C, D
END MODULE MOD_A

PROGRAM TEST
  USE MOD_A
  CALL SUB_1 (B)
  ...
END PROGRAM TEST
```

```
SUBROUTINE SUB_1 (F)
  USE MOD_A
  ...
  WRITE (*,*) F
END SUBROUTINE SUB_1
```

Variable B must not be directly referenced during the execution of SUB_1 because it is being defined through dummy argument F. However, B can be indirectly referenced through F (and directly referenced when SUB_1 completes execution).

The following sections provide more details on arguments:

- "Optional Arguments"
- The different kinds of arguments:
  - "Array Arguments"
  - "Pointer Arguments"
  - "Assumed-Length Character Arguments"
  - "Character Constant and Hollerith Arguments"
  - "Alternate Return Arguments"
  - "Dummy Procedure Arguments"
- "References to Generic Procedures"
- "References to Non-Fortran Procedures"

See Also

- "CALL Statement" for details on argument keywords in subroutine references
- "Function References" for details on argument keywords in function references
- "%REF and %VAL Argument List Functions" for details on built-in functions to pass actual arguments

Optional Arguments

Dummy arguments can be made optional if they are declared with the OPTIONAL attribute. In this case, an actual argument does not have to be supplied for it in a procedure reference.

Positional arguments (if any) must appear first in an actual argument list, followed by keyword arguments (if any). If an optional argument is the last positional argument, it can simply be omitted if desired.

However, if the optional argument is to be omitted but it is not the last positional argument, keyword arguments must be used for any subsequent arguments in the list.

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

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

The following example shows optional arguments:

```
PROGRAM RESULT
TEST_RESULT = LGFUNC(A, B=D)
...
CONTAINS
  FUNCTION LGFUNC(G, H, B)
    OPTIONAL H, B
    ...
  END FUNCTION
END
```

In the function reference, A is a positional argument associated with required dummy argument G. The second actual argument D is associated with optional dummy argument B by its keyword name (B). No actual argument is associated with optional argument H.

See Also

- "Argument Association"
- "OPTIONAL Attribute and Statement"
- "PRESENT"
- "CALL Statement" for details on argument keywords in subroutine references
- "Function References" for details on argument keywords in function references

Array Arguments

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

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

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

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

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

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

If a dummy argument is allocatable, the actual argument must be allocatable and the type parameters and ranks must agree. An example of an allocatable function with allocatable arrays appears in "Functions".

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

See Also

- "Argument Association"
- "Arrays"
- "Array Association"
- "Assumed-Shape Specifications" for details on assumed-shape arrays
- "Array Elements" for details on array element order
- "Explicit-Shape Specifications" for details on explicit-shape arrays
- "Assumed-Size Specifications" for details on assumed-size arrays

Pointer Arguments

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

When a procedure is invoked, the dummy argument pointer receives the pointer association status of the actual argument. If the actual argument is currently associated, the dummy argument becomes associated with the same target.

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

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

A dummy argument that is a pointer can be associated only with an actual argument that is a pointer. However, an actual argument that is a pointer can be associated with a nonpointer dummy argument. In this case, the actual argument is associated with a target and the dummy argument, through argument association, also becomes associated with that target.

If the dummy argument does not have the TARGET or POINTER attribute, any pointers associated with the actual argument do not become associated with the corresponding dummy argument when the procedure is invoked.

If the dummy argument has the TARGET attribute, and is either a scalar or assumed-shape array, and the corresponding actual argument has the TARGET attribute but is not an array section with a vector subscript, the following occurs:

- Any pointer associated with the actual argument becomes associated with the corresponding dummy argument when the procedure is invoked.
- Any pointers associated with the dummy argument remain associated with the actual argument when execution of the procedure completes.

If the dummy argument has the TARGET attribute, and is an explicit-shape or assumed-size array, and the corresponding actual argument has the TARGET attribute but is not an array section with a vector subscript, association of actual and corresponding dummy arguments when the procedure is invoked or when execution is completed is processor dependent.

If the dummy argument has the TARGET attribute and the corresponding actual argument does not have that attribute or is an array section with a vector subscript, any pointer associated with the dummy argument becomes undefined when execution of the procedure completes.

See Also

- "Argument Association"
- "Pointer Assignments"
- "TARGET Attribute and Statement"
- "POINTER Attribute and Statement" for details on pointers
- Your user's guide for details on passing pointers as arguments

Assumed-Length Character Arguments

An assumed-length character argument is a dummy argument that assumes the length attribute of its corresponding actual argument. An asterisk (*) specifies the length of the dummy character argument.

A character array dummy argument can also have an assumed length. The length of each element in the dummy argument is the length of the elements in the actual argument. The assumed length and the array declarator together determine the size of the assumed-length character array.

The following example shows an assumed-length character argument:

```
INTEGER FUNCTION ICMAX(CVAR)
  CHARACTER*(*) CVAR
  ICMAX = 1
  DO I=2,LEN(CVAR)
    IF (CVAR(I:I) .GT. CVAR(ICMAX:ICMAX)) ICMAX=I
  END DO
```

```
  RETURN
END
```

The function ICMAX finds the position of the character with the highest ASCII code value. It uses the length of the assumed-length character argument to control the iteration. Intrinsic function LEN determines the length of the argument.

The length of the dummy argument is determined each time control transfers to the function. The length of the actual argument can be the length of a character variable, array element, substring, or expression. Each of the following function references specifies a different length for the dummy argument:

```
CHARACTER VAR*10, CARRAY(3,5)*20
...
I1 = ICMAX(VAR)
I2 = ICMAX(CARRAY(2,2))
I3 = ICMAX(VAR(3:8))
I4 = ICMAX(CARRAY(1,3)(5:15))
I5 = ICMAX(VAR(3:4)//CARRAY(3,5))
```

See Also

- "LEN"
- "Argument Association"

Character Constant and Hollerith Arguments

If an actual argument is a character constant (for example, `'ABCD'`), the corresponding dummy argument must be of type character. If an actual argument is a Hollerith constant (for example, 4HABCD), the corresponding dummy argument must have a numeric data type.

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

```
SUBROUTINE S(CHARSUB, HOLLSUB, A, B)
EXTERNAL CHARSUB, HOLLSUB
...
CALL CHARSUB(A, 'STRING')
CALL HOLLSUB(B, 6HSTRING)
```

The subroutines CHARSUB and HOLLSUB are themselves dummy arguments of the subroutine S. Therefore, the actual argument `'STRING'` in the call to CHARSUB must correspond to a character dummy argument, and the actual argument 6HSTRING in the call to HOLLSUB must correspond to a numeric dummy argument.

See Also

"Argument Association"

Alternate Return Arguments

Alternate return (dummy) arguments can appear in a subroutine argument list. They cause execution to transfer to a labeled statement rather than to the statement immediately following the statement that called the routine. The alternate return is indicated by an asterisk (*). (An alternate return is an obsolescent feature in Fortran 90 and Fortran 95.)

There can be any number of alternate returns in a subroutine argument list, and they can be in any position in the list.

An actual argument associated with an alternate return dummy argument is called an alternate return specifier; it is indicated by an asterisk (*) or ampersand (&) followed by the label of an executable branch target statement in the same scoping unit as the CALL statement.

Alternate returns cannot be declared optional.

In Fortran 90, you can also use the RETURN statement to specify alternate returns.

The following example shows alternate return actual and dummy arguments:

```
CALL MINN(X, Y, *300, *250, Z)
...
SUBROUTINE MINN(A, B, *, *, C)
```

See Also

- "Argument Association"
- "Subroutines"
- "CALL Statement"
- "RETURN Statement"
- Appendix A, "Deleted and Obsolescent Language Features", for details on obsolescent features in Fortran 90 and Fortran 95

Dummy Procedure Arguments

If an actual argument is a procedure, its corresponding dummy argument is a dummy procedure. Dummy procedures can appear in function or subroutine subprograms.

The actual argument must be the specific name of an external, module, intrinsic, or another dummy procedure. If the specific name is also a generic name, only the specific name is associated with the dummy argument. Not all specific intrinsic procedures can appear as actual arguments. (For more information, see Table 9-1.)

The actual argument and corresponding dummy procedure must both be subroutines or both be functions.

If the interface of the dummy procedure is explicit, the type and kind parameters, and rank of the associated actual procedure must be the same as that of the dummy procedure.

If the interface of the dummy procedure is implicit and the procedure is referenced as a subroutine, the actual argument must be a subroutine or a dummy procedure.

If the interface of the dummy procedure is implicit and the procedure is referenced as a function or is explicitly typed, the actual argument must be a function or a dummy procedure.

Dummy procedures can be declared optional, but they must not be declared with an intent.

The following is an example of a procedure used as an argument:

```
REAL FUNCTION LGFUNC(BAR)
  INTERFACE
    REAL FUNCTION BAR(Y)
      REAL, INTENT(IN) :: Y
    END
  END INTERFACE
  ...
  LGFUNC = BAR(2.0)
  ...
END FUNCTION LGFUNC
```

See Also

"Argument Association"

References to Generic Procedures

Generic procedures are procedures with different specific names that can be accessed under one generic (common) name. In FORTRAN 77, generic procedures were limited to intrinsic procedures. In Fortran 90, you can use generic interface blocks to specify generic properties for intrinsic and user-defined procedures.

If you refer to a procedure by using its generic name, the selection of the specific routine is based on the number of arguments and the type and kind parameters, and rank of each argument.

All procedures given the same generic name must be subroutines, or all must be functions. Any two must differ enough so that any invocation of the procedure is unambiguous.

The following sections describe references to generic intrinsic functions and show an example of using intrinsic function names.

See Also

- "Unambiguous Generic Procedure References"
- Chapter 9, "Intrinsic Procedures"
- "Defining Generic Names for Procedures" for details on user-defined generic procedures
- "Resolving Procedure References" for details on the rules for resolving ambiguous references

References to Generic Intrinsic Functions

The generic intrinsic function name COS lists six specific intrinsic functions that calculate cosines: COS, DCOS, QCOS, CCOS, CDCOS, and CQCOS. These functions return different values: REAL(4), REAL(8), REAL(16), COMPLEX(4), COMPLEX(8), and COMPLEX(16) respectively.

If you invoke the cosine function by using the generic name COS, the compiler selects the appropriate routine based on the arguments that you specify. For example, if the argument is REAL(4), COS is selected; if it is REAL(8), DCOS is selected; and if it is COMPLEX(4), CCOS is selected.

You can also explicitly refer to a particular routine. For example, you can invoke the double-precision cosine function by specifying DCOS.

Procedure selection occurs independently for each generic reference, so you can use a generic reference repeatedly in the same program unit to access different intrinsic procedures.

You cannot use generic function names to select intrinsic procedures if you use them as follows:

- The name of a statement function
- A dummy argument name, a common block name, or a variable or array name

When an intrinsic function is passed as an actual argument to a procedure, its specific name must be used, and when called, its arguments must be scalar. Not all specific intrinsic functions can appear as actual arguments. (For more information, see Table 9-1.)

Generic procedure names are local to the program unit that refers to them, so they can be used for other purposes in other program units.

Normally, an intrinsic procedure name refers to the Fortran 90 library procedure with that name. However, the name can refer to a user-defined procedure when the name appears in an EXTERNAL statement.

NOTE. *If you call an intrinsic procedure by using the wrong number of arguments or an incorrect argument type, the compiler assumes you are referring to an external procedure. For example, intrinsic procedure* SIN *requires one argument; if you specify two arguments, such as SIN(10,4), the compiler assumes SIN is external and not intrinsic.*

Except when used in an EXTERNAL statement, intrinsic procedure names are local to the program unit that refers to them, so they can be used for other purposes in other program units. The data type of an intrinsic procedure does not change if you use an IMPLICIT statement to change the implied data type rules.

Intrinsic and user-defined procedures cannot have the same name if they appear in the same program unit.

Examples

Example 8-1 shows the local and global properties of an intrinsic function name. It uses intrinsic function SIN as follows:

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

Example 8-1 Using and Redefining an Intrinsic Function Name

```
!   Compare ways of computing sine
        PROGRAM SINES
          DOUBLE PRECISION X, PI
          PARAMETER (PI=3.14159265358979323 8D0)
          COMMON V(3)

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

Example 8-1 Using and Redefining an Intrinsic Function Name

```
   100    FORMAT (5F10.7)
        END
        SUBROUTINE COMPUT(Y)
          DOUBLE PRECISION Y
3   !   Use intrinsic function SIN as an actual argument
          INTRINSIC SIN
          COMMON V(3)
4   !   Define generic reference to double-precision sine
          V(1) = SIN(Y)
5   !   Use intrinsic SIN as an actual argument
          CALL SUB(REAL(Y),SIN)
        END

        SUBROUTINE SUB(A,S)

6   !   Declare SIN as the name of a user function
          EXTERNAL SIN

7   !   Declare SIN as type DOUBLE PRECISION
          DOUBLE PRECISION SIN
          COMMON V(3)

8   !   Evaluate intrinsic function SIN
          V(2) = S(A)
9   !   Evaluate user-defined SIN function
          V(3) = SIN(A)
        END

10  !   Define the user SIN function
        DOUBLE PRECISION FUNCTION SIN(X)
          INTEGER FACTOR
          SIN = X - X**3/FACTOR(3) + X**5/FACTOR(5)     &
              - X**7/FACTOR(7)
        END
```

1 The statement function named SIN is defined in terms of the generic function name COS. Because the argument of COS is double precision, the double-precision cosine function is evaluated. The statement function SIN is itself single precision.

2 The statement function SIN is called.

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

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

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

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

7 The type of SIN is declared double precision.

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

9 The user-defined SIN function is evaluated.

10 The user-defined SIN function is defined as a simple Taylor series using a user-defined function FACTOR to compute the factorial function.

See Also

- "EXTERNAL Attribute and Statement"
- "INTRINSIC Attribute and Statement"
- "Names" for details on the scope of names
- Chapter 9, "Intrinsic Procedures", for details on generic and specific intrinsic functions

References to Elemental Intrinsic Procedures

An *elemental intrinsic procedure* has scalar dummy arguments that can be called with scalar or array actual arguments. If actual arguments are array-valued, they must have the same shape. There are many elemental intrinsic functions, but only one elemental intrinsic subroutine (MVBITS).

If the actual arguments are scalar, the result is scalar. If the actual arguments are array-valued, the scalar-valued procedure is applied element-by-element to the actual argument, resulting in an array that has the same shape as the actual argument.

The values of the elements of the resulting array are the same as if the scalar-valued procedure had been applied separately to the corresponding elements of each argument.

For example, if A and B are arrays of shape (5,6), MAX(A, 0.0, B) is an array expression of shape (5,6) whose elements have the value MAX(A (i, j), 0.0, B (i, j)), where i = 1, 2,..., 5, and j = 1, 2,..., 6.

A reference to an elemental intrinsic procedure is an elemental reference if one or more actual arguments are arrays and all array arguments have the same shape.

See Also

- "Arrays"

- Chapter 9, "Intrinsic Procedures", for details on elemental procedures

References to Non-Fortran Procedures

To facilitate references to non-Fortran procedures, Intel Fortran provides built-in functions %REF and %VAL to pass actual arguments, and %LOC, which computes the internal address of a storage item.

%REF and %VAL Argument List Functions

When a procedure is called, Fortran (by default) passes the address of the actual argument, and its length if it is of type character. To call non-Fortran procedures, you may need to pass the actual arguments in a form different from that used by Fortran.

The built-in functions %REF and %VAL let you change the form of an actual argument. You must specify these functions in the actual argument list of a CALL statement or function reference. You cannot use them in any other context.

These functions specify how to pass an actual argument (for example, *a*) to a non-Fortran procedure, as follows:

Function	Effect
%REF (*a*)	Passes argument *a* by reference.
%VAL (*a*)	Passes argument *a* as an *n*-bit[1] immediate value. If *a* is integer (or logical) and shorter than *n* bits, it is sign-extended to an *n*-bit value. For complex data types, %VAL passes two *n*-bit arguments.

1. *n* is 64 on Intel® Itanium® processors; 32 on IA-32 processors.

Table 8-1 lists the Intel Fortran defaults for argument passing, and the allowed uses of %VAL and %REF.

Table 8-1 Defaults for Argument List Functions

		Allowed Functions	
Actual Argument Data Type	**Default**	**%VAL**	**%REF**
Expressions:			
Logical	REF	Yes[1]	Yes
Integer	REF	Yes[1]	Yes
REAL(4)	REF	Yes	Yes
REAL(8)	REF	Yes[2]	Yes

Table 8-1 Defaults for Argument List Functions

Actual Argument Data Type	Default	Allowed Functions	
		%VAL	%REF
REAL(16)	REF	No	Yes
COMPLEX(4)	REF	Yes	Yes
COMPLEX(8)	REF	Yes	Yes
COMPLEX(16)	REF	No	Yes
Character	N/A[3]	No	Yes
Hollerith	REF	No	No
Aggregate[4]	REF	No	Yes
Derived	REF	No	Yes
Array Name:			
Numeric	REF	No	Yes
Character	N/A[3]	No	Yes
Aggregate[4]	REF	No	Yes
Derived	REF	No	Yes
Procedure Name:			
Numeric	REF	No	Yes
Character	N/A[3]	No	Yes

1. If a logical or integer value occupies less than 64 bits of storage on Intel Itanium processors, or 32 bits of storage on IA-32 processors, it is converted to the correct size by sign extension. Use the ZEXT function if zero extension is desired.
2. i64 only
3. A character argument is passed by address and hidden length. For more information, see your user's guide.
4. In Intel Fortran record structures

The %REF and %VAL functions override related cDEC$ ATTRIBUTE settings.

See Also

Your user's guide for details on how to use the %REF and %VAL functions

%LOC Function

The built-in function %LOC computes the internal address of a storage item. It takes the following form:

%LOC (*arg*)

arg

Is the name of an actual argument. It must be a variable, an expression, or the name of a procedure. (It must not be the name of an internal procedure or statement function.)

The %LOC function produces an integer value that represents the location of the given argument. The value is INTEGER(8) on Intel Itanium processors; INTEGER(4) on IA-32 processors. You can use this integer value as an item in an arithmetic expression.

The LOC intrinsic function serves the same purpose as the %LOC built-in function.

See Also

- "LOC" for details on the LOC intrinsic function
- Your user's guide for details on how to use the %LOC function

Procedure Interfaces

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

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

Kind of Procedure	Interface
External procedure	Implicit [1]
Module procedure	Explicit
Internal procedure	Explicit
Intrinsic procedure	Explicit
Dummy procedure	Implicit[1]
Statement function	Implicit

1. Unless an interface block is supplied for the procedure.

The interface of a recursive subroutine or function is explicit within the subprogram that defines it.

An explicit interface can appear in a procedure's definition, in an interface block, or both. (Internal procedures must not appear in an interface block.)

The following sections describe when explicit interfaces are required, how to define explicit interfaces, and how to define generic names, operators, and assignment.

Determining When Procedures Require Explicit Interfaces

A procedure must have an explicit interface in the following cases:

- If the procedure has any of the following:
  - A dummy argument that has the ALLOCATABLE, OPTIONAL, POINTER, TARGET, or VOLATILE attribute
  - A dummy argument that is an assumed-shape array
  - A result that is an array, or a pointer, or is allocatable (functions only)
  - A result whose length is neither assumed nor a constant (character functions only)
- If a reference to the procedure appears as follows:
  - With an argument keyword
  - As a reference by its generic name
  - As a defined assignment (subroutines only)
  - In an expression as a defined operator (functions only)
  - In a context that requires it to be pure
  - If the procedure is elemental

See Also

- "Optional Arguments"
- "Array Arguments"
- "Pointer Arguments"
- "Pure Procedures"
- "Elemental Procedures"
- "CALL Statement" for details on argument keywords in subroutine references
- "Function References" for details on argument keywords in function references
- "Defining Generic Names for Procedures" for details on user-defined generic procedures
- "Defining Generic Operators" for details on defined operators
- "Defining Generic Assignment" for details on defined assignment
- Your user's guide for details on explicit interfaces when calling other languages

Defining Explicit Interfaces

Interface blocks define explicit interfaces for external or dummy procedures. They can also be used to define a generic name for procedures, a new operator for functions, and a new form of assignment for subroutines.

An interface block takes the following form:

INTERFACE [*generic-spec*]

[*interface-body*]...
[MODULE PROCEDURE *name-list*]...
END INTERFACE [*generic-spec*]

generic-spec

Is one of the following:

- A generic name
- OPERATOR (*op*)

  Defines a generic operator (*op*). It can be a defined unary, defined binary, or extended intrinsic operator.
- ASSIGNMENT (=)

  Defines generic assignment.

interface-body

Is one or more function or subroutine subprograms. A function must end with END FUNCTION and a subroutine must end with END SUBROUTINE.

The subprogram must not contain a statement function or a DATA, ENTRY, or FORMAT statement; an entry name can be used as a procedure name.

The subprogram can contain a USE statement.

name-list

Is the name of one or more module procedures that are accessible in the host. The MODULE PROCEDURE statement is only allowed if the interface block specifies a *generic-spec* and has a host that is a module (or accesses a module by use association).

The characteristics of module procedures are not given in interface blocks, but are assumed from the module subprogram definitions.

Rules and Behavior

Interface blocks can appear in the specification part of the program unit that invokes the external or dummy procedure.

A *generic-spec* can only appear in the END INTERFACE statement (a Fortran 95 feature) if one appears in the INTERFACE statement; they must be identical.

The characteristics specified for the external or dummy procedure must be consistent with those specified in the procedure's definition.

An interface block must not appear in a block data program unit.

An interface block comprises its own scoping unit, and does not inherit anything from its host through host association.

A procedure must not have more than one explicit interface in a given scoping unit.

A interface block containing *generic-spec* specifies a generic interface for the following procedures:

- The procedures within the interface block

  Any generic name, defined operator, or equals symbol that appears is a generic identifier for all the procedures in the interface block. For the rules on how any two procedures with the same generic identifier must differ, see "Unambiguous Generic Procedure References".
- The module procedures listed in the MODULE PROCEDURE statement

  The module procedures must be accessible by a USE statement.

To make an interface block available to multiple program units (through a USE statement), place the interface block in a module.

The following rules apply to interface blocks containing pure procedures:

- The interface specification of a pure procedure must declare the INTENT of all dummy arguments except pointer and procedure arguments.
- A procedure that is declared pure in its definition can also be declared pure in an interface block. However, if it is not declared pure in its definition, it must not be declared pure in an interface block.

Examples

The following example shows a simple procedure interface block with no generic specification:

```
SUBROUTINE SUB_B (B, FB)
  REAL B
  ...
  INTERFACE
    FUNCTION FB (GN)
      REAL FB, GN
    END FUNCTION
  END INTERFACE
```

See Also

- "Functions"
- "Subroutines"
- "Use and Host Association"
- "Modules and Module Procedures"
- "Pure Procedures"
- "Determining When Procedures Require Explicit Interfaces"

- "Defining Generic Names for Procedures" for details on user-defined generic procedures
- "Defining Generic Operators" for details on defined operators
- "Defining Generic Assignment" for details on defined assignment
- Your user's guide for details on when you should not use interface blocks

Defining Generic Names for Procedures

An interface block can be used to specify a generic name to reference all of the procedures within the interface block.

The initial line for such an interface block takes the following form:

INTERFACE *generic-name*

generic-name

Is the generic name. It can be the same as any of the procedure names in the interface block, or the same as any accessible generic name (including a generic intrinsic name).

This kind of interface block can be used to extend or redefine a generic intrinsic procedure.

The procedures that are given the generic name must be the same kind of subprogram: all must be functions, or all must be subroutines.

Any procedure reference involving a generic procedure name must be resolvable to one specific procedure; it must be unambiguous. For more information, see "Unambiguous Generic Procedure References".

The following is an example of a procedure interface block defining a generic name:

```
INTERFACE GROUP_SUBS
  SUBROUTINE INTEGER_SUB (A, B)
    INTEGER, INTENT(INOUT) :: A, B
  END SUBROUTINE INTEGER_SUB
  SUBROUTINE REAL_SUB (A, B)
    REAL, INTENT(INOUT) :: A, B
  END SUBROUTINE REAL_SUB

  SUBROUTINE COMPLEX_SUB (A, B)
    COMPLEX, INTENT(INOUT) :: A, B
  END SUBROUTINE COMPLEX_SUB
END INTERFACE
```

The three subroutines can be referenced by their individual specific names or by the group name GROUP_SUBS.

The following example shows a reference to INTEGER_SUB:

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

See Also

"Defining Explicit Interfaces" for details on interface blocks

Defining Generic Operators

An interface block can be used to define a generic operator. The only procedures allowed in the interface block are functions that can be referenced as defined operations.

The initial line for such an interface block takes the following form:

INTERFACE OPERATOR (*op*)

op

Is one of the following:

- A defined unary operator (one argument)
- A defined binary operator (two arguments)
- An extended intrinsic operator (number of arguments must be consistent with the intrinsic uses of that operator)

The functions within the interface block must have one or two nonoptional arguments with intent IN, and the function result must not be of type character with assumed length. A defined operation is treated as a reference to the function.

The following shows the form (and an example) of a defined unary and defined binary operation:

Operation	Form	Example
Defined unary	.defined-operator. operand[1]	.MINUS. C
Defined binary	operand[2] .defined-operator. operand[3]	B .MINUS. C

1. The operand corresponds to the function's dummy argument.
2. The left operand corresponds to the first dummy argument of the function.
3. The right operand corresponds to the second argument.

For intrinsic operator symbols, the generic properties include the intrinsic operations they represent. Both forms of each relational operator have the same interpretation, so extending one form (such as >=) defines both forms (>= and .GE.).

The following is an example of a procedure interface block defining a new operator:

```
INTERFACE OPERATOR(.BAR.)
  FUNCTION BAR(A_1)
    INTEGER, INTENT(IN) :: A_1
```

```
    INTEGER :: BAR
  END FUNCTION BAR
END INTERFACE
```

The following example shows a way to reference function BAR by using the new operator:

```
INTEGER B
I = 4 + (.BAR. B)
```

The following is an example of a procedure interface block with a defined operator extending an existing operator:

```
INTERFACE OPERATOR(+)
  FUNCTION LGFUNC (A, B)
    LOGICAL, INTENT(IN) :: A(:), B(SIZE(A))
    LOGICAL :: LGFUNC(SIZE(A))
  END FUNCTION LGFUNC
END INTERFACE
```

The following example shows two equivalent ways to reference function LGFUNC:

```
LOGICAL, DIMENSION(1:10) :: C, D, E
N = 10
E = LGFUNC(C(1:N), D(1:N))
E = C(1:N) + D(1:N)
```

See Also

- "Defining Explicit Interfaces" for details on interface blocks
- "Expressions" for details on intrinsic operators
- "Defined Operations" for details on defined operators and operations
- "INTENT Attribute and Statement" for details on intent

Defining Generic Assignment

An interface block can be used to define generic assignment. The only procedures allowed in the interface block are subroutines that can be referenced as defined assignments.

The initial line for such an interface block takes the following form:

INTERFACE ASSIGNMENT (=)

The subroutines within the interface block *must* have two arguments, the first with intent OUT or INOUT, and the second with intent IN.

A defined assignment is treated as a reference to a subroutine. The left side of the assignment corresponds to the first dummy argument of the subroutine; the right side of the assignment corresponds to the second argument.

The ASSIGNMENT keyword extends or redefines an assignment operation if both sides of the equal sign are of the same derived type.

Defined elemental assignment is indicated by specifying ELEMENTAL in the SUBROUTINE statement.

Any procedure reference involving generic assignment must be resolvable to one specific procedure; it must be unambiguous. For more information, see "Unambiguous Generic Procedure References".

The following is an example of a procedure interface block defining assignment:

```
INTERFACE ASSIGNMENT (=)
  SUBROUTINE BIT_TO_NUMERIC (NUM, BIT)
    INTEGER, INTENT(OUT) :: NUM
    LOGICAL, INTENT(IN)  :: BIT(:)
  END SUBROUTINE BIT_TO_NUMERIC

  SUBROUTINE CHAR_TO_STRING (STR, CHAR)
    USE STRING_MODULE                     ! Contains definition of type STRING
    TYPE(STRING), INTENT(OUT) :: STR      ! A variable-length string
    CHARACTER(*), INTENT(IN)  :: CHAR
  END SUBROUTINE  CHAR_TO_STRING
END  INTERFACE
```

The following example shows two equivalent ways to reference subroutine BIT_TO_NUMERIC:

```
CALL BIT_TO_NUMERIC(X, (NUM(I:J)))
X = NUM(I:J)
```

The following example shows two equivalent ways to reference subroutine CHAR_TO_STRING:

```
CALL CHAR_TO_STRING(CH, '432C')
CH = '432C'
```

See Also

- "Defined Assignments"
- "Defining Explicit Interfaces" for details on interface blocks
- "INTENT Attribute and Statement" for details on intent

CONTAINS Statement

A CONTAINS statement separates the body of a main program, module, or external subprogram from any internal or module procedures it may contain. It is not executable.

The CONTAINS statement takes the following form:

CONTAINS

Any number of internal procedures can follow a CONTAINS statement, but a CONTAINS statement cannot appear in the internal procedures themselves.

See Also

- "Modules and Module Procedures"
- "Internal Procedures"

ENTRY Statement

The ENTRY statement provides one or more entry points within a subprogram. It is not executable and must precede any CONTAINS statement (if any) within the subprogram.

The ENTRY statement takes the following form:

ENTRY *name* [([*d-arg* [, *d-arg*]...]) [RESULT (*r-name*)]]

name

Is the name of an entry point. If RESULT is specified, this entry name must not appear in any specification statement in the scoping unit of the function subprogram.

d-arg

Is a dummy argument. The dummy argument can be an alternate return indicator (*) if the ENTRY statement is within a subroutine subprogram.

r-name

Is the name of a function result. This name must not be the same as the name of the entry point, or the name of any other function or function result. This parameter can only be specified for function subprograms.

Rules and Behavior

ENTRY statements can only appear in external procedures or module procedures.

An ENTRY statement must not appear in a CASE, DO, IF, FORALL, or WHERE construct, or a nonblock DO loop.

When the ENTRY statement appears in a subroutine subprogram, it is referenced by a CALL statement. When the ENTRY statement appears in a function subprogram, it is referenced by a function reference.

An entry name within a function subprogram can appear in a type declaration statement.

Within the subprogram containing the ENTRY statement, the entry name must not appear as a dummy argument in the FUNCTION or SUBROUTINE statement, and it must not appear in an EXTERNAL or INTRINSIC statement. For example, neither of the following are valid:

```
(1)  SUBROUTINE SUB(E)
     ENTRY E
     ...
(2)  SUBROUTINE SUB
     EXTERNAL E
     ENTRY E
     ...
```

An ENTRY statement can reference itself if the function or subroutine subprogram was defined as RECURSIVE.

Dummy arguments can be used in ENTRY statements even if they differ in order, number, type and kind parameters, and name from the dummy arguments used in the FUNCTION, SUBROUTINE, and other ENTRY statements in the same subprogram. However, each reference to a function, subroutine, or entry must use an actual argument list that agrees in order, number, and type with the dummy argument list in the corresponding FUNCTION, SUBROUTINE, or ENTRY statement.

Dummy arguments can be referred to only in executable statements that follow the first SUBROUTINE, FUNCTION, or ENTRY statement in which the dummy argument is specified. If a dummy argument is not currently associated with an actual argument, the dummy argument is undefined and cannot be referenced. Arguments do not retain their association from one reference of a subprogram to another.

For specific information on ENTRY statements in function subprograms and subroutine subprograms (including examples), see "ENTRY Statements in Function Subprograms" and "ENTRY Statements in Subroutine Subprograms".

See Also

- "Functions"
- "Subroutines"
- "Function References"
- "CALL Statement"
- "Argument Association" for details on procedure arguments

ENTRY Statements in Function Subprograms

If the ENTRY statement is contained in a function subprogram, it defines an additional function. The name of the function is the name specified in the ENTRY statement, and its result variable is the entry name or the name specified by RESULT (if any).

If the entry result variable has the same characteristics as the FUNCTION statement's result variable, their result variables identify the same variable, even if they have different names. Otherwise, the result variables are storage associated and must all be nonpointer scalars of intrinsic type, in one of the following groups:

Group 1	Type default integer, default real, double precision real, default complex, double complex, or default logical
Group 2	Type REAL(16) and COMPLEX(16)
Group 3	Type default character (with identical lengths)

All entry names within a function subprogram are associated with the name of the function subprogram. Therefore, defining any entry name or the name of the function subprogram defines all the associated names with the same data type. All associated names with different data types become undefined.

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

Examples

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

```
REAL FUNCTION TANH(X)
  TSINH(Y) = EXP(Y) - EXP(-Y)
  TCOSH(Y) = EXP(Y) + EXP(-Y)

  TANH = TSINH(X)/TCOSH(X)
  RETURN

ENTRY SINH(X)
  SINH = TSINH(X)/2.0
  RETURN
  ENTRY COSH(X)
  COSH = TCOSH(X)/2.0
  RETURN
END
```

See Also

"RESULT Keyword"

ENTRY Statements in Subroutine Subprograms

If the ENTRY statement is contained in a subroutine subprogram, it defines an additional subroutine. The name of the subroutine is the name specified in the ENTRY statement.

If RECURSIVE is specified on the SUBROUTINE statement, the interface of the subroutine defined by the ENTRY statement is explicit within the subroutine subprogram.

Examples

The following example shows a main program calling a subroutine containing an ENTRY statement:

```
PROGRAM TEST
  ...
  CALL SUBA(A, B, C)        ! A, B, and C are actual arguments
  ...                       !   passed to entry point SUBA
END
SUBROUTINE SUB(X, Y, Z)
  ...
  ENTRY SUBA(Q, R, S)       ! Q, R, and S are dummy arguments
  ...                       ! Execution starts with this statement
END SUBROUTINE
```

The following example shows an ENTRY statement specifying alternate returns:

```
CALL SUBC(M, N, *100, *200, P)
SUBROUTINE SUB(K, *, *)
  ...
  ENTRY SUBC(J, K, *, *, X)
  ...
  RETURN 1
  RETURN 2
END
```

Note that the CALL statement for entry point SUBC includes actual alternate return arguments. The RETURN 1 statement transfers control to statement label 100 and the RETURN 2 statement transfers control to statement label 200 in the calling program.

See Also

Your user's guide for details on implementation of argument association in ENTRY statements

Intrinsic Procedures

9

Intrinsic procedures are functions and subroutines that are included in the Fortran 95/90 library. There are four classes of these intrinsic procedures, as follows:

- Elemental procedures

  These procedures have scalar dummy arguments that can be called with scalar or array actual arguments. There are many elemental intrinsic functions and one elemental intrinsic subroutine (MVBITS).

  If the arguments are all scalar, the result is scalar. If an actual argument is array-valued, the intrinsic procedure is applied to each element of the actual argument, resulting in an array that has the same shape as the actual argument.

  If there is more than one array-valued argument, they must all have the same shape.
- Inquiry functions

  These functions have results that depend on the properties of their principal argument, not the value of the argument (the argument value can be undefined).
- Transformational functions

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

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

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

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

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

When an intrinsic function is passed as an actual argument to a procedure, its specific name must be used, and when called, its arguments must be scalar. Some specific intrinsic functions are not allowed as actual arguments in all circumstances. Table 9-1 lists specific functions that cannot be passed as actual arguments.

Table 9-1 Functions Not Allowed as Actual Arguments

AIMAX0	EOF	INT8	LGE
AIMIN0	FLOAT	INT_PTR_KIND	LGT
AJMAX0	FLOATI	IQINT	LLE
AJMIN0	FLOATJ	IZEXT	LLT
AKMAX0	FLOATK	JFIX	LOC
AKMIN0	HFIX	JIDINT	MALLOC
AMAX0	IADDR	JIFIX	MAX0
AMAX1	IARGC	JINT	MAX1
AMIN0	ICHAR	JIQINT	MIN0
AMIN1	IDINT	JMAX0	MIN1
BADDRESS	IFIX	JMAX1	MULT_HIGH
CACHESIZE	IIDINT	JMIN0	NARGS
CHAR	IIFIX	JMIN1	QCMPLX
CMPLX	IINT	JZEXT	QEXT
DBLE	IIQINT	KIDINT	QEXTD
DBLEQ	IJINT	KIFIX	QMAX1
DCMPLX	IMAX0	KINT	QMIN1
DFLOTI	IMAX1	KIQINT	QREAL
DFLOTJ	IMIN0	KIQNNT	RAN
DFLOTK	IMIN1	KMAX0	REAL
DMAX1	INT	KMAX1	SECNDS
DMIN1	INT1	KMIN0	SNGL
DPROD	INT2	KMIN1	SNGLQ
DREAL	INT4	KZEXT	ZEXT

This chapter contains information on the following topics:

- "Argument Keywords in Intrinsic Procedures"
- "Overview of Intrinsic Procedures"
- "Descriptions of Intrinsic Procedures"

See Also

- "Argument Association"
- "MERGE"
- "Optional Arguments"
- Appendix D, "Data Representation Models"
- "References to Generic Intrinsic Functions"
- "References to Elemental Intrinsic Procedures" for details on elemental references to intrinsic procedures
- Your user's guide for details on Intel® Fortran numeric data format

Argument Keywords in Intrinsic Procedures

For all intrinsic procedures, the arguments shown are the names you must use as keywords when using the keyword form for actual arguments. For example, a reference to function CMPLX (X, Y, KIND) can be written as follows:

Using positional arguments:	CMPLX (F, G, L)
Using argument keywords:	CMPLX (KIND=L, Y=G, X=F)[1]

1. Note that argument keywords can be written in any order.

Some argument keywords are optional (denoted by square brackets). The following describes some of the most commonly used optional arguments:

BACK	Specifies that a string scan is to be in reverse order (right to left).
DIM	Specifies a selected dimension of an array argument.
KIND	Specifies the kind type parameter of the function result.
MASK	Specifies that a mask can be applied to the elements of the argument array to exclude the elements that are not to be involved in an operation.

Examples

The syntax for the DATE_AND_TIME intrinsic subroutine shows four optional positional arguments: DATE, TIME, ZONE, and VALUES (see "DATE_AND_TIME").

The following shows some valid ways to specify these arguments:

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

See Also

- "Argument Association"
- "CALL Statement" for details on argument keywords in subroutine references
- "Function References" for details on argument keywords in function references

Overview of Intrinsic Procedures

This section describes the categories of generic intrinsic functions (including a summarizing table), lists the intrinsic subroutines, and provides general information on bit functions.

Intrinsic procedures are fully described (in alphabetical order) in "Descriptions of Intrinsic Procedures".

Categories of Intrinsic Functions

Generic intrinsic functions can be divided into categories, as shown in Table 9-2.

Table 9-2 Categories of Intrinsic Functions

Category	Subcategory	Description
Numeric	Computation	Perform type conversions or simple numeric operations: ABS, AIMAG, AINT, AMAX0, AMIN0, ANINT, CEILING, CMPLX, CONJG, DBLE, DCMPLX, DFLOAT, DIM, DNUM, DPROD, DREAL, FLOAT, FLOOR, IFIX, ILEN, IMAG, INT, INUM, JNUM, MAX, MAX1, MIN, MIN1, MOD, MODULO, NINT, QCMPLX, QEXT, QFLOAT, QNUM, QREAL, RAN, REAL, RNUM, SIGN, SNGL, ZEXT
	Manipulation[1]	Return values related to the components of the model values associated with the actual value of the argument: EXPONENT, FRACTION, NEAREST, RRSPACING, SCALE, SET_EXPONENT, SPACING
	Inquiry[1]	Return scalar values from the models associated with the type and kind parameters of their arguments[2]: DIGITS, EPSILON, HUGE, MAXEXPONENT, MINEXPONENT, PRECISION, RADIX, RANGE, SIZEOF, TINY

Table 9-2 Categories of Intrinsic Functions

Category	Subcategory	Description
	Transformational	Perform vector and matrix multiplication: DOT_PRODUCT, MATMUL
	System	Return information about a process or processor: MCLOCK, SECNDS
Kind type		Return kind type parameters: SELECTED_INT_KIND, SELECTED_REAL_KIND, KIND
Mathematical		Perform mathematical operations: ACOS, ACOSD, ACOSH, ASIN, ASIND, ASINH, ATAN, ATAN2, ATAN2D, ATAND, ATANH, COS, COSD, COSH, COTAN, COTAND, EXP, LOG, LOG10, SIN, SIND, SINH, SQRT, TAN, TAND, TANH
Bit	Manipulation	Perform single-bit processing, logical and shift operations, and allow bit subfields to be referenced: AND, BTEST, DSHIFTL, DSHIFTR, IAND, IBCHNG, IBCLR, IBITS, IBSET, IEOR, IOR, ISHA, ISHC, ISHFT, ISHFTC, ISHL, IXOR, LSHIFT (or LSHFT), NOT, OR, RSHIFT (or RSHFT), SHIFTL, SHIFTR, XOR
	Inquiry	Lets you determine parameter *s* (the bit size) in the bit model[3]: BIT_SIZE
	Representation	Return information on bit representation of integers: LEADZ, POPCNT, POPPAR, TRAILZ
Character	Comparison	Lexically compare character-string arguments and return a default logical result: LGE, LGT, LLE, LLT
	Conversion	Convert character arguments to integer, ASCII, or character values[4]: ACHAR, CHAR, IACHAR, ICHAR
	String handling	Perform operations on character strings, return lengths of arguments, and search for certain arguments: ADJUSTL, ADJUSTR, INDEX, LEN_TRIM, REPEAT, SCAN, TRIM, VERIFY
	Inquiry	Returns the length of an argument or information about command-line arguments: IARG, IARGC, LEN, NARGS, NUMARG
Array	Construction	Construct new arrays from the elements of existing array: MERGE, PACK, SPREAD, UNPACK
	Inquiry	Let you determine if an array argument is allocated, and return the size or shape of an array, and the lower and upper bounds of subscripts along each dimension: ALLOCATED, LBOUND, SHAPE, SIZE, UBOUND
	Location	Returns the geometric locations of the maximum and minimum values of an array: MAXLOC, MINLOC
	Manipulation	Let you shift an array, transpose an array, or change the shape of an array: CSHIFT, EOSHIFT, RESHAPE, TRANSPOSE

Table 9-2 Categories of Intrinsic Functions

Category	Subcategory	Description
	Reduction	Perform operations on arrays. The functions "reduce" elements of a whole array to produce a scalar result, or they can be applied to a specific dimension of an array to produce a result array with a rank reduced by one: ALL, ANY, COUNT, MAXVAL, MINVAL, PRODUCT
Miscellaneous		Do the following: • Check for pointer association (ASSOCIATED) • Return an address (BADDRESS) • Return the size of a level of the memory cache (CACHESIZE) • Check for end-of-file (EOF) • Return error functions (ERF and ERFC) • Return the class of a floating-point argument (FP_CLASS) • Return a pointer to an actual argument list for a routine (IARGPTR) • Return the INTEGER KIND that will hold an address (INT_PTR_KIND) • Test for Not-a-Number values (ISNAN) • Return the internal address of a storage item (LOC) • Return a logical value of an argument (LOGICAL) • Allocate memory (MALLOC) • Return the upper 64 bits of a 128-bit unsigned result (MULT_HIGH) • Return a disassociated pointer (NULL) • Check for argument presence (PRESENT) • Convert a bit pattern (TRANSFER)

1. All of the numeric manipulation and many of the numeric inquiry functions are defined by the model sets for integers (“Model for Integer Data”) and reals (“Model for Real Data”).
2. The value of the argument does not have to be defined.
3. For more information on bit functions, see “Bit Functions”.
4. The Intel Fortran processor character set is ASCII, so ACHAR = CHAR and IACHAR = ICHAR.

Table 9-3 summarizes the generic intrinsic functions and indicates whether they are elemental, inquiry, or transformational functions. Optional arguments are shown within square brackets.

Table 9-3 Summary of Generic Intrinsic Functions

Generic Function	Class[1]	Value Returned
ABS (a)	E	The absolute value of an argument
ACHAR (i)	E	The character in the specified position of the ASCII character set
ACOS (x)	E	The arccosine (in radians) of the argument
ACOSD (x)	E	The arccosine (in degrees) of the argument
ACOSH (x)	E	The hyperbolic arccosine of the argument
ADJUSTL (string)	E	The specified string with leading blanks removed and placed at the end of the string
ADJUSTR (string)	E	The specified string with trailing blanks removed and placed at the beginning of the string
AIMAG (z)	E	The imaginary part of a complex argument
AINT (a [, kind])	E	A real value truncated to a whole number
ALL (mask [, dim])	T	.TRUE. if all elements of the masked array are true
ALLOCATED (array)	I	The allocation status of the argument array
AMAX0 (a1, a2 [, a3,...])	E	The maximum value in a list of integers (returned as a real value)
AMIN0 (a1, a2 [, a3,...])	E	The minimum value in a list of integers (returned as a real value)
AND (i, j)	E	See IAND
ANINT (a [, kind])	E	A real value rounded to a whole number
ANY (mask [, dim])	T	.TRUE. if any elements of the masked array are true
ASIN (x)	E	The arcsine (in radians) of the argument
ASIND (x)	E	The arcsine (in degrees) of the argument
ASINH (x)	E	The hyperbolic arcsine of the argument
ASSOCIATED (pointer [, target])	I	.TRUE. if the pointer argument is associated or the pointer is associated with the specified target
ATAN (x)	E	The arctangent (in radians) of the argument
ATAN2 (y, x)	E	The arctangent (in radians) of the arguments
ATAN2D (y, x)	E	The arctangent (in degrees) of the arguments
ATAND (x)	E	The arctangent (in degrees) of the argument
ATANH (x)	E	The hyperbolic arctangent of the argument
BADDRESS (x)	I	The address of the argument

Table 9-3 Summary of Generic Intrinsic Functions

Generic Function	Class[1]	Value Returned
BIT_SIZE (i)	I	The number of bits (s) in the bit model
BTEST (i, pos)	E	.TRUE. if the specified position of argument I is one
CEILING (a [, kind])	E	The smallest integer greater than or equal to the argument value
CHAR (i [, kind])	E	The character in the specified position of the processor character set
CONJG (z)	E	The conjugate of a complex number
COS (x)	E	The cosine of the argument, which is in radians
COSD (x)	E	The cosine of the argument, which is in degrees
COSH (x)	E	The hyperbolic cosine of the argument
COTAN (x)	E	The cotangent of the argument, which is in radians
COTAND (x)	E	The cotangent of the argument, which is in degrees
COUNT (mask [, dim] [, kind])	T	The number of .TRUE. elements in the argument array
CSHIFT (array, shift [, dim])	T	An array that has the elements of the argument array circularly shifted
DBLE (a)	E	The corresponding double precision value of the argument
DFLOAT (a)	E	The corresponding double precision value of the integer argument
DIGITS (x)	I	The number of significant digits in the model for the argument
DIM (x, y)	E	The positive difference between the two arguments
DOT_PRODUCT (vector_a, vector_b)	T	The dot product of two rank-one arrays (also called a vector multiply function)
DSHIFTL (ileft, iright, ishift)	E	The upper (leftmost) 64 bits of a left-shifted 128-bit integer
DSHIFTR (ileft, iright, ishift)	E	The lower (rightmost) 64 bits of a right-shifted 128-bit integer
EOSHIFT (array, shift [, boundary] [, dim])	T	An array that has the elements of the argument array end-off shifted
EOF (a)	I	.TRUE. or .FALSE. depending on whether a file is beyond the end-of-file record
EPSILON (x)	I	The number that is almost negligible when compared to one
ERF (x)	E	The error function of an argument

Table 9-3 Summary of Generic Intrinsic Functions

Generic Function	Class[1]	Value Returned
ERFC (x)	E	The complementary error function of an argument
EXP (x)	E	The exponential e^x for the argument x
EXPONENT (x)	E	The value of the exponent part of a real argument
FLOAT (x)	E	The corresponding real value of the integer argument
FLOOR (a [, kind])	E	The largest integer less than or equal to the argument value
FP_CLASS (x)	E	The class of the IEEE floating-point argument
FRACTION (x)	E	The fractional part of a real argument
HUGE (x)	I	The largest number in the model for the argument
IACHAR (c)	E	The position of the specified character in the ASCII character set
IADDR (x)	E	See BADDRESS.
IAND (i, j)	E	The logical AND of the two arguments
IBCHNG (i, pos)	E	The reversed value of a specified bit
IBCLR (i, pos)	E	The specified position of argument I cleared (set to zero)
IBITS (i, pos, len)	E	The specified substring of bits of argument I
IBSET (i, pos)	E	The specified bit in argument I set to one
ICHAR (c [, kind])	E	The position of the specified character in the processor character set
IEOR (i, j)	E	The logical exclusive OR of the corresponding bit arguments
IFIX (x)	E	The corresponding integer value of the real argument rounded as if it were an implied conversion in an assignment
ILEN (i)	I	The length (in bits) in the two's complement representation of an integer
IMAG (z)	E	See AIMAG
INDEX (string, substring [, back] [, kind])	E	The position of the specified substring in a character expression
INT (a [, kind])	E	The corresponding integer value (truncated) of the argument
IOR (i, j)	E	The logical inclusive OR of the corresponding bit arguments

Table 9-3 Summary of Generic Intrinsic Functions

Generic Function	Class[1]	Value Returned
ISHA (i, shift)	E	Argument I shifted left or right by a specified number of bits
ISHC (i, shift)	E	Argument I rotated left or right by a specified number of bits
ISHFT (i, shift)	E	The logical end-off shift of the bits in argument I
ISHFTC (i, shift [, size])	E	The logical circular shift of the bits in argument I
ISHL (i, shift)	E	Argument I logically shifted left or right by a specified number of bits
ISNAN (x)	E	Tests for Not-a-Number (NaN) values
IXOR (i, j)	E	See IEOR
KIND (x)	I	The kind type parameter of the argument
LBOUND (array [, dim] [, kind])	I	The lower bounds of an array (or one of its dimensions)
LEADZ (i)	E	The number of leading zero bits in an integer.
LEN (string [, kind])	I	The length (number of characters) of the argument character string
LEN_TRIM (string [, kind])	E	The length of the specified string without trailing blanks
LGE (string_a, string_b)	E	A logical value determined by a > or = comparison of the arguments
LGT (string_a, string_b	E	A logical value determined by a > comparison of the arguments
LLE (string_a, string_b)	E	A logical value determined by a < or = comparison of the arguments
LLT (string_a, string_b)	E	A logical value determined by a < comparison of the arguments
LOC (a)	I	The internal address of the argument.
LOG (x)	E	The natural logarithm of the argument
LOG10 (x)	E	The common logarithm (base 10) of the argument
LOGICAL (l [, kind])	E	The logical value of the argument converted to a logical of type KIND
LSHIFT (i, positive_shift) [2]	E	Can also be specified as LSHFT; see ISHFT
MATMUL (matrix_a, matrix_b)	T	The result of matrix multiplication (also called a matrix multiply function)
MAX (a1, a2 [, a3,...])	E	The maximum value in the set of arguments

Table 9-3 Summary of Generic Intrinsic Functions

Generic Function	Class[1]	Value Returned
MAX1 (a1, a2 [, a3,...])	E	The maximum value in the set of real arguments (returned as an integer)
MAXEXPONENT (x)	I	The maximum exponent in the model for the argument
MAXLOC (array [, dim] [, mask] [, kind])	T	The rank-one array that has the location of the maximum element in the argument array
MAXVAL (array [, dim] [, mask])	T	The maximum value of the elements in the argument array
MERGE (tsource, fsource, mask)	E	An array that is the combination of two conformable arrays (under a mask)
MIN (a1, a2 [, a3,...])	E	The minimum value in the set of arguments
MIN1 (a1, a2 [, a3,...])	E	The minimum value in the set of real arguments (returned as an integer)
MINEXPONENT (x)	I	The minimum exponent in the model for the argument
MINLOC (array [, dim] [, mask] [, kind])	T	The rank-one array that has the location of the minimum element in the argument array
MINVAL (array [, dim] [, mask])	T	The minimum value of the elements in the argument array
MOD (a, p)	E	The remainder of the arguments (has the sign of the first argument)
MODULO (a, p)	E	The modulo of the arguments (has the sign of the second argument)
NEAREST (x, s)	E	The nearest different machine-representable number in a given direction
NINT (a [, kind])	E	A real value rounded to the nearest integer
NOT (i)	E	The logical complement of the argument
NULL ([mold])	T	A disassociated pointer
OR (i, j)	E	See IOR
PACK (array, mask [, vector])	T	A packed array of rank one (under a mask)
POPCNT (i)	E	The number of 1 bits in the integer argument
POPPAR (i)	E	The parity of the integer argument
PRECISION (x)	I	The decimal precision (real or complex) of the argument
PRESENT (a)	I	.TRUE. if an actual argument has been provided for an optional dummy argument
PRODUCT (array [, dim] [, mask])	T	The product of the elements of the argument array

Table 9-3 Summary of Generic Intrinsic Functions

Generic Function	Class[1]	Value Returned
QEXT (a)	E	The corresponding REAL(16) precision value of the argument.
QFLOAT (a)	E	The corresponding REAL(16) precision value of the integer argument.
RADIX (x)	I	The base of the model for the argument
RANGE (x)	I	The decimal exponent range of the model for the argument
REAL (a [, kind])	E	The corresponding real value of the argument
REPEAT (string, ncopies)	T	The concatenation of zero or more copies of the specified string
RESHAPE (source, shape [, pad] [, order])	T	An array that has a different shape than the argument array, but the same elements
RRSPACING (x)	E	The reciprocal of the relative spacing near the argument
RSHIFT (i, negative_shift) [3]	E	Can also be specified as RSHFT; see ISHFT
SCALE (x, I)	E	The value of the exponent part (of the model for the argument) changed by a specified value
SCAN (string, SET [, back] [, kind])	E	The position of the specified character (or set of characters) within a string
SELECTED_INT_KIND (r)	T	The integer kind parameter of the argument
SELECTED_REAL_KIND ([p] [, r])	T	The real kind parameter of the argument; one of the optional arguments must be specified
SET_EXPONENT (x, i)	E	The value of the exponent part (of the model for the argument) set to a specified value
SHAPE (SOURCE [, kind])	I	The shape (rank and extents) of an array or scalar
SHIFTL (ivalue, ishift)	E	Argument "ivalue" shifted left by a specified number of bits
SHIFTR (ivalue, ishift)	E	Argument "ivalue" shifted right by a specified number of bits
SIGN (a, b)	E	A value with the sign transferred from its second argument
SIN (x)	E	The sine of the argument, which is in radians
SIND (x)	E	The sine of the argument, which is in degrees
SINH (x)	E	The hyperbolic sine of the argument

Table 9-3 Summary of Generic Intrinsic Functions

Generic Function	Class[1]	Value Returned
SIZE (array [, dim] [, kind])	I	The size (total number of elements) of the argument array (or one of its dimensions)
SIZEOF(x)	I	The bytes of storage used by an argument
SNGL (x)	E	The corresponding real value of the argument
SPACING (x)	E	The value of the absolute spacing of model numbers near the argument
SPREAD (source, dim, ncopies)	T	A replicated array that has an added dimension
SQRT (x)	E	The square root of the argument
SUM (array [, dim] [, mask])	T	The sum of the elements of the argument array
TAN (x)	E	The tangent of the argument, which is in radians
TAND (x)	E	The tangent of the argument, which is in degrees
TANH (x)	E	The hyperbolic tangent of the argument
TINY (x)	I	The smallest positive number in the model for the argument
TRAILZ (i)	E	The number of trailing zero bits in an integer.
TRANSFER (source, mold [, size])	T	The bit pattern of SOURCE converted to the type and kind parameters of MOLD
TRANSPOSE (matrix)	T	The matrix transpose for the rank-two argument array
TRIM (string)	T	The argument with trailing blanks removed
UBOUND (array [, dim] [, kind])	I	The upper bounds of an array (or one of its dimensions)
UNPACK (vector, mask, field)	T	An array (under a mask) unpacked from a rank-one array
VERIFY (string, set [, back] [, kind])	E	The position of the first character in a string that does not appear in the given set of characters
XOR (i, j)	E	See IEOR
ZEXT (x [, kind])	E	A zero-extended value of the argument

1. Key to Classes:
   E - Elemental
   I - Inquiry
   T - Transformational
2. Or LSHFT.
3. Or RSHFT.

Table 9-4 lists specific functions that have no generic function associated with them and indicates whether they are elemental, nonelemental, or inquiry functions. Optional arguments are shown within square brackets.

Table 9-4 Specific Functions with No Generic Association

Specific Function	Class[1]	Value Returned
CACHESIZE (n)[2]	I	The size of a level of the memory cache
CMPLX (x [, y] [, kind])	E	The corresponding complex value of the argument
DCMPLX (x, y)	E	The corresponding double complex value of the argument
DNUM (i)	E	The corresponding double-precision value of a character string
DPROD (x, y)	E	The higher precision product of two real arguments
DREAL (a)	E	The corresponding double-precision value of the real part of a double-complex argument
IARG ( )	I	See IARGC
IARGC ( )	I	The index of the last command-line argument
IARGPTR ( )	I	The count of actual arguments passed to the current routine
INUM (i)	E	The corresponding INTEGER(2) value of a character string
JNUM (i)	E	The corresponding INTEGER(4) value of a character string
MALLOC (i)	E	The starting address for the block of memory allocated
MCLOCK ( )	I	The sum (in units of microseconds) of the current process's user time and the user and system time of all its child processes
MULT_HIGH (i, j)[2]	E	The upper (leftmost) 64 bits of the 128-bit unsigned result.
NARGS ( )	I	The total number of command-line arguments, including the command
NUMARG ( )	I	See IARGC
QCMPLX (x, y)	E	The corresponding COMPLEX(16) value of the argument
QNUM (i)	E	The corresponding REAL(16) value of a character string
QREAL (a)	E	The corresponding REAL(16) value of the real part of a COMPLEX(16) argument
RAN (i)	N	The next number from a sequence of pseudorandom numbers (uniformly distributed in the range 0 to 1)
RNUM (i)	E	The corresponding REAL(4) value of a character string

Table 9-4 Specific Functions with No Generic Association

Specific Function	Class[1]	Value Returned
SECNDS (x)	E	The system time of day (or elapsed time) as a floating-point value in seconds

1. Key to Classes:
   E - Elemental
   I - Inquiry
   N - Nonelemental
2. i64 only

Intrinsic Subroutines

Table 9-5 lists the intrinsic subroutines. Optional arguments are shown within square brackets. All these subroutines are nonelemental except for MVBITS.

Table 9-5 Intrinsic Subroutines

Subroutine	Value Returned or Result
CPU_TIME (time)	The processor time in seconds
DATE (buf)	The ASCII representation of the current date (in dd-mmm-yy form)
DATE_AND_TIME ([date] [, time] [, zone] [, values])	Date and time information from the real-time clock
ERRSNS ([io_err] [, sys_err] [, stat] [, unit] [, cond])	Information about the most recently detected error condition
EXIT ([status])	Image exit status is optionally returned; the program is terminated, all files closed, and control is returned to the operating system
FREE (a)	Frees memory that is currently allocated
GETARG (n, buffer [,status])	The specified command-line argument (where the command itself is argument zero)
IDATE (i, j, k)	Three integer values representing the current month, day, and year
MM_PREFETCH (address [, hint] [, fault] [, exclusive])	Data from the specified address on one memory cache line
MVBITS (from, frompos, len, to, topos)[1]	A sequence of bits (bit field) is copied from one location to another
RANDOM_NUMBER (harvest)	A pseudorandom number taken from a sequence of pseudorandom numbers uniformly distributed within the range 0.0 to 1.0

Table 9-5 Intrinsic Subroutines

Subroutine	Value Returned or Result
RANDOM_SEED ([size] [, put] [, get])	Initializes or retrieves the pseudorandom number generator seed value
RANDU (i1, i2, x)	A pseudorandom number as a single-precision value (within the range 0.0 to 1.0)
SYSTEM_CLOCK ([count] [, count_rate] [, count_max])	Data from the processors real-time clock
TIME (buf)	The ASCII representation of the current time (in hh:mm:ss form)

1. An elemental subroutine

Bit Functions

Integer data types are represented internally in binary two's complement notation. Bit positions in the binary representation are numbered from right (least significant bit) to left (most significant bit); the rightmost bit position is numbered 0.

The intrinsic functions IAND, IOR, IEOR, and NOT operate on all of the bits of their argument (or arguments). Bit 0 of the result comes from applying the specified logical operation to bit 0 of the argument. Bit 1 of the result comes from applying the specified logical operation to bit 1 of the argument, and so on for all of the bits of the result.

The functions ISHFT and ISHFTC shift binary patterns.

The functions IBSET, IBCLR, BTEST, and IBITS and the subroutine MVBITS operate on bit fields.

A *bit field* is a contiguous group of bits within a binary pattern. Bit fields are specified by a starting bit position and a length. A bit field must be entirely contained in its source operand.

For example, the integer 47 is represented by the following:

Binary pattern: 0...0101111

Bit position: n...6543210
Where *n* is the number of bit positions in the numeric storage unit.

You can refer to the bit field contained in bits 3 through 6 by specifying a starting position of 3 and a length of 4.

Negative integers are represented in two's complement notation. For example, the integer –47 is represented by the following:

Binary pattern: 1...1010001

Bit position: n...6543210
Where *n* is the number of bit positions in the numeric storage unit.

The value of bit position n is as follows:

```
1 for a negative number
0 for a non-negative number
```

All the high-order bits in the pattern from the last significant bit of the value up to bit n are the same as bit n.

IBITS and MVBITS operate on general bit fields. Both the starting position of a bit field and its length are arguments to these intrinsics. IBSET, IBCLR, and BTEST operate on 1-bit fields. They do not require a length argument.

For IBSET, IBCLR, and BTEST, the bit position range is as follows:

- 0 to 63 for INTEGER(8) and LOGICAL(8)
- 0 to 31 for INTEGER(4) and LOGICAL(4)
- 0 to 15 for INTEGER(2) and LOGICAL(2)
- 0 to 7 for BYTE, INTEGER(1), and LOGICAL(1)

For IBITS, the bit position can be any number. The length range is 0 to 63 on Intel® Itanium® processors; 0 to 31 on IA-32 processors.

The following example shows IBSET, IBCLR, and BTEST:

```
I = 4
J = IBSET (I,5)
PRINT *, 'J = ',J
K = IBCLR (J,2)
PRINT *, 'K = ',K
PRINT *, 'Bit 2 of K is ',BTEST(K,2)
END
```

The results are: J = 36, K = 32, and Bit 2 of K is F.

For optimum selection of performance and memory requirements, Intel Fortran provides the following integer data types:

Data Type	Storage Required (in bytes)
INTEGER(1)	1
INTEGER(2)	2
INTEGER(4)	4
INTEGER(8)	8

The bit manipulation functions each have a generic form that operates on all of these integer types and a specific form for each type.

When you specify the intrinsic functions that refer to bit positions or that shift binary patterns within a storage unit, be careful that you do not create a value that is outside the range of integers representable by the data type. If you shift by an amount greater than or equal to the size of the object you're shifting, the result is 0.

Consider the following:

```
INTEGER(2) I,J
I = 1
J = 17
I = ISHFT(I,J)
```

The variables I and J have INTEGER(2) type. Therefore, the generic function ISHFT maps to the specific function IISHFT, which returns an INTEGER(2) result. INTEGER(2) results must be in the range –32768 to 32767, but the value 1, shifted left 17 positions, yields the binary pattern 1 followed by 17 zeros, which represents the integer 131072. In this case, the result in I is 0.

The previous example would be valid if I was INTEGER(4), because ISHFT would then map to the specific function JISHFT, which returns an INTEGER(4) value.

If ISHFT is called with a constant first argument, the result will either be the default integer size or the smallest integer size that can contain the first argument, whichever is larger.

Descriptions of Intrinsic Procedures

This section contains detailed information on all generic and specific intrinsic procedures. These procedures are described in alphabetical order by generic name (if there is one).

Optional arguments are identified by square brackets in syntax and the label "(opt)" in descriptive text.

ABS

Description:	Computes an absolute value.
Syntax:	result = ABS (*a*)
Class:	Elemental function; Generic
Arguments:	*a* must be of type integer, real, or complex.
Results:	If *a* is an integer or real value, the value of the result is | *a* |; if *a* is a complex value (X, Y), the result is the real value SQRT (X**2 + Y**2).

Specific Name	Argument Type	Result Type
BABS	INTEGER(1)	INTEGER(1)
IIABS[1]	INTEGER(2)	INTEGER(2)
IABS[2]	INTEGER(4)	INTEGER(4)
KIABS	INTEGER(8)	INTEGER(8)
ABS	REAL(4)	REAL(4)
DABS	REAL(8)	REAL(8)
QABS	REAL(16)	REAL(16)
CABS[3]	COMPLEX(4)	REAL(4)
CDABS[4]	COMPLEX(8)	REAL(8)
CQABS	COMPLEX(16)	REAL(16)

1. Or HABS.
2. Or JIABS. For compatibility with older versions of Fortran, IABS can also be specified as a generic function.
3. The settings of compiler options specifying real size can affect CABS.
4. This function can also be specified as ZABS.

Examples

ABS (–7.4) has the value 7.4.

ABS ((6.0, 8.0)) has the value 10.0.

ACHAR

Description: Returns the character in a specified position of the ASCII character set, even if the processor's default character set is different. It is the inverse of the IACHAR function. In Intel Fortran, ACHAR is equivalent to the CHAR function.

Syntax: result = ACHAR (*i*)

Class: Elemental function; Generic

Arguments: *i* must be of type integer.

Results: The result type is character with length 1; it has the kind parameter value of KIND ('A').

If I has a value within the range 0 to 127, the result is the character in position I of the ASCII character set. ACHAR (IACHAR(C)) has the value C for any character C capable of representation in the processor.

Examples

ACHAR (71) has the value 'G'.

ACHAR (63) has the value '?'.

ACOS

Description: Produces the arccosine of *x*.

Syntax: result = ACOS (*x*)

Class: Elemental function; Generic

Arguments: *x* must be of type real. The | *x* | must be less than or equal to 1.

Results: The result type is the same as *x* and is expressed in radians. The value lies in the range 0 to π.

Specific Name	Argument Type	Result Type
ACOS	REAL(4)	REAL(4)
DACOS	REAL(8)	REAL(8)
QACOS	REAL(16)	REAL(16)

Example

ACOS (0.68032123) has the value .8225955.

ACOSD

Description: Produces the arccosine of *x*.

Syntax: result = ACOSD (*x*)

Class: Elemental function; Generic

Arguments: *x* must be of type real. The | *x* | must be less than or equal to 1.

Results: The result type is the same as *x* and is expressed in degrees. The value lies in the range –90 to 90 degrees.

Specific Name	Argument Type	Result Type
ACOSD	REAL(4)	REAL(4)
DACOSD	REAL(8)	REAL(8)
QACOSD	REAL(16)	REAL(16)

Example

ACOSD (0.886579) has the value 27.55354.

ACOSH

Description: Produces the hyperbolic arccosine of x.

Syntax: result = ACOSH (*x*)

Class: Elemental function; Generic

Arguments: *x* must be of type real and must be greater than or equal to 1.

Results: The result type is the same as x.

Specific Name	Argument Type	Result Type
ACOSH	REAL(4)	REAL(4)
DACOSH	REAL(8)	REAL(8)
QACOSH	REAL(16)	REAL(16)

Example

ACOSH (180.0) has the value 5.8861.

ADJUSTL

Description: Adjusts a character string to the left, removing leading blanks and inserting trailing blanks.

Syntax: result = ADJUSTL (*string*)

Class: Elemental function; Generic

Arguments: *string* must be of type character.

Results: The result type is character with the same length and kind parameter as *string*.

The value of the result is the same as *string*, except that any leading blanks have been removed and inserted as trailing blanks.

Example

ADJUSTL ('ΔΔΔΔ SUMMERTIME') has the value 'SUMMERTIMEΔΔΔΔ'.

ADJUSTR

Description: Adjusts a character string to the right, removing trailing blanks and inserting leading blanks.

Syntax: result = ADJUSTR (*string*)

Class: Elemental function; Generic

Arguments: *string* must be of type character.

Results: The result type is character with the same length and kind parameter as *string*.

The value of the result is the same as *string*, except that any trailing blanks have been removed and inserted as leading blanks.

Example

ADJUSTR ('SUMMERTIME∆∆∆∆') has the value '∆∆∆∆SUMMERTIME'.

AIMAG

Description: Returns the imaginary part of a complex number. This function can also be specified as IMAG.

Syntax: result = AIMAG (*z*)

Class: Elemental function; Generic

Arguments: *z* must be of type complex.

Results: The result type is real with the same kind parameter as *z*. If *z* has the value (x,y), the result has the value y.

Specific Name	Argument Type	Result Type
AIMAG[1]	COMPLEX(4)	REAL(4)
DIMAG	COMPLEX(8)	REAL(8)
QIMAG	COMPLEX(16)	REAL(16)

1. The setting of compiler options specifying real size can affect AIMAG.

Example

AIMAG ((4.0, 5.0)) has the value 5.0.

AINT

Description: Truncates a value to a whole number.

Syntax: result = AINT (*a* [, *kind*])

Class: Elemental function; Generic

Arguments:

a Must be of type real.

kind Must be a scalar integer initialization expression.

Results: The result is of type real. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter is that of *a*.

The result is defined as the largest integer whose magnitude does not exceed the magnitude of *a* and whose sign is the same as that of *a*. If | *a* | is less than 1, AINT(*a*) has the value zero.

Specific Name	Argument Type	Result Type
AINT	REAL(4)	REAL(4)
DINT	REAL(8)	REAL(8)
QINT	REAL(16)	REAL(16)

Examples

AINT (3.678) has the value 3.0.

AINT (–1.375) has the value –1.0.

ALL

Description: Determines if *all* values are true in an entire array or in a specified dimension of an array.

Syntax: result = ALL (*mask* [, *dim*])

Class: Transformational function; Generic

Arguments:

mask Must be a logical array.

dim (opt) Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of *mask*.

Results: The result is an array or a scalar of type logical.

The result is a scalar if *dim* is omitted or *mask* has rank one. A scalar result is true only if all elements of *mask* are true, or *mask* has size zero. The result has the value false if any element of *mask* is false.

An array result has the same type and kind parameters as *mask*, and a rank that is one less than *mask*. Its shape is (d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n), where (d_1, d_2,..., d_n) is the shape of *mask*.

Each element in an array result is true only if all elements in the one dimensional array defined by *mask* (s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n) are true.

Examples

ALL ((/.TRUE., .FALSE., .TRUE./)) has the value false because some elements of MASK are not true.

ALL ((/.TRUE., .TRUE., .TRUE./)) has the value true because *all* elements of MASK are true.

A is the array $\begin{bmatrix} 1 & 5 & 7 \\ 3 & 6 & 8 \end{bmatrix}$ and B is the array $\begin{bmatrix} 0 & 5 & 7 \\ 2 & 6 & 9 \end{bmatrix}$.

ALL (A .EQ. B, DIM=1) tests to see if all elements in each column of A are equal to the elements in the corresponding column of B. The result has the value (false, true, false) because only the second column has elements that are all equal.

ALL (A .EQ. B, DIM=2) tests to see if all elements in each row of A are equal to the elements in the corresponding row of B. The result has the value (false, false) because each row has some elements that are not equal.

ALLOCATED

Description: Indicates whether an allocatable array is currently allocated.

Syntax: result = ALLOCATED (*array*)

Class: Inquiry function; Generic

Arguments: *array* must be an allocatable array.

Results: The result is a scalar of type default logical.

The result has the value true if *array* is currently allocated, false if *array* is not currently allocated, or undefined if its allocation status is undefined.

Examples

Consider the following:

```
REAL, ALLOCATABLE, DIMENSION (:,:,:) :: E
PRINT *, ALLOCATED (E)       ! Returns the value false
ALLOCATE (E (12, 15, 20))
PRINT *, ALLOCATED (E)       ! Returns the value true
```

ANINT

Description: Calculates the nearest whole number.

Syntax: result = ANINT (*a* [, *kind*])

Class: Elemental function; Generic

Arguments:

a Must be of type real.

kind (opt) Must be a scalar integer initialization expression.

Results: The result type is real. If *kind* is present, the kind parameter is that specified by *kind*; otherwise, the kind parameter is that of *a*. If *a* is greater than zero, ANINT (*a*) has the value AINT (*a* + 0.5); if *a* is less than or equal to zero, ANINT (*a*) has the value AINT (*a* – 0.5).

Specific Name	Argument Type	Result Type
ANINT	REAL(4)	REAL(4)
DNINT	REAL(8)	REAL(8)
QNINT	REAL(16)	REAL(16)

Examples

ANINT (3.456) has the value 3.0.

ANINT (–2.798) has the value –3.0.

ANY

Description: Determines if *any* value is true in an entire array or in a specified dimension of an array.

Syntax: result = ANY (*mask* [, *dim*])

Class: Transformational function; Generic

Arguments:

mask	Must be a logical array.
dim (opt)	Must be a scalar integer expression with a value in the range 1 to n, where *n* is the rank of *mask*.

Results: The result is an array or a scalar of type logical.

The result is a scalar if *dim* is omitted or *mask* has rank one. A scalar result is true if any elements of *mask* are true. The result has the value false if no element of *mask* is true, or *mask* has size zero.

An array result has the same type and kind parameters as *mask*, and a rank that is one less than *mask*. Its shape is $(d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n)$, where $(d_1, d_2, ..., d_n)$ is the shape of *mask*.

Each element in an array result is true if any elements in the one dimensional array defined by *mask* $(s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n)$ are true.

Examples

ANY ((/.FALSE., .FALSE., .TRUE./)) has the value true because one element is true.

A is the array $\begin{bmatrix} 1 & 5 & 7 \\ 3 & 6 & 8 \end{bmatrix}$ and B is the array $\begin{bmatrix} 0 & 5 & 7 \\ 2 & 6 & 9 \end{bmatrix}$.

ANY (A .EQ. B, DIM=1) tests to see if *any* elements in each column of A are equal to the elements in the corresponding column of B. The result has the value (false, true, true) because the second and third columns have at least one element that is equal.

ANY (A .EQ. B, DIM=2) tests to see if any elements in each row of A are equal to the elements in the corresponding row of B. The result has the value (true, true) because each row has at least one element that is equal.

ASIN

Description: Produces the arcsine of *x*.

Syntax: result = ASIN (*x*)

Class: Elemental function; Generic

Arguments: *x* must be of type real. The | *x* | must be less than or equal to 1.

Results: The result type is the same as *x* and is expressed in radians. The value lies in the range $-\pi/2$ to $\pi/2$.

Specific Name	Argument Type	Result Type
ASIN	REAL(4)	REAL(4)
DASIN	REAL(8)	REAL(8)
QASIN	REAL(16)	REAL(16)

Example

ASIN (0.79345021) has the value 0.9164571.

ASIND

Description: Produces the arcsine of *x*.

Syntax: result = ASIND (*x*)

Class: Elemental function; Generic

Arguments: *x* must be of type real. The | *x* | must be less than or equal to 1.

Results: The result type is the same as *x* and is expressed in degrees. The value lies in the range –90 to 90 degrees.

Specific Name	Argument Type	Result Type
ASIND	REAL(4)	REAL(4)
DASIND	REAL(8)	REAL(8)
QASIND	REAL(16)	REAL(16)

Example

ASIND (0.2467590) has the value 14.28581.

ASINH

Description: Produces the hyperbolic arcsine of *x*.

Syntax: result = ASINH (*x*)

Class: Elemental function; Generic

Arguments: *x* must be of type real.

Results: The result type is the same as *x*.

Specific Name	Argument Type	Result Type
ASINH	REAL(4)	REAL(4)
DASINH	REAL(8)	REAL(8)
QASINH	REAL(16)	REAL(16)

Example

ASINH (180.0) has the value 5.88611.

ASSOCIATED

Description: Returns the association status of its pointer argument or indicates whether the pointer is associated with the target.

Syntax: result = ASSOCIATED (*pointer* [, *target*])

Class: Inquiry function; Generic

Arguments:

pointer Must be a pointer (of any data type).

target (opt) Must be a pointer or target.

Results: The result is a scalar of type default logical. The setting of compiler options specifying integer size can affect this function.

If only *pointer* appears, the result is true if it is currently associated with a target; otherwise, the result is false.

If *target* also appears and is a target, the result is true if *pointer* is currently associated with *target*; otherwise, the result is false.

If *target* is a pointer, the result is true if both *pointer* and *target* are currently associated with the same target; otherwise, the result is false. (If either *pointer* or *target* is disassociated, the result is false.)

Examples

Consider the following:

```
REAL, TARGET, DIMENSION (0:50) :: TAR
REAL, POINTER, DIMENSION (:) :: PTR
PTR => TAR
PRINT *, ASSOCIATED (PTR, TAR)     ! Returns the value true
```

The subscript range for PTR is 0:50. Consider the following pointer assignment statements:

```
(1) PTR => TAR (:)
(2) PTR => TAR (0:50)
(3) PTR => TAR (0:49)
```

For statements 1 and 2, ASSOCIATED (PTR, TAR) is true because TAR has not changed (the subscript range for PTR in both cases is 1:51, following the rules for deferred-shape arrays). For statement 3, ASSOCIATED (PTR, TAR) is false because the upper bound of TAR has changed.

Consider the following:

```
REAL, POINTER, DIMENSION (:) :: PTR2, PTR3
ALLOCATE (PTR2 (0:15))
PTR3 => PTR2
PRINT *, ASSOCIATED (PTR2, PTR3)    ! Returns the value true
...
NULLIFY (PTR2)
NULLIFY (PTR3)
PRINT *, ASSOCIATED (PTR2, PTR3)    ! Returns the value false
```

ATAN

Description: Produces the arctangent of *x*.

Syntax: result = ATAN (*x*)

Class: Elemental function; Generic

Arguments: *x* must be of type real.

Results: The result type is the same as *x* and is expressed in radians. The value lies in the range $-\pi/2$ to $\pi/2$.

Specific Name	Argument Type	Result Type
ATAN	REAL(4)	REAL(4)
DATAN	REAL(8)	REAL(8)
QATAN	REAL(16)	REAL(16)

Example

ATAN (1.5874993) has the value 1.008666.

ATAN2

Description: Produces an arctangent. The result is the principal value of the argument of the nonzero complex number (*x*, *y*).

Syntax: result = ATAN2 (*y*, *x*)

Class: Elemental function; Generic

Arguments:

y Must be of type real.

x Must have the same type and kind parameters as *y*. If *y* has the value zero, *x* cannot have the value zero.

Results: The result type is the same as *x* and is expressed in radians. The value lies in the range $-\pi < \mathrm{ATAN2}(y, x) \le \pi$. If $x \ne \mathrm{zero}$, the result is approximately equal to the value of arctan (*y*/*x*).

If *y* > zero, the result is positive.

If *y* < zero, the result is negative.

If *y* = zero, the result is zero (if *x* > zero) or π (if *x* < zero).

If *x* = zero, the absolute value of the result is $\pi/2$.

Specific Name	Argument Type	Result Type
ATAN2	REAL(4)	REAL(4)
DATAN2	REAL(8)	REAL(8)
QATAN2	REAL(16)	REAL(16)

Examples

ATAN2 (2.679676, 1.0) has the value 1.213623.

If Y has the value $\begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix}$ and X has the value $\begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix}$, then ATAN2 (Y, X) is

$$\begin{bmatrix} \frac{3\pi}{4} & \frac{\pi}{4} \\ \frac{-3\pi}{4} & \frac{-\pi}{4} \end{bmatrix}.$$

ATAN2D

Description: Produces an arctangent. The result is the principal value of the argument of the nonzero complex number (*x*, *y*).

Syntax: result = ATAN2D (*y*, *x*)

Class: Elemental function; Generic

Arguments:

y Must be of type real.

x Must have the same type and kind parameters as *y*. If *y* has the value zero, *x* cannot have the value zero.

Results: The result type is the same as *x* and is expressed in degrees. The value lies in the range –180 degrees to 180 degrees. If $x \neq \mathrm{zero}$, the result is approximately equal to the value of arctan (*y*/*x*).

If *y* > zero, the result is positive.

If *y* < zero, the result is negative.

If *y* = zero, the result is zero (if *x* > zero) or 180 degrees (if *x* < zero).

If *x* = zero, the absolute value of the result is 90 degrees.

Specific Name	Argument Type	Result Type
ATAN2D	REAL(4)	REAL(4)
DATAN2D	REAL(8)	REAL(8)
QATAN2D	REAL(16)	REAL(16)

Example

ATAN2D (2.679676, 1.0) has the value 69.53546.

ATAND

Description: Produces the arctangent of *x*.

Syntax: result = ATAND (*x*)

Class: Elemental function; Generic

Arguments: *x* must be of type real and must be greater than or equal to zero.

Results: The result type is the same as *x* and is expressed in degrees.

Specific Name	Argument Type	Result Type
ATAND	REAL(4)	REAL(4)
DATAND	REAL(8)	REAL(8)
QATAND	REAL(16)	REAL(16)

Example

ATAND (0.0874679) has the value 4.998819.

ATANH

Description: Produces the hyperbolic arctangent of *x*.

Syntax: result = ATANH (*x*)

Class: Elemental function; Generic

Arguments: *x* must be of type real, where $|x| \leq 1$.

Results: The result type is the same as *x*. The value lies in the range –1.0 to 1.0.

Specific Name	Argument Type	Result Type
ATANH	REAL(4)	REAL(4)
DATANH	REAL(8)	REAL(8)
QATANH	REAL(16)	REAL(16)

Example

ATANH (–0.77) has the value –1.02033.

BADDRESS

Description: Returns the address of *x*. It cannot be passed as an actual argument. This function can also be specified as IADDR.

Syntax: result = BADDRESS (*x*)

Class: Inquiry function; Generic

Arguments: *x* is a variable, an array or record field reference, a procedure, or a constant; it can be of any data type. It must not be the name of an internal procedure or statement function. If it is a pointer, it must be defined and associated with a target.

Results: The result type is INTEGER(4) on IA-32 processors; INTEGER(8) on Intel® Itanium® processors. The value of the result represents the address of the data object or, in the case of pointers, the address of its associated target. If the argument is not valid, the result is undefined.

Example

Consider the following:

```
PROGRAM batest
  INTEGER X(5), I
  DO I=1, 5
    PRINT *, BADDRESS(X(I))
  END DO
END
```

BIT_SIZE

Description: Returns the number of bits in an integer type.

Syntax: result = BIT_SIZE (*i*)

Class: Inquiry function; Generic

Arguments: *i* must be of type integer.

Results: The result is a scalar integer with the same kind parameter as *i*. The result value is the number of bits (*s*) defined by the bit model for integers with the kind parameter of the argument. For information on the bit model, see "Model for Bit Data".

Example

BIT_SIZE (1_2) has the value 16 because the KIND=2 integer type contains 16 bits.

BTEST

Description: Tests a bit of an integer argument.

Syntax: result = BTEST (*i*, *pos*)

Class: Elemental function; Generic

Arguments:

i Must be of type integer.

pos Must be of type integer. It must not be negative and it must be less than BIT_SIZE (*i*).
The rightmost (least significant) bit of *i* is in position 0.

Results: The result type is default logical.

The result is true if bit *pos* of I has the value 1. The result is false if *pos* has the value zero. For more information on bit functions, see "Bit Functions".

The setting of compiler options specifying integer size can affect this function.

Specific Name	Argument Type	Result Type
BBTEST	INTEGER(1)	LOGICAL(1)
BITEST[1]	INTEGER(2)	LOGICAL(2)
BTEST[2]	INTEGER(4)	LOGICAL(4)
BKTEST	INTEGER(8)	LOGICAL(8)

1. Or HTEST
2. Or BJTEST

Examples

BTEST (9, 3) has the value true.

If A has the value $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, the value of BTEST (A, 2) is $\begin{bmatrix} \mathtt{false} & \mathtt{false} \\ \mathtt{false} & \mathtt{true} \end{bmatrix}$ and the value of BTEST (2, A) is $\begin{bmatrix} \mathtt{true} & \mathtt{false} \\ \mathtt{false} & \mathtt{false} \end{bmatrix}$.

CACHESIZE (i64 only)

Description: Returns the size of a level of the memory cache. This specific function has no generic function associated with it and is only available on Intel Itanium processors. It must not be passed as an actual argument.

Syntax: result = CACHESIZE (*n*)

Class: Inquiry function; Specific

Arguments: *n* must be scalar and of type INTEGER(4).

Results: The result type is INTEGER(4). The result value is the number of kilobytes in the level *n* memory cache.

n = 1 specifies the first level cache; *n* = 2 specifies the second level cache; etc. If cache level *n* does not exist, the result value is 0.

Example

CACHESIZE(1) returns 16 for a processor with a 16KB first level memory cache.

CEILING

Description: Returns the smallest integer greater than or equal to its argument.

Syntax: result = CEILING (*a* [, *kind*])

Class: Elemental function; Generic

Arguments:

a Must be of type real.

kind (opt) Must be a scalar integer initialization expression. This argument is a Fortran 95 feature.

Results: The result type is integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The value of the result is equal to the smallest integer greater than or equal to *a*.

The setting of compiler options specifying integer size can affect this function.

Examples

CEILING (4.8) has the value 5.

CEILING (–2.55) has the value –2.0.

CHAR

Description: Returns the character in the specified position of the processor's character set. It is the inverse of the function ICHAR.

Syntax: result = CHAR (*i* [, *kind*])

Class: Elemental function; Generic

Arguments:

i Must be of type integer with a value in the range 0 to n – 1, where *n* is the number of characters in the processor's character set.

kind (opt) Must be a scalar integer initialization expression.

Results: The result is of type character with length 1. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default character. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The result is the character in position *i* of the processor's character set. ICHAR(CHAR (I, KIND(C))) has the value I for 0 to n – 1 and CHAR(ICHAR(C), KIND(C)) has the value C for any character C capable of representation in the processor.

Specific Name	Argument Type	Result Type
	INTEGER(1)	CHARACTER
	INTEGER(2)	CHARACTER
CHAR[1]	INTEGER(4)	CHARACTER
	INTEGER(8)	CHARACTER

1. This specific function cannot be passed as an actual argument.

Examples

CHAR (76) has the value 'L'.

CHAR (94) has the value '^'.

CMPLX

Description: Converts the argument to complex type. This function cannot be passed as an actual argument.

Syntax: result = CMPLX (*x* [, *y*] [, *kind*])

Class: Elemental function; Specific

Arguments:

x Must be of type integer, real, or complex.

y (opt) Must be of type integer or real. It must not be present if *x* is of type complex.

kind (opt) Must be a scalar integer initialization expression.

Results: The result type is complex. If *kind* is present, the kind parameter is that specified by *kind*; otherwise, the kind parameter is that of default real type.

If only one noncomplex argument appears, it is converted into the real part of the result value and zero is assigned to the imaginary part. If *y* is not specified and *x* is complex, the result value is CMPLX (REAL(*x*), AIMAG(*x*)).

If two noncomplex arguments appear, the complex value is produced by converting the first argument into the real part of the value, and converting the second argument into the imaginary part.

CMPLX(*x*, *y*, *kind*) has the complex value whose real part is REAL(*x*, *kind*) and whose imaginary part is REAL(*y*, *kind*).

The setting of compiler options specifying real size can affect this function.

Examples

CMPLX (–3) has the value (–3.0, 0.0).

CMPLX (4.1, 2.3) has the value (4.1, 2.3).

CONJG

Description: Calculates the conjugate of a complex number.

Syntax: result = CONJG (*z*)

Class: Elemental function; Generic

Arguments: *z* must be of type complex.

Results: The result type is the same as *z*. If *z* has the value (x, y), the result has the value (x, –y).

Specific Name	Argument Type	Result Type
CONJG	COMPLEX(4)	COMPLEX(4)
DCONJG	COMPLEX(8)	COMPLEX(8)
QCONJG	COMPLEX(16)	COMPLEX(16)

Examples

CONJG ((2.0, 3.0)) has the value (2.0,–3.0).

CONJG ((1.0, –4.2)) has the value (1.0, 4.2).

COS

Description: Produces the cosine of *x*.

Syntax: result = COS (*x*)

Class: Elemental function; Generic

Arguments: *x* must be of type real or complex. It must be in radians and is treated as modulo 2*π.

If *x* is of type complex, its real part is regarded as a value in radians.

Results: The result type is the same as *x*.

Specific Name	Argument Type	Result Type
COS	REAL(4)	REAL(4)
DCOS	REAL(8)	REAL(8)
QCOS	REAL(16)	REAL(16)
CCOS[1]	COMPLEX(4)	COMPLEX(4)
CDCOS[2]	COMPLEX(8)	COMPLEX(8)
CQCOS	COMPLEX(16)	COMPLEX(16)

1. The setting of compiler options specifying real size can affect CCOS.
2. This function can also be specified as ZCOS.

Examples

COS (2.0) has the value –0.4161468.

COS (0.567745) has the value 0.8431157.

COSD

Description: Produces the cosine of *x*.

Syntax: result = COSD (*x*)

Class: Elemental function; Generic

Arguments: *x* must be of type real. It must be in degrees and is treated as modulo 360.

Results: The result type is the same as *x*.

Specific Name	Argument Type	Result Type
COSD	REAL(4)	REAL(4)

Specific Name	Argument Type	Result Type
DCOSD	REAL(8)	REAL(8)
QCOSD	REAL(16)	REAL(16)

Examples

COSD (2.0) has the value 0.9993908.

COSD (30.4) has the value 0.8625137.

COSH

Description: Produces a hyperbolic cosine.

Syntax: result = COSH (*x*)

Class: Elemental function; Generic

Arguments: *x* must be of type real.

Results: The result type is the same as *x*.

Specific Name	Argument Type	Result Type
COSH	REAL(4)	REAL(4)
DCOSH	REAL(8)	REAL(8)
QCOSH	REAL(16)	REAL(16)

Examples

COSH (2.0) has the value 3.762196.

COSH (0.65893) has the value 1.225064.

COTAN

Description: Produces the cotangent of *x*.

Syntax: result = COTAN (*x*)

Class: Elemental function; Generic

Arguments: *x* must be of type real; it cannot be zero. It must be in radians and is treated as modulo $2*\pi$.

Results: The result type is the same as *x*.

Specific Name	Argument Type	Result Type
COTAN	REAL(4)	REAL(4)
DCOTAN	REAL(8)	REAL(8)
QCOTAN	REAL(16)	REAL(16)

Examples

COTAN (2.0) has the value –4.576575E–01.

COTAN (0.6) has the value 1.461696.

COTAND

Description: Produces the cotangent of X.

Syntax: result = COTAND (*x*)

Class: Elemental function; Generic

Arguments: *x* must be of type real. It must be in degrees and is treated as modulo 360.

Results: The result type is the same as *x*.

Specific Name	Argument Type	Result Type
COTAND	REAL(4)	REAL(4)
DCOTAND	REAL(8)	REAL(8)
QCOTAND	REAL(16)	REAL(16)

Examples

COTAND (2.0) has the value 0.2863625E+02.

COTAND (0.6) has the value 0.9548947E+02.

COUNT

Description: Counts the number of true elements in an entire array or in a specified dimension of an array.

Syntax: result = COUNT (*mask* [, *dim*] [, *kind*])

Class: Transformational function; Generic

Arguments:

mask	Must be a logical array.
dim (opt)	Must be a scalar integer expression with a value in the range 1 to n, where *n* is the rank of *mask*.
kind (opt)	Must be a scalar integer initialization expression.

Results: The result is an array or scalar of type integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The result is a scalar if *dim* is omitted or *mask* has rank one. A scalar result has a value equal to the number of true elements of *mask*. If *mask* has size zero, the result is zero.

An array result has a rank that is one less than *mask*, and shape (d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n), where (d_1, d_2,..., d_n) is the shape of *mask*.

Each element in an array result equals the number of elements that are true in the one dimensional array defined by *mask* (s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n).

The setting of compiler options specifying integer size can affect this function.

Examples

COUNT ((/.TRUE., .FALSE., .TRUE./)) has the value 2 because two elements are true.

COUNT ((/.TRUE., .TRUE., .TRUE./)) has the value 3 because three elements are true.

A is the array $\begin{bmatrix} 1 & 5 & 7 \\ 3 & 6 & 8 \end{bmatrix}$ and B is the array $\begin{bmatrix} 0 & 5 & 7 \\ 2 & 6 & 9 \end{bmatrix}$.

COUNT (A .NE. B, DIM=1) tests to see how many elements in each column of A are not equal to the elements in the corresponding column of B. The result has the value (2, 0, 1) because:

- The first column of A and B have 2 elements that are not equal.
- The second column of A and B have 0 elements that are not equal.
- The third column of A and B have 1 element that is not equal.

COUNT (A .NE. B, DIM=2) tests to see how many elements in each row of A are not equal to the elements in the corresponding row of B. The result has the value (1, 2) because:

- The first row of A and B have 1 element that is not equal.
- The second row of A and B have 2 elements that are not equal.

CPU_TIME

Description: Returns a processor-dependent approximation of the processor time in seconds. This is a new intrinsic procedure in Fortran 95.

Syntax: CALL CPU_TIME (*time*)

Class: Subroutine

Arguments: *time* must be scalar and of type real. It is an INTENT(OUT) argument.

If a meaningful time cannot be returned, a processor-dependent negative value is returned.

Example

Consider the following:

```
REAL time_begin, time_end
...
CALL CPU_TIME(time_begin)
...
CALL CPU_TIME(time_end)
PRINT (*,*) 'Time of operation was ', time_end - time_begin, ' seconds'
```

CSHIFT

Description: Performs a *circular* shift on a rank-one array, or performs circular shifts on all the complete rank-one sections (vectors) along a given dimension of an array of rank two or greater.

Elements shifted off one end are inserted at the other end. Different sections can be shifted by different amounts and in different directions.

Syntax: result = CSHIFT (*array*, *shift* [, *dim*])

Class: Transformational function; Generic

Arguments:

array Must be an array; it can be of any data type.

shift Must be a scalar integer or an array with a rank that is one less than *array*, and shape (d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n), where (d_1, d_2, ..., d_n) is the shape of *array*.

dim (opt) Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of *array*. If *dim* is omitted, it is assumed to be 1.

Results: The result is an array with the same type and kind parameters, and shape as *array.*

If *array* has rank one, element i of the result is *array* (1 + MODULO (i + *shift* – 1, SIZE (*array*))). (The same shift is applied to each element.)

If *array* has rank greater than one, each section (s_1,s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n) of the result is shifted as follows:

- By the value of *shift*, if *shift* is scalar
- According to the corresponding value in *shift*(s_1, s_2,..., s_{DIM-1}, s_{DIM+1},..., s_n), if *shift* is an array

The value of *shift* determines the amount and direction of the circular shift. A positive *shift* value causes a shift to the left (in rows) or up (in columns). A negative *shift* value causes a shift to the right (in rows) or down (in columns). A zero *shift* value causes no shift.

Examples

V is the array (1, 2, 3, 4, 5, 6).

CSHIFT (V, SHIFT=2) shifts the elements in V circularly to the *left* by 2 positions, producing the value (3, 4, 5, 6, 1, 2). 1 and 2 are shifted off the beginning and inserted at the end.

CSHIFT (V, SHIFT= –2) shifts the elements in V circularly to the *right* by 2 positions, producing the value (5, 6, 1, 2, 3, 4). 5 and 6 are shifted off the end and inserted at the beginning.

M is the array $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$. CSHIFT (M, SHIFT = 1, DIM = 2) produces the result $\begin{bmatrix} 2 & 3 & 1 \\ 5 & 6 & 4 \\ 8 & 9 & 7 \end{bmatrix}$.

Each element in rows 1, 2, and 3 is shifted to the *left* by 2 positions. The elements shifted off the beginning are inserted at the end.

CSHIFT (M, SHIFT = –1, DIM = 1) produces the result $\begin{bmatrix} 7 & 8 & 9 \\ 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$.

Each element in columns 1, 2, and 3 is shifted down by 1 position. The elements shifted off the end are inserted at the beginning.

CSHIFT (M, SHIFT = (/1, –1, 0/), DIM = 2) produces the result $\begin{bmatrix} 2 & 3 & 1 \\ 6 & 4 & 5 \\ 7 & 8 & 9 \end{bmatrix}$.

Each element in row 1 is shifted to the *left* by 1 position; each element in row 2 is shifted to the *right* by 1 position; no element in row 3 is shifted at all.

DATE

Description: Returns the current date as set within the system.

Syntax: CALL DATE (*buf*)

Class: Subroutine

Arguments: *buf* is a 9-byte variable, array, array element, or character substring.

The date is returned as a 9-byte ASCII character string taking the form dd-mmm-yy, where:

dd	is the 2-digit date
mmm	is the 3-letter month
yy	is the last two digits of the year

If *buf* is of numeric type and smaller than 9 bytes, data corruption can occur.

If *buf* is of character type, its associated length is passed to the subroutine. If *buf* is smaller than 9 bytes, the subroutine truncates the date to fit in the specified length. If an array of type character is passed, the subroutine stores the date in the first array element, using the element length, not the length of the entire array.

CAUTION. *The two-digit year return value may cause problems with the year 2000 or later. Use* DATE_AND_TIME *instead (see* "DATE_AND_TIME"*).*

Example

Consider the following:

```
CHARACTER*1 DAY(9)
...
CALL DATE (DAY)
```

The length of the first array element in CHARACTER array DAY is passed to the DATE subroutine. The subroutine then truncates the date to fit into the one-character element, producing an incorrect result.

DATE_AND_TIME

Description: Returns character data on the real-time clock and date in a form compatible with the representations defined in Standard ISO 8601:1988.

Syntax: CALL DATE_AND_TIME ([*date*] [, *time*] [, *zone*] [, *values*])

Class: Subroutine

Arguments: There are four optional arguments:[1]

date (opt) Must be scalar and of type default character; its length must be at least 8 to contain the complete value. Its leftmost 8 characters are set to a value of the form CCYYMMDD, where:

CC is the century
MM is the month within the year
DD is the day within the month

time (opt) Must be scalar and of type default character; its length must be at least 10 to contain the complete value. Its leftmost 10 characters are set to a value of the form hhmmss.sss, where:

hh is the hour of the day
mm is the minutes of the hour
ss.sss is the seconds and milliseconds of the minute

zone (opt) Must be scalar and of type default character; its length must be at least 5 to contain the complete value. Its leftmost 5 characters are set to a value of the form hhmm, where *hh* and *mm* are the time difference with respect to Coordinated Universal Time (UTC)[2] in hours and parts of an hour expressed in minutes, respectively.

values (opt) Must be of type default integer and of rank one. Its size must be at least 8. The values returned in VALUES are as follows:

VALUES (1) is the 4-digit year.
VALUES (2) is the month of the year.
VALUES (3) is the day of the month.
VALUES (4) is the time difference with respect to Coordinated Universal Time (UTC) in minutes.
VALUES (5) is the hour of the day (range 0 to 23). [3]
VALUES (6) is the minutes of the hour (range 0 to 59). [3]
VALUES (7) is the seconds of the minute (range 0 to 59). [3]
VALUES (8) is the milliseconds of the second (range 0 to 999).[3]

1. All are INTENT(OUT) arguments. (See "INTENT Attribute and Statement".)
2. UTC (also known as Greenwich Mean Time) is defined by CCIR Recommendation 460-2.
3. In local time.

The setting of compiler options specifying integer size can affect this subroutine.

Examples

Consider the following example executed 2000 March 28 at 11:04:14.5:

```
INTEGER DATE_TIME (8)
CHARACTER (LEN = 12) REAL_CLOCK (3)
CALL DATE_AND_TIME (REAL_CLOCK (1), REAL_CLOCK (2), &
                    REAL_CLOCK (3), DATE_TIME)
```

This assigns the value "20000328" to REAL_CLOCK (1), the value "110414.500" to REAL_CLOCK (2), and the value "–0500" to REAL_CLOCK (3). The following values are assigned to DATE_TIME: 2000, 3, 28, -300, 11, 4, 14, and 500.

DBLE

Description: Converts a number to double-precision real type.

Syntax: result = DBLE (*a*)

Class: Elemental function; Generic

Arguments: *a* must be of type integer, real, or complex.

Results: The result type is double precision real (REAL(8) or REAL*8). Functions that cause conversion of one data type to another type have the same effect as the implied conversion in assignment statements.

If *a* is of type double precision, the result is the value of the *a* with no conversion (DBLE(*a*) = *a*).

If *a* is of type integer or real, the result has as much precision of the significant part of *a* as a double precision value can contain.

If *a* is of type complex, the result has as much precision of the significant part of the real part of *a* as a double precision value can contain.

Specific Name[1]	Argument Type	Result Type
	INTEGER(1)	REAL(8)
	INTEGER(2)	REAL(8)
	INTEGER(4)	REAL(8)
	INTEGER(8)	REAL(8)
DBLE[2]	REAL(4)	REAL(8)
	REAL(8)	REAL(8)
DBLEQ	REAL(16)	REAL(8)
	COMPLEX(4)	REAL(8)
	COMPLEX(8)	REAL(8)

Specific Name[1]	Argument Type	Result Type
	COMPLEX(16)	REAL(8)

1. These specific functions cannot be passed as actual arguments.
2. For compatibility with older versions of Fortran, DBLE can be specified as a specific function.

Examples

DBLE (4) has the value 4.0.

DBLE ((3.4, 2.0)) has the value 3.4.

DCMPLX

Description: Converts the argument to double complex type. This function cannot be passed as an actual argument.

Syntax: result = DCMPLX (*x* [, *y*])

Class: Elemental function; Specific

Arguments:

x Must be of type integer, real, or complex.

y Must be of type integer or real. It must not be present if *x* is of type complex.

Results: The result type is double complex (COMPLEX(8) or COMPLEX*16).

If only one noncomplex argument appears, it is converted into the real part of the result value and zero is assigned to the imaginary part. If *y* is not specified and *x* is complex, the result value is CMPLX (REAL(*x*), AIMAG(*x*)).

If two noncomplex arguments appear, the complex value is produced by converting the first argument into the real part of the value, and converting the second argument into the imaginary part.

DCMPLX(*x*, *y*) has the complex value whose real part is REAL(*x*, KIND=8) and whose imaginary part is REAL(*y*, KIND=8).

Examples

DCMPLX (–3) has the value (–3.0, 0.0).

DCMPLX (4.1, 2.3) has the value (4.1, 2.3).

DFLOAT

Description: Converts an integer to double-precision real (REAL(8)) type.

Syntax: result = DFLOAT (*a*)

Class: Elemental function; Generic

Arguments: *a* must be of type integer.

Results: The result type is double precision real (REAL(8) or REAL*8).

Functions that cause conversion of one data type to another type have the same affect as the implied conversion in assignment statements.

Specific Name[1]	Argument Type	Result Type
	INTEGER(1)	REAL(8)
DFLOTI	INTEGER(2)	REAL(8)
DFLOTJ	INTEGER(4)	REAL(8)
DFLOTK	INTEGER(8)	REAL(8)

1. These specific functions cannot be passed as actual arguments.

Example

DFLOAT (–4) has the value –4.0.

DIGITS

Description: Returns the number of significant digits for numbers of the same type and kind parameters as the argument.

Syntax: result = DIGITS (*x*)

Class: Inquiry function; Generic

Arguments: *x* must be of type integer or real; it can be scalar or array valued.

Results: The result is a scalar of type default integer.

The result has the value *q* if *x* is of type integer; it has the value *p* if *x* is of type real. Integer parameter *q* is defined in "Model for Integer Data"; real parameter *p* is defined in "Model for Real Data".

Example

If X is of type REAL(4), DIGITS (X) has the value 24.

DIM

Description: Returns the difference between two numbers (if the difference is positive).

Syntax: result = DIM (*x*, *y*)

Class: Elemental function; Generic

Arguments:

x Must be of type integer or real.

y Must have the same type and kind parameters as *x*.

Results: The result type is the same as *x*. The value of the result is *x* – *y* if *x* is greater than *y*; otherwise, the value of the result is zero.

Specific Name	Argument Type	Result Type
BDIM	INTEGER(1)	INTEGER(1)
IIDIM[1]	INTEGER(2)	INTEGER(2)
IDIM[2]	INTEGER(4)	INTEGER(4)
KIDIM[3]	INTEGER(8)	INTEGER(8)
DIM	REAL(4)	REAL(4)
DDIM	REAL(8)	REAL(8)
QDIM	REAL(16)	REAL(16)

1. Or HDIM
2. Or JIDIM. For compatibility, IDIM can also be specified as a generic function.
3. Or KDIM

Examples

DIM (6, 2) has the value 4.

DIM (–4.0, 3.0) has the value 0.0.

DNUM

Description: Converts a character string to a double-precision real value.

Syntax: result = DNUM (*i*)

Class: Elemental function; Specific

Arguments: *i* must be of type character.

Results: The result type is double-precision real. The result value is the double-precision real value represented by the character string *i*.

Examples

DNUM ("3.14159") has the double-precision value 3.14159.

The following sets x to 311.0:

```
CHARACTER(3) i
DOUBLE PRECISION x
i = "311"
x = DNUM(i)
```

DOT_PRODUCT

Description: Performs dot-product multiplication of numeric or logical vectors (rank-one arrays).

Syntax: result = DOT_PRODUCT (*vector_a*, *vector_b*)

Class: Transformational function; Generic

Arguments:

vector_a Must be a rank-one array of numeric (integer, real, or complex) or logical type.

vector_b Must be a rank-one array of numeric type if *vector_a* is of numeric type, or of logical type if *vector_a* is of logical type. It must be the same size as *vector_a*.

Results: The result is a scalar whose type depends on the types of *vector_a* and *vector_b*.

If *vector_a* is of type integer or real, the result value is SUM (*vector_a***vector_b*).

If *vector_a* is of type complex, the result value is SUM (CONJG (*vector_a*)**vector_b*).

If *vector_a* is of type logical, the result has the value ANY (*vector_a* .AND. *vector_b*).

If either rank-one array has size zero, the result is zero if the array is of numeric type, and false if the array is of logical type. (For more information on expressions, see "Expressions".)

Examples

DOT_PRODUCT ((/1, 2, 3/), (/3, 4, 5/)) has the value 26, calculated as follows:

```
((1 x 3) + (2 x 4) + (3 x 5)) = 26
```

DOT_PRODUCT ((/ (1.0, 2.0), (2.0, 3.0) /), (/ (1.0, 1.0), (1.0, 4.0) /)) has the value (17.0, 4.0).

DOT_PRODUCT ((/ .TRUE., .FALSE. /), (/ .FALSE., .TRUE. /)) has the value false.

DPROD

Description: Produces a higher precision product. This is a specific function that has no generic name associated with it. It cannot be passed as an actual argument.

Syntax: result = DPROD (*x*, *y*)

Class: Elemental function; Specific

Arguments:

x Must be of type REAL(4) or REAL(8).

y Must have the same type and kind parameters as *x*.

Results: If *x* and *y* are of type REAL(4), the result type is double-precision real (REAL(8) or REAL*8). If *x* and *y* are of type REAL(8), the result is of type REAL(16). The result value is equal to *x**y*.

The setting of compiler options specifying real size can affect this function.

Examples

DPROD (2.0, –4.0) has the value –8.00D0.

DPROD (5.0D0, 3.0D0) has the value 15.00Q0.

The following shows another example:

```
REAL(4) e
REAL(8) d
e = 123456.7
d = 123456.7D0   ! DPROD (e,e) returns 15241557546.4944
! DPROD (d,d) returns 15241556774.8899992813874268904328
```

DREAL

Description: Converts the real part of a double complex argument to double-precision real type. This specific function has no generic function associated with it. It cannot be passed as an actual argument.

Syntax: result = DREAL (*a*)

Class: Elemental function; Specific

Arguments: *a* must be of type double complex (COMPLEX(8) or COMPLEX*16).

Results: The result type is double precision real (REAL(8) or REAL*8).

Example

DREAL ((2.0d0, 3.0d0)) has the value 2.0d0.

DSHIFTL

Description: Arithmetically shifts a 128-bit integer to the left.

Syntax: result = DSHIFTL (*ileft*, *iright*, *ishift*)

Class: Elemental function; Specific

Arguments:

ileft Must be of type INTEGER(8).

iright Must be of type INTEGER(8).

ishift Must be of type INTEGER(8). It must be nonnegative and less than or equal to 64. This is the shift count.

Results: The result type is INTEGER(8). The result value is the 64-bit value starting at bit 128 – *ishift* of the 128-bit concatenation of the values of *ileft* and *iright*.

Example

Consider the following:

```
INTEGER(8) ILEFT / Z'111122221111222' /
INTEGER(8) IRIGHT / Z'FFFFFFFFFFFFF' /
PRINT *, DSHIFTL (ILEFT, IRIGHT, 16_8)! prints 1306643199093243919
```

DSHIFTR

Description: Arithmetically shifts a 128-bit integer to the right.

Syntax: result = DSHIFTR (*ileft*, *iright*, *ishift*)

Class: Elemental function; Specific

Arguments:

ileft Must be of type INTEGER(8).

iright Must be of type INTEGER(8).

ishift Must be of type INTEGER(8). It must be nonnegative and less than or equal to 64. This is the shift count.

Results: The result type is INTEGER(8). The result value is the 64-bit value starting at bit 64 + *ishift* of the 128-bit concatenation of the values of *ileft* and *iright*.

Example

Consider the following:

```
INTEGER(8) ILEFT / Z'111122221111222' /
INTEGER(8) IRIGHT / Z'FFFFFFFFFFFFF' /
PRINT *, DSHIFTR (ILEFT, IRIGHT, 16_8)! prints 1306606910610341887
```

EOF

Description: Checks whether a file is at or beyond the end-of-file record. This specific function has no generic function associated with it. It cannot be passed as an actual argument.

Syntax: result = EOF (*a*)

Class: Inquiry function; Specific

Arguments: *a* must be of type integer. It represents a unit specifier corresponding to an open file. It cannot be zero unless you have reconnected unit zero to a unit other than the screen or keyboard.

Results: The result type is logical. The value of the result is .TRUE. if the file connected to *a* is at or beyond the end-of-file record; otherwise, .FALSE..

Example

Consider the following:

```
!  Creates a file of random numbers, and reads them back
       REAL x, total
       INTEGER count
       OPEN (1, FILE = 'TEST.DAT')
       DO I = 1, 20
         CALL RANDOM_NUMBER(x)
         WRITE (1, '(F6.3)') x * 100.0
       END DO
       CLOSE(1)
```

```
      OPEN (1, FILE = 'TEST.DAT')
      DO WHILE (.NOT. EOF(1))
       count = count + 1
       READ (1, *) value
       total = total + value
      END DO
100   IF ( count .GT. 0) THEN
       WRITE (*,*) 'Average is: ', total / count
      ELSE
       WRITE (*,*) 'Input file is empty '
      END IF
STOP
      END
```

EOSHIFT

Description: Performs an *end-off* shift on a rank-one array, or performs end-off shifts on all the complete rank-one sections along a given dimension of an array of rank two or greater.

Elements are shifted off at one end of a section and copies of a boundary value are filled in at the other end. Different sections can have different boundary values and can be shifted by different amounts and in different directions.

Syntax: result = EOSHIFT (*array*, *shift* [, *boundary*] [, *dim*])

Class: Transformational function; Generic

Arguments:

array Must be an array (of any data type).

shift Must be a scalar integer or an array with a rank that is one less than *array*, and shape (d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n), where (d_1, d_2, ..., d_n) is the shape of *array*.

boundary (opt) Must have the same type and kind parameters as *array*. It must be a scalar or an array with a rank that is one less than *array*, and shape (d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n).
If *boundary* is not specified, it is assumed to have the following default values (depending on the data type of *array*):

ARRAY Type	BOUNDARY Value
Integer	0
Real	0.0
Complex	(0.0, 0.0)
Logical	false
Character(len)	len blanks

dim (opt) Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of *array*. If *dim* is omitted, it is assumed to be 1.

Results: The result is an array with the same type and kind parameters, and shape as *array*.

If *array* has rank one, the same shift is applied to each element. If an element is shifted off one end of the array, the *boundary* value is placed at the other end the array.

If *array* has rank greater than one, each section (s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n) of the result is shifted as follows:

- By the value of *shift*, if *shift* is scalar
- According to the corresponding value in *shift*(s_1, s_2,..., s_{DIM-1}, s_{DIM+1},..., s_n), if *shift* is an array

If an element is shifted off one end of a section, the *boundary* value is placed at the other end of the section.

The value of *shift* determines the amount and direction of the end- off shift. A positive *shift* value causes a shift to the left (in rows) or up (in columns). A negative *shift* value causes a shift to the right (in rows) or down (in columns).

Examples

V is the array (1, 2, 3, 4, 5, 6).

EOSHIFT (V, SHIFT=2) shifts the elements in V to the *left* by 2 positions, producing the value (3, 4, 5, 6, 0, 0). 1 and 2 are shifted off the beginning and two elements with the default BOUNDARY value are placed at the end.

EOSHIFT (V, SHIFT= -3, BOUNDARY= 99) shifts the elements in V to the *right* by 3 positions, producing the value (99, 99, 99, 1, 2, 3). 4, 5, and 6 are shifted off the end and three elements with BOUNDARY value 99 are placed at the beginning.

M is the character array $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$.

EOSHIFT (M, SHIFT = 1, BOUNDARY = '*', DIM = 2) produces the result $\begin{bmatrix} 2 & 3 & * \\ 5 & 6 & * \\ 8 & 9 & * \end{bmatrix}$.

Each element in rows 1, 2, and 3 is shifted to the *left* by 1 position. This causes the first element in each row to be shifted off the beginning, and the BOUNDARY value to be placed at the end.

EOSHIFT (M, SHIFT = –1, DIM = 1) produces the result $\begin{bmatrix} 0 & 0 & 0 \\ 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$.

Each element in columns 1, 2, and 3 is shifted *down* by 1 position. This causes the last element in each column to be shifted off the end and the BOUNDARY value to be placed at the beginning.

EOSHIFT (M, SHIFT = (/1, –1, 0/), BOUNDARY = (/ '*', '?', '/' /), DIM = 2) produces the result $\begin{bmatrix} 2 & 3 & * \\ ? & 4 & 5 \\ 7 & 8 & 9 \end{bmatrix}$.

Each element in row 1 is shifted to the *left* by 1 position, causing the first element to be shifted off the beginning and the BOUNDARY value * to be placed at the end. Each element in row 2 is shifted to the *right* by 1 position, causing the last element to be shifted off the end and the BOUNDARY value ? to be placed at the beginning. No element in row 3 is shifted at all, so the specified BOUNDARY value is not used.

EPSILON

Description: Returns a positive model number that is almost negligible compared to unity in the model representing real numbers.

Syntax: result = EPSILON (*x*)

Class: Inquiry function; Generic

Arguments: *x* must be of type real; it can be scalar or array valued.

Results: The result is a scalar of the same type and kind parameters as *x*. The result has the value b^{1-p}. Parameters *b* and *p* are defined in "Model for Real Data".

Example

If X is of type REAL(4), EPSILON (X) has the value 2^{-23}.

ERF

Description: Returns the error function of an argument.

Syntax: result = ERF (*x*)

Class: Elemental function; Generic

Arguments: *x* must be of type real.

Results: The result type is the same as *x*. The result is in the range –1 to 1.

ERF returns the error function of *x* defined as follows:

$$\frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}\,dt$$

Specific Name	Argument Type	Result Type
ERF	REAL(4)	REAL(4)
DERF	REAL(8)	REAL(8)
QERF	REAL(16)	REAL(16)

Example

ERF (1.0) has the value 0.842700794.

ERFC

Description: Returns the complementary error function of an argument.

Syntax: result = ERFC (x)

Class: Elemental function; Generic

Arguments: x must be of type real.

Results: The result type is the same as x. The result is in the range 0 to 2.

ERFC returns 1 – ERF(x) and is defined as follows:

$$\frac{2}{\sqrt{\pi}}\int_x^\infty e^{-t^2}\,dt$$

ERFC is provided because of the extreme loss of relative accuracy if ERF(x) is called for large x and the result is subtracted from 1.

Specific Name	Argument Type	Result Type
ERFC	REAL(4)	REAL(4)
DERFC	REAL(8)	REAL(8)
QERFC	REAL(16)	REAL(16)

Example

ERFC (1.0) has the value 0.1572992057.

ERRSNS

Description: Returns information about the most recently detected I/O system error condition.

Syntax: CALL ERRSNS ([*io_err*] [, *sys_err*] [, *stat*] [, *unit*] [, *cond*])

Class: Subroutine

Arguments: There are five optional arguments:

io_err (opt) Is an integer variable or array element that stores the most recent Run-Time Library error number that occurred during program execution. (For a listing of error numbers, see your user's guide.)

A zero indicates no error has occurred since the last call to ERRSNS or since the start of program execution.

sys_err (opt) Is an integer variable or array element that stores the most recent system error number associated with *io_err*. This code is one of the following:

- On Linux* systems, it is an `errno` value. (See `errno(2)`.)
- On Windows* systems, it is the value returned by `GetLastError( )` at the time of the error.

stat (opt) Is an integer variable or array element that stores a status value that occurred during program execution. This value is zero.

unit (opt) Is an integer variable or array element that stores the logical unit number, if the last error was an I/O error.

cond (opt) Is an integer variable or array element that stores the actual processor value. This value is always zero.

If you specify INTEGER(2) arguments, only the low-order 16 bits of information are returned or adjacent data can be overwritten. Because of this, it is best to use INTEGER(4) arguments.

The saved error information is set to zero after each call to ERRSNS.

Example

Any of the arguments can be omitted. For example, the following is valid:

```
CALL ERRSNS (SYS_ERR, STAT, , UNIT)
```

EXIT

Description: Terminates program execution, closes all files, and returns control to the operating system.

Syntax: CALL EXIT ([*status*])

Class: Subroutine

Arguments: *status* (opt)
Is an integer argument you can use to specify the image exit-status value.

Example

CALL EXIT (100)

EXP

Description: Computes an exponential value.

Syntax: result = EXP (*x*)

Class: Elemental function; Generic

Arguments: *x* must be of type real or complex.

Results: The result type is the same as *x*. The value of the result is e^x. If *x* is of type complex, its imaginary part is regarded as a value in radians.

Specific Name	Argument Type	Result Type
EXP	REAL(4)	REAL(4)
DEXP	REAL(8)	REAL(8)
QEXP	REAL(16)	REAL(16)
CEXP[1]	COMPLEX(4)	COMPLEX(4)
CDEXP[2]	COMPLEX(8)	COMPLEX(8)
CQEXP	COMPLEX(16)	COMPLEX(16)

1. The setting of compiler options specifying real size can affect CEXP.
2. This function can also be specified as ZEXP.

Examples

EXP (2.0) has the value 7.389056.

EXP (1.3) has the value 3.669297.

EXPONENT

Description: Returns the exponent part of the argument when represented as a model number.

Syntax: result = EXPONENT (*x*)

Class: Elemental function; Generic

Arguments: *x* must be of type real.

Results: The result type is default integer. If *x* is not equal to zero, the result value is the exponent part of *x*. The exponent must be within default integer range; otherwise, the result is undefined.

If *x* is zero, the exponent of *x* is zero. For more information on the exponent part (*e*) in the real model, see "Model for Real Data".

Examples

EXPONENT (2.0) has the value 2.

If 4.1 is a REAL(4) value, EXPONENT (4.1) has the value 3.

FLOOR

Description: Returns the greatest integer less than or equal to its argument.

Syntax: result = FLOOR (*a* [, *kind*])

Class: Elemental function; Generic

Arguments: *a* must be of type real.

kind (opt)
Must be a scalar integer initialization expression. This argument is a Fortran 95 feature.

Results: The result type is integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The value of the result is equal to the greatest integer less than or equal to *a*.

The setting of compiler options specifying integer size can affect this function.

Examples

FLOOR (4.8) has the value 4.

FLOOR (–5.6) has the value –6.

FP_CLASS

Description: Returns the class of an IEEE* real (S_floating, T_floating, or X_floating) argument.

Syntax: result = FP_CLASS (*x*)

Class: Elemental function; Generic

Arguments: *x* must be of type real.

Results: The result type is default integer. The return value is one of the following:

Class of Argument	Return Value
Signaling NaN	FOR_K_FP_SNAN
Quiet NaN	FOR_K_FP_QNAN
Positive Infinity	FOR_K_FP_POS_INF
Negative Infinity	FOR_K_FP_NEG_INF
Positive Normalized Number	FOR_K_FP_POS_NORM
Negative Normalized Number	FOR_K_FP_NEG_NORM
Positive Denormalized Number	FOR_K_FP_POS_DENORM
Negative Denormalized Number	FOR_K_FP_NEG_DENORM
Positive Zero	FOR_K_FP_POS_ZERO
Negative Zero	FOR_K_FP_NEG_ZERO

The preceding return values are defined in file `fordef.f` on Linux* systems and file `fordef.for` on Windows* systems. For information on the location of these files, see your user's guide.

Example

FP_CLASS (4.0_8) has the value 4 (FOR_K_FP_POS_NORM).

FRACTION

Description: Returns the fractional part of the model representation of the argument value.

Syntax: result = FRACTION (*x*)

Class: Elemental function; Generic

Arguments: *x* must be of type real.

Results: The result type is the same as *x*. The result has the value $x * b^e$. Parameters *b* and *e* are defined in "Model for Real Data". If *x* has the value zero, the result has the value zero.

Examples

If 3.0 is a REAL(4) value, FRACTION (3.0) has the value 0.75.

FREE

Description: Frees a block of memory that is currently allocated.

Syntax: CALL FREE (*a*)

Class: Subroutine

Arguments: *a* must be of type INTEGER(4) on IA-32 processors; INTEGER(8) on Intel Itanium processors. This value is the starting address of the memory to be freed, previously allocated by MALLOC (see "MALLOC").

If the freed address was not previously allocated by MALLOC, or if an address is freed more than once, results are unpredictable.

Example

Consider the following:

```
INTEGER(4) SIZE
  REAL(4) STORAGE(*)
  POINTER (ADDR, STORAGE)     ! ADDR will point to STORAGE
  SIZE = 1024                 ! Size in bytes
  ADDR = MALLOC(SIZE)         ! Allocate the memory
  CALL FREE(ADDR)             ! Free it
```

GETARG

Description: Returns the specified command-line argument (where the command itself is argument number zero). This subroutine cannot be passed as an actual argument.

Syntax: CALL GETARG (*n*, *buffer* [, *status*])

Class: Subroutine

Arguments:

n Must be a scalar of type INTEGER(2) or INTEGER(4). This value is the position of the command-line argument to retrieve. The command itself is argument number 0.

buffer Must be a scalar of type default character. Its value is the returned command-line argument.

status (opt) Must be a scalar and have the same type and kind parameters as *n*. If specified, its value is the returned completion status.

If there were no errors, *status* returns the number of characters in the retrieved command-line argument before truncation or blank-padding. (That is, *status* is the original number of characters in the command-line argument.) Errors return a value of –1. Errors include specifying an argument position less than 0 or greater than the value returned by IARGC.

GETARG returns the *n*th command-line argument. If *n* is zero, the name of the executing program file is returned.

GETARG returns command-line arguments as they were entered. There is no case conversion.

If the command-line argument is shorter than *buffer*, GETARG pads *buffer* on the right with blanks. If the argument is longer than *buffer*, GETARG truncates the argument on the right. If there is an error, GETARG fills *buffer* with blanks.

Example

Assume a command-line invocation of `PROG1 -g -c -a`, and that *buffer* is at least five characters long. The following calls to GETARG return the corresponding arguments in *buffer* and *status*:

Statement	String returned in *buffer*	Length returned in *status*
CALL GETARG (0, buffer, status)	PROG1	5
CALL GETARG (1, buffer)	-g	undefined
CALL GETARG (2, buffer, status)	-c	2
CALL GETARG (3, buffer)	-a	undefined
CALL GETARG (4, buffer, status)	all blanks	–1

See Also

- "IARGC"
- "NARGS"

HUGE

Description: Returns the largest number in the model representing the same type and kind parameters as the argument.

Syntax: result = HUGE (*x*)

Class: Inquiry function; Generic

Arguments: *x* must be of type integer or real; it can be scalar or array valued.

Results: The result is a scalar of the same type and kind parameters as *x*. If *x* is of type integer, the result has the value $r^q - 1$. If *x* is of type real, the result has the value $(1 - b^{-p})b^{emax}$.

Integer parameters *r* and *q* are defined in "Model for Integer Data"; real parameters *b*, *p*, and e_{max} are defined in "Model for Real Data".

Example

If X is of type REAL(4), HUGE (X) has the value $(1 - 2^{-24}) \times 2^{128}$.

IACHAR

Description: Returns the position of a character in the ASCII character set, even if the processor's default character set is different. In Intel Fortran, IACHAR is equivalent to the ICHAR function.

Syntax: result = IACHAR (*c*)

Class: Elemental function; Generic

Arguments: *c* must be of type character of length 1.

Results: The result type is default integer. If *c* is in the ASCII collating sequence, the result is the position of *c* in that sequence and satisfies the inequality $(0 \le \mathrm{IACHAR}(c) \le 127)$.

The results must be consistent with the LGE, LGT, LLE, and LLT lexical comparison functions. For example, if LLE(C, D) is true, IACHAR(C) .LE. IACHAR(D) is also true.

Examples

IACHAR ('Y') has the value 89.

IACHAR ('%') has the value 37.

IAND

Description: Performs a logical AND on corresponding bits. This function can also be specified as AND.

Syntax: result = IAND (*i*, *j*)

Class: Elemental function; Generic

Arguments:

i Must be of type integer.

j Must be of type integer with the same kind parameter as *i*.

Results: The result type is the same as I. The result value is derived by combining *i* and *j* bit-by-bit according to the following :

I	J	IAND(I, J)
1	1	1
1	0	0
0	1	0
0	0	0

The model for the interpretation of an integer value as a sequence of bits is shown in "Model for Bit Data".

Specific Name	Argument Type	Result Type
BIAND	INTEGER(1)	INTEGER(1)
IIAND[1]	INTEGER(2)	INTEGER(2)
JIAND	INTEGER(4)	INTEGER(4)
KIAND	INTEGER(8)	INTEGER(8)

1. Or HIAND

Examples

IAND (2, 3) has the value 2.

IAND (4, 6) has the value 4.

IARGC

Description: Returns the index of the last command-line argument. It cannot be passed as an actual argument. This function can also be specified as IARG or NUMARG.

Syntax: result = IARGC ( )

Class: Inquiry function; Specific

Arguments: None.

Results: The result type is INTEGER(4). The result is the index of the last command-line argument, which is also the number of arguments on the command line. The command is not included in the count. For example, IARGC returns 3 for the command-line invocation of `PROG1 -g -c -a`.

IARGC returns a value that is 1 less than that returned by NARGS.

Example

Consider the following:

```
integer(4) no_of_arguments
no_of_arguments = IARGC ()
print *, 'total command line arguments are ', no_of_arguments
```

For a command-line invocation of `PROG1 -g -c -a`, the program above prints:

```
total command line arguments are 3
```

See Also

- "GETARG"
- "NARGS"

IARGPTR

Description: Returns a pointer to the actual argument list for the current routine.

Syntax: result = IARGPTR ( )

Class: Inquiry function; Specific

Arguments: None.

Results: The result type is INTEGER(4) on IA-32 processors; INTEGER(8) on Intel Itanium processors. The actual argument list is an array of values of the same type.

An argument count is not present and the first element has the address of the first argument.

Formal (dummy) arguments that can be omitted must be declared VOLATILE. For more information, see "VOLATILE Attribute and Statement".

Example

Consider the following:

```
WRITE (*,'(" Address of argument list is ",Z16.8)') IARGPTR( )
```

IBCHNG

Description: Reverses the value of a specified bit in an integer.

Syntax: result = IBCHNG (*i*, *pos*)

Class: Elemental function; Generic

Arguments:

i Must be of type integer. This argument contains the bit to be reversed.

pos Must be of type integer. This argument is the position of the bit to be changed.

The rightmost (least significant) bit of *i* is in position 0.

Results: The result type is the same as *i*. The result is equal to *i* with the bit in position *pos* reversed.

For more information on bit functions, see "Bit Functions".

Example

Consider the following:

```
INTEGER J, K
J = IBCHNG(10, 2)       ! returns 14 = 1110
K = IBCHNG(10, 1)       ! returns  8 = 1000
```

IBCLR

Description: Clears one bit to zero.

Syntax: result = IBCLR (*i*, *pos*)

Class: Elemental function; Generic

Arguments:

i Must be of type integer.

pos Must be of type integer. It must not be negative and it must be less than BIT_SIZE (*i*).

The rightmost (least significant) bit of *i* is in position 0.

Results: The result type is the same as *i*. The result has the value of the sequence of bits of *i*, except that bit *pos* of *i* is set to zero.

For more information on bit functions, see "Bit Functions".

The model for the interpretation of an integer value as a sequence of bits is shown in "Model for Bit Data".

Specific Name	Argument Type	Result Type
BBCLR	INTEGER(1)	INTEGER(1)
IIBCLR[1]	INTEGER(2)	INTEGER(2)
JIBCLR	INTEGER(4)	INTEGER(4)
KIBCLR	INTEGER(8)	INTEGER(8)

1. Or HBCLR.

Examples

IBCLR (18, 1) has the value 16.

If V has the value (1, 2, 3, 4), the value of IBCLR (POS = V, I = 15) is (13, 11, 7, 15).

IBITS

Description: Extracts a sequence of bits (a bit field).

Syntax: result = IBITS (*i*, *pos*, *len*)

Class: Elemental function; Generic

Arguments:

i Must be of type integer.

pos Must be of type integer. It must not be negative and *pos* + *len* must be less than or equal to BIT_SIZE (*i*).

The rightmost (least significant) bit of *i* is in position 0.

len Must be of type integer. It must not be negative.

Results: The result type is the same as *i*. The result has the value of the sequence of *len* bits in *i*, beginning at *pos* right-adjusted and with all other bits zero.

For more information on bit functions, see "Bit Functions".

The model for the interpretation of an integer value as a sequence of bits is shown in "Model for Bit Data".

Specific Name	Argument Type	Result Type
BBITS	INTEGER(1)	INTEGER(1)
IIBITS[1]	INTEGER(2)	INTEGER(2)
JIBITS	INTEGER(4)	INTEGER(4)
KIBITS	INTEGER(8)	INTEGER(8)

1. Or HBITS.

Examples

IBITS (12, 1, 4) has the value 6.

IBITS (10, 1, 7) has the value 5.

IBSET

Description: Sets one bit to 1.

Syntax: result = IBSET (*i*, *pos*)

Class: Elemental function; Generic

Arguments:

i Must be of type integer.

pos Must be of type integer. It must not be negative and it must be less than BIT_SIZE (*i*).

The rightmost (least significant) bit of *i* is in position 0.

Results: The result type is the same as *i*. The result has the value of the sequence of bits of *i*, except that bit *pos* of *i* is set to 1.

For more information on bit functions, see "Bit Functions".

The model for the interpretation of an integer value as a sequence of bits is shown in "Model for Bit Data".

Specific Name	Argument Type	Result Type
BBSET	INTEGER(1)	INTEGER(1)
IIBSET[1]	INTEGER(2)	INTEGER(2)
JIBSET	INTEGER(4)	INTEGER(4)
KIBSET	INTEGER(8)	INTEGER(8)

1. Or HBSET

Examples

IBSET (8, 1) has the value 10.

If V has the value (1, 2, 3, 4), the value of IBSET (POS = V, I = 2) is (2, 6, 10, 18).

ICHAR

Description: Returns the position of a character in the processor's character set.

Syntax: result = ICHAR (*c* [, *kind*])

Class: Elemental function; Generic

Arguments:

c Must be of type character of length 1.

kind (opt) Must be a scalar integer initialization expression.

Results: The result type is integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer.

The result value is the position of *c* in the processor's character set. Argument *c* is in the range zero to n – 1, where *n* is the number of characters in the character set.

For any characters C and D (capable of representation in the processor), C .LE. D is true only if ICHAR(C) .LE. ICHAR(D) is true, and C .EQ. D is true only if ICHAR(C) .EQ. ICHAR(D) is true.

Specific Name	Argument Type	Result Type
	CHARACTER	INTEGER(2)
ICHAR[1]	CHARACTER	INTEGER(4)
	CHARACTER	INTEGER(8)

1. This specific function cannot be passed as an actual argument.

Examples

ICHAR ('W') has the value 87.

ICHAR ('#') has the value 35.

IDATE

Description: Returns three integer values representing the current month, day, and year.

Syntax: CALL IDATE (*i*, *j*, *k*)

Class: Subroutine

Arguments:

i Must be of type INTEGER(4). The current month.

j Must be of type INTEGER(4). The current day.

k Must be of type INTEGER(4). The current year.

The current month is returned in *i*; the current day in *j*. The last two digits of the current year are returned in *k*.

CAUTION. *The two-digit year return value may cause problems with the year 2000 or later. Use* DATE_AND_TIME *instead (see* "DATE_AND_TIME"*).*

Example

If the current date is September 16, 1996, the values of the integer variables upon return are: I = 9, J = 16, and K = 96.

IEOR

Description: Performs an exclusive OR on corresponding bits. This function can also be specified as XOR or IXOR.

Syntax: result = IEOR (*i*, *j*)

Class: Elemental function; Generic

Arguments:

i Must be of type integer.

j Must be of type integer with the same kind parameter as *i*.

Results: The result type is the same as *i*. The result value is derived by combining *i* and *j* bit-by-bit according to the following truth table:

I	J	IEOR(I, J)
1	1	0
1	0	1
0	1	1
0	0	0

The model for the interpretation of an integer value as a sequence of bits is shown in "Model for Bit Data".

Specific Name	Argument Type	Result Type
BIEOR[1]	INTEGER(1)	INTEGER(1)
IIEOR[2]	INTEGER(2)	INTEGER(2)
JIEOR[3]	INTEGER(4)	INTEGER(4)
KIEOR[4]	INTEGER(8)	INTEGER(8)

1. Or BIXOR
2. Or HIEOR, HIXOR, or IIXOR
3. Or JIXOR
4. For compatibility, this specific function can also be specified as IXOR.

Example

IEOR (12, 7) has the value 11; binary 1100 exclusive OR with binary 0111 is binary 1011.

ILEN

Description: Returns the length (in bits) of the two's complement representation of an integer.

Syntax: result = ILEN (*i*)

Class: Elemental function; Generic

Arguments: *i* must be of type integer.

Results: The result type is the same as *i*. The result value is (LOG_2(I + 1)) if *i* is not negative; otherwise, the result value is (LOG_2(–I)).

Examples

ILEN (4) has the value 3.

ILEN (–4) has the value 2.

INDEX

Description: Returns the starting position of a substring within a string.

Syntax: result = INDEX (*string*, *substring* [, *back*] [, *kind*])

Class: Elemental function; Generic

Arguments:

string Must be of type character.

substring Must be of type character.

back (opt) Must be of type logical.

kind (opt) Must be a scalar integer initialization expression.

Results: The result type is integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

If *back* does not appear (or appears with the value false), the value returned is the minimum value of I such that *string* (I : I + LEN (*substring*) – 1) = *substring* (or zero if there is no such value). If LEN (*string*) < LEN (*substring*), zero is returned. If LEN (*substring*) = zero, 1 is returned.

If *back* appears with the value true, the value returned is the maximum value of I such that *string* (I : I + LEN (*substring*) – 1) = *substring* (or zero if there is no such value). If LEN(*string*) < LEN (*substring*), zero is returned. If LEN (*substring*) = zero, LEN (*string*) + 1 is returned.

Specific Name	Argument Type	Result Type
INDEX[1]	CHARACTER	INTEGER(4)
	CHARACTER	INTEGER(8)

1. The setting of compiler options specifying integer size can affect this function.

Examples

INDEX ('FORTRAN', 'O', BACK = .TRUE.) has the value 2.

INDEX ('XXXX', " ", BACK = .TRUE.) has the value 5.

INT

De cription: Converts a value to integer type.

Syntax: result = INT (*a* [, *kind*])

Class: Elemental function; Generic

Arguments:

a Must be of type integer, real, or complex.

kind (opt) Must be a scalar integer initialization expression.

Results: The result type is integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is shown in the following table. If the processor cannot represent the result value in the kind of the result, the result is undefined.

Functions that cause conversion of one data type to another type have the same affect as the implied conversion in assignment statements.

The result value depends on the type and absolute value of *a*, as follows:

- If *a* is of type integer, INT (*a*) = *a*.
- If *a* is of type real and $|a| < 1$, INT (*a*) has the value zero.
  If *a* is of type real and $|a| \geq 1$, INT (*a*) is the integer whose magnitude is the largest integer that does not exceed the magnitude of *a* and whose sign is the same as the sign of *a*.
- If *a* is of type complex, INT (*a*) = *a* is the value obtained by applying the preceding rules (for a real argument) to the real part of *a*.

Specific Name[1]	Argument Type	Result Type
	INTEGER(1), INTEGER(2), INTEGER(4)	INTEGER(4)
	INTEGER(1), INTEGER(2), INTEGER(4), INTEGER(8)	INTEGER(8)
IJINT	INTEGER(4)	INTEGER(2)
IIFIX[2]	REAL(4)	INTEGER(2)
IINT	REAL(4)	INTEGER(2)
IFIX[3,4]	REAL(4)	INTEGER(4)
JFIX	INTEGER(1), INTEGER(2), INTEGER(4), INTEGER(8), REAL(4), REAL(8), REAL(16), COMPLEX(4), COMPLEX(8), COMPLEX(16)	INTEGER(4)
INT[5,6,7]	REAL(4)	INTEGER(4)
KIFIX	REAL(4)	INTEGER(8)
KINT	REAL(4)	INTEGER(8)
IIDINT	REAL(8)	INTEGER(2)
IDINT[6,8]	REAL(8)	INTEGER(4)

Specific Name[1]	Argument Type	Result Type
KIDINT	REAL(8)	INTEGER(8)
IIQINT	REAL(16)	INTEGER(2)
IQINT[6,9]	REAL(16)	INTEGER(4)
KIQINT	REAL(16)	INTEGER(8)
INT1[10]	INTEGER(1), INTEGER(2), INTEGER(4), INTEGER(8), REAL(4), REAL(8), REAL(16), COMPLEX(4), COMPLEX(8), COMPLEX(16)	INTEGER(1)
INT2[10]	INTEGER(1), INTEGER(2), INTEGER(4), INTEGER(8), REAL(4), REAL(8), REAL(16), COMPLEX(4), COMPLEX(8), COMPLEX(16)	INTEGER(2)
INT4[10]	INTEGER(1), INTEGER(2), INTEGER(4), INTEGER(8), REAL(4), REAL(8), REAL(16), COMPLEX(4), COMPLEX(8), COMPLEX(16)	INTEGER(4)
INT8[10]	INTEGER(1), INTEGER(2), INTEGER(4), INTEGER(8), REAL(4), REAL(8), REAL(16), COMPLEX(4), COMPLEX(8), COMPLEX(16)	INTEGER(8)

1. These specific functions cannot be passed as actual arguments.
2. This function can also be specified as HFIX.
3. The setting of compiler options specifying integer size or real size can affect IFIX.
4. For compatibility with older versions of Fortran, IFIX can also be specified as a generic function.
5. Or JINT.
6. The setting of compiler options specifying integer size can affect INT, IDINT, and IQINT.
7. OR JIFIX.
8. Or JIDINT. For compatibility with older versions of Fortran, IDINT can also be specified as a generic function.
9. Or JIQINT. For compatibility with older versions of Fortran, IQINT can also be specified as a generic function.
10. For compatibility, these functions can be specified as generic functions.

Examples

INT (–4.2) has the value –4.

INT (7.8) has the value 7.

INT_PTR_KIND

Description: Returns the INTEGER KIND that will hold an address. This is a specific function that has no generic function associated with it. It cannot be passed as an actual argument.

Syntax: result = INT_PTR_KIND ( )

Class: Inquiry function; Specific

Arguments: None.

Results: The result type is default integer. The result is a scalar with the value equal to the value of the kind parameter of the integer data type that can represent an address on the host platform.

The result value is 4 on IA-32 processors; 8 on Intel Itanium processors.

Example

Consider the following:

```
REAL A(100)
POINTER (P, A)
INTEGER (KIND=INT_PTR_KIND()) SAVE_P
P = MALLOC (400)
SAVE_P = P
```

INUM

Description: Converts a character string to an INTEGER(2) value.

Syntax: result = INUM (*i*)

Class: Elemental function; Specific

Arguments: *i* must be of type character.

Results: The result type is INTEGER(2). The result value is the INTEGER(2) value represented by the character string *i*.

Example

INUM ("451.92") has the value 451 of type INTEGER(2).

IOR

Description: Performs an inclusive OR on corresponding bits. This function can also be specified as OR.

Syntax: result = IOR (*i*, *j*)

Class: Elemental function; Generic

Arguments:

i Must be of type integer.

j Must be of type integer with the same kind parameter as *i*.

Results: The result type is the same as *i*. The result value is derived by combining *i* and *j* bit-by-bit according to the following truth table:

I	J	IOR(I, J)
1	1	1
1	0	1
0	1	1
0	0	0

The model for the interpretation of an integer value as a sequence of bits is shown in "Model for Bit Data".

Specific Name	Argument Type	Result Type
BIOR	INTEGER(1)	INTEGER(1)
IIOR[1]	INTEGER(2)	INTEGER(2)
JIOR	INTEGER(4)	INTEGER(4)
KIOR	INTEGER(8)	INTEGER(8)

1. Or HIOR.

Examples

IOR (1, 4) has the value 5.

IOR (1, 2) has the value 3.

ISHA

Description: Arithmetically shifts an integer left or right by a specified number of bits.

Syntax: result = ISHA (*i*, *shift*)

Class: Elemental function; Generic

Arguments:

i Must be of type integer. This argument is the value to be shifted.

shift Must be of type integer. This argument is the direction and distance of shift.

Positive shifts are left (toward the most significant bit); negative shifts are right (toward the least significant bit).

Results: The result type is the same as *i*. The result is equal to *i* shifted arithmetically by *shift* bits.

If *shift* is positive, the shift is to the left; if *shift* is negative, the shift is to the right. If *shift* is zero, no shift is performed.

Bits shifted out from the left or from the right, as appropriate, are lost. If the shift is to the left, zeros are shifted in on the right. If the shift is to the right, copies of the sign bit (0 for non-negative *i*; 1 for negative *i*) are shifted in on the left.

The kind of integer is important in arithmetic shifting because sign varies among integer representations (see the following example). If you want to shift a one-byte or two-byte argument, you must declare it as INTEGER(1) or INTEGER(2).

Example

Consider the following:

```
INTEGER(1) i, res1
INTEGER(2) j, res2
i = -128              ! equal to  10000000
j = -32768            ! equal to  10000000 00000000
res1  = ISHA (i, -4)  ! returns 11111000 = -8
res2  = ISHA (j, -4)  ! returns 11111000 00000000 = -2048
```

ISHC

Description: Rotates an integer left or right by specified number of bits. Bits shifted out one end are shifted in the other end. No bits are lost.

Syntax: result = ISHC (*i*, *shift*)

Class: Elemental function; Generic

Arguments:

i Must be of type integer. This argument is the value to be rotated.

shift Must be of type integer. This argument is the direction and distance of rotation.

Positive rotations are left (toward the most significant bit); negative rotations are right (toward the least significant bit).

Results: The result type is the same as *i*. The result is equal to *i* circularly rotated by *shift* bits.

The kind of integer is important in circular shifting. With an INTEGER(4) argument, all 32 bits are shifted. If you want to rotate a one-byte or two-byte argument, you must declare it as INTEGER(1) or INTEGER(2).

Example

Consider the following:

```
INTEGER(1) i, res1
INTEGER(2) j, res2
i = 10                  ! equal to  00001010
j = 10                  ! equal to  00000000 00001010
res1  = ISHC (i, -3)    ! returns 01000001 =  65
res2  = ISHC (j, -3)    ! returns 01000000 00000001 = 16385
```

ISHFT

Description: Performs a logical shift.

Syntax: result = ISHFT (*i*, *shift*)

Class: Elemental function; Generic

Arguments:

i Must be of type integer.

shift Must be of type integer. The absolute value for *shift* must be less than or equal to BIT_SIZE (*i*).

Results: The result type is the same as *i*. The result has the value obtained by shifting the bits of *i* by *shift* positions. If *shift* is positive, the shift is to the left; if *shift* is negative, the shift is to the right. If *shift* is zero, no shift is performed.

Bits shifted out from the left or from the right, as appropriate, are lost. Zeros are shifted in from the opposite end.

ISHFT with a positive shift can also be specified as LSHIFT (or LSHFT). ISHFT with a negative shift can also be specified as RSHIFT (or RSHFT) with | shift |.

For more information on bit functions, see "Bit Functions".

The model for the interpretation of an integer value as a sequence of bits is shown in "Model for Bit Data".

Specific Name	Argument Type	Result Type
BSHFT	INTEGER(1)	INTEGER(1)
IISHFT[1]	INTEGER(2)	INTEGER(2)
JISHFT	INTEGER(4)	INTEGER(4)
KISHFT	INTEGER(8)	INTEGER(8)

1. Or HSHFT.

Examples

ISHFT (2, 1) has the value 4.

ISHFT (2, –1) has the value 1.

ISHFTC

Description: Performs a circular shift of the rightmost bits.

Syntax: result = ISHFTC (*i*, *shift* [, *size*])

Class: Elemental function; Generic

Arguments:

i Must be of type integer.

shift Must be of type integer. The absolute value for *shift* must be less than or equal to *size*.

size (opt) Must be of type integer. The value of *size* must be positive and must not exceed BIT_SIZE(*i*). If *size* is omitted, it is assumed to have the value of BIT_SIZE(*i*).

Results: The result type is the same as *i*. The result value is obtained by circular shifting the *size* rightmost bits of *i* by *shift* positions. If *shift* is positive, the shift is to the left; if *shift* is negative, the shift is to the right. If *shift* is zero, no shift is performed.

No bits are lost. Bits in *i* beyond the value specified by *size* are unaffected. For more information on bit functions, see "Bit Functions".

The model for the interpretation of an integer value as a sequence of bits is shown in "Model for Bit Data".

Specific Name	Argument Type	Result Type
BSHFTC	INTEGER(1)	INTEGER(1)
IISHFTC[1]	INTEGER(2)	INTEGER(2)

Specific Name	Argument Type	Result Type
JISHFTC	INTEGER(4)	INTEGER(4)
KISHFTC	INTEGER(8)	INTEGER(8)

1. Or HSHFTC.

Examples

ISHFTC (4, 2, 4) has the value 1.

ISHFTC (3, 1, 3) has the value 6.

ISHL

Description: Logically shifts an integer left or right by the specified bits. Zeros are shifted in from the opposite end.

Syntax: result = ISHL (*i*, *shift*)

Class: Elemental function; Generic

Arguments:

i Must be of type integer. This argument is the value to be shifted.

shift Must be of type integer. This argument is the direction and distance of shift.

If positive, *i* is shifted left (toward the most significant bit). If negative, *i* is shifted right (toward the least significant bit).

Results: The result type is the same as *i*. The result is equal to *i* logically shifted by *shift* bits. Zeros are shifted in from the opposite end.

Unlike circular or arithmetic shifts, which can shift ones into the number being shifted, logical shifts shift in zeros only, regardless of the direction or size of the shift. The integer kind, however, still determines the end that bits are shifted out of, which can make a difference in the result (see the following example).

Example

Consider the following:

```
INTEGER(1) i, res1
INTEGER(2) j, res2
i = 10                  ! equal to  00001010
j = 10                  ! equal to  00000000 00001010
res1  = ISHL (i, 5)     ! returns 01000000 = 64
res2  = ISHL (j, 5)     ! returns 00000001 01000000 = 320
```

ISNAN

Description: Tests whether IEEE real (S_floating, T_floating, and X_floating) numbers are Not-a-Number (NaN) values.

Syntax: result = ISNAN (*x*)

Class: Elemental function; Generic

Arguments: *x* must be of type real.

Results: The result type is default logical. The result is .TRUE. if *x* is an IEEE NaN; otherwise, the result is .FALSE..

Example

Consider the following:

```
LOGICAL A
DOUBLE PRECISION B
...
A = ISNAN(B)
```

A is assigned the value .TRUE. if B is an IEEE NaN; otherwise, the value assigned is .FALSE..

JNUM

Description: Converts a character string to an INTEGER(4) value.

Syntax: result = JNUM (*i*)

Class: Elemental function; Specific

Arguments: *i* must be of type character.

Results: The result type is INTEGER(4). The result value is the integer value represented by the character string *i*.

Example

JNUM ("46616.725") has the value 46616 of type INTEGER(4).

KIND

Description: Returns the value of the kind type parameter of the argument. For more information on kind type parameters, see "Intrinsic Data Types".

Syntax: result = KIND (*x*)

Class: Inquiry function; Generic

Arguments: *x* can be of any intrinsic type.

Results: The result is a scalar of type default integer. The result has a value equal to the kind type parameter value of *x*.

Examples

KIND (0.0) has the kind value of default real type.

KIND (12) has the kind value of default integer type.

LBOUND

Description: Returns the lower bounds for all dimensions of an array, or the lower bound for a specified dimension.

Syntax: result = LBOUND (*array* [, *dim*] [, *kind*])

Class: Inquiry function; Generic

Arguments:

array Must be an array (of any data type). It must not be an allocatable array that is not allocated, or a disassociated pointer.

dim (opt) Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of *array.*

kind (opt) Must be a scalar integer initialization expression.

Results: The result type is integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

If *dim* is present, the result is a scalar. Otherwise, the result is a rank-one array with one element for each dimension of *array.* Each element in the result corresponds to a dimension of *array.*

If *array* is an array section or an array expression that is not a whole array or array structure component, each element of the result has the value 1.

If *array* is a whole array or array structure component, LBOUND (*array*, *dim*) has a value equal to the lower bound for subscript *dim* of *array* (if *dim* is nonzero or *array* is an assumed-size array of rank *dim*). Otherwise, the corresponding element of the result has the value 1.

The setting of compiler options specifying integer size can affect this function.

Examples

Consider the following:

```
REAL ARRAY_A (1:3, 5:8)
REAL ARRAY_B (2:8, -3:20)
```

LBOUND (ARRAY_A) is (1, 5). LBOUND (ARRAY_A, DIM=2) is 5.

LBOUND (ARRAY_B) is (2, –3). LBOUND (ARRAY_B (5:8, :)) is (1,1) because the arguments are array sections.

LEADZ

Description: Returns the number of leading zero bits in an integer.

Syntax: result = LEADZ (*i*)

Class: Elemental function; Generic

Arguments: *i* must be of type integer or logical.

Results: The result type is the same as *i*. The result value is the number of leading zeros in the binary representation of the integer *i*.

The model for the interpretation of an integer value as a sequence of bits is shown in "Model for Bit Data".

Example

Consider the following:

```
INTEGER*8 J, TWO
PARAMETER (TWO=2)
DO J= -1, 40
  TYPE *, LEADZ(TWO**J)  ! Prints 64 down to 23 (leading zeros)
ENDDO
END
```

LEN

Description: Returns the length of a character expression.

Syntax: result = LEN (*string* [, *kind*])

Class: Inquiry function; Generic

Arguments:

string Must be of type character; it can be scalar or array valued.

kind (opt) Must be a scalar integer initialization expression.

Results: The result is a scalar of type integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The result has a value equal to the number of characters in *string* (if it is scalar) or in an element of *string* (if it is array valued).

Specific Name	Argument Type	Result Type
LEN[1]	CHARACTER	INTEGER(4)
	CHARACTER	INTEGER(8)

1. The setting of compiler options specifying integer size can affect this function.

Example

Consider the following:

```
CHARACTER (15) C (50)
CHARACTER (25) D
```

LEN (C) has the value 15, and LEN (D) has the value 25.

LEN_TRIM

Description: Returns the length of the character argument without counting trailing blank characters.

Syntax: result = LEN_TRIM (*string* [, *kind*])

Class: Elemental function; Generic

Arguments:

string Must be of type character.

kind (opt) Must be a scalar integer initialization expression.

Results: The result is a scalar of type integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The result has a value equal to the number of characters remaining after any trailing blanks in *string* are removed. If the argument contains only blank characters, the result is zero.

The setting of compiler options specifying integer size can affect this function.

Examples

LEN_TRIM ('ΔΔΔ CΔΔ DΔΔΔ') has the value 7.

LEN_TRIM ('ΔΔΔΔΔ') has the value 0.

LGE

Description: Determines if a string is lexically greater than or equal to another string, based on the ASCII collating sequence, even if the processor's default collating sequence is different. In Intel Fortran, LGE is equivalent to the ≥ (.GE.) operator.

Syntax: result = LGE (*string_a*, *string_b*)

Class: Elemental function; Generic

Arguments:

string_a Must be of type character.

string_b Must be of type character.

Results: The result type is default logical. If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks, to the length of the longer string.

The result is true if the strings are equal, both strings are of zero length, or if *string_a* follows *string_b* in the ASCII collating sequence; otherwise, the result is false.

Specific Name	Argument Type	Result Type
LGE[1]	CHARACTER	LOGICAL(4)

1. This specific function cannot be passed as an actual argument.

Examples

LGE ('ONE', 'SIX') has the value false.

LGE ('TWO', 'THREE') has the value true.

LGT

Description: Determines whether a string is lexically greater than another string, based on the ASCII collating sequence, even if the processor's default collating sequence is different. In Intel Fortran, LGT is equivalent to the > (.GT.) operator.

Syntax: result = LGT (*string_a*, *string_b*)

Class: Elemental function; Generic

Arguments:

string_a Must be of type character.

string_b Must be of type character.

Results: The result type is default logical. If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks, to the length of the longer string.

The result is true if *string_a* follows *string_b* in the ASCII collating sequence; otherwise, the result is false. If both strings are of zero length, the result is also false.

Specific Name	Argument Type	Result Type
LGT[1]	CHARACTER	LOGICAL(4)

1. This specific function cannot be passed as an actual argument.

Examples

LGT ('TWO', 'THREE') has the value true.

LGT ('ONE', 'FOUR') has the value true.

LLE

Description: Determines whether a string is lexically less than or equal to another string, based on the ASCII collating sequence, even if the processor's default collating sequence is different. In Intel Fortran, LLE is equivalent to the ≤ (.LE.) operator.

Syntax: result = LLE (*string_a*, *string_b*)

Class: Elemental function; Generic

Arguments:

string_a Must be of type character.

string_b Must be of type character.

Results: The result type is default logical. If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks, to the length of the longer string.

The result is true if the strings are equal, both strings are of zero length, or if *string_a* precedes *string_b* in the ASCII collating sequence; otherwise, the result is false.

Specific Name	Argument Type	Result Type
LLE[1]	CHARACTER	LOGICAL(4)

1. This specific function cannot be passed as an actual argument.

Examples

LLE ('TWO', 'THREE') has the value false.

LLE ('ONE', 'FOUR') has the value false.

LLT

Description: Determines whether a string is lexically less than another string, based on the ASCII collating sequence, even if the processor's default collating sequence is different. In Intel Fortran, LLT is equivalent to the < (.LT.) operator.

Syntax: result = LLT (*string_a*, *string_b*)

Class: Elemental function; Generic

Arguments:

string_a Must be of type character.

string_b Must be of type character.

Results: The result type is default logical. If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks, to the length of the longer string.

The result is true if *string_a* precedes *string_b* in the ASCII collating sequence; otherwise, the result is false. If both strings are of zero length, the result is also false.

Specific Name	Argument Type	Result Type
LLT[1]	CHARACTER	LOGICAL(4)

1. This specific function cannot be passed as an actual argument.

Examples

LLT ('ONE', 'SIX') has the value true.

LLT ('ONE', 'FOUR') has the value false.

LOC

Description: Returns the internal address of a storage item. This function cannot be passed as an actual argument.

Syntax: result = LOC (*x*)

Class: Inquiry function; Generic

Arguments: *x* is a variable, an array or record field reference, a procedure, or a constant; it can be of any data type. It must not be the name of an internal procedure or statement function. If it is a pointer, it must be defined and associated with a target.

Results: The result type is INTEGER(4) on IA-32 processors; INTEGER(8) on Intel Itanium processors. The value of the result represents the address of the data object or, in the case of pointers, the address of its associated target. If the argument is not valid, the result is undefined.

This function serves the same purpose as the %LOC built-in function.

LOG

Description: Returns the natural logarithm of the argument.

Syntax: result = LOG (*x*)

Class: Elemental function; Generic

Arguments: *x* must be of type real or complex. If *x* is real, its value must be greater than zero. If *x* is complex, its value must not be zero.

Results: The result type is the same as *x*. The result value is approximately equal to $\log_e x$.

If the arguments are complex, the result is the principal value of imaginary part omega in the range $-\pi < \omega \leq \pi$. The imaginary part of the result is π if the real part of the argument is less than zero and the imaginary part of the argument is zero.

Specific Name	Argument Type	Result Type
ALOG[1,2]	REAL(4)	REAL(4)
DLOG	REAL(8)	REAL(8)
QLOG	REAL(16)	REAL(16)
CLOG	COMPLEX(4)	COMPLEX(4)
CDLOG[3]	COMPLEX(8)	COMPLEX(8)
CQLOG	COMPLEX(16)	COMPLEX(16)

1. This function can also be specified as LOG.
2. The setting of compiler options specifying real size can affect ALOG, LOG, and CLOG.
3. This function can also be specified as ZLOG.

Examples

LOG (8.0) has the value 2.079442.

LOG (25.0) has the value 3.218876.

LOG10

Description: Returns the common logarithm of the argument.

Syntax: result = LOG10 (*x*)

Class: Elemental function; Generic

Arguments: *x* must be of type real. The value of *x* must be greater than zero.

Results: The result type is the same as *x*. The result value is approximately equal to $\log_{10} x$.

Specific Name	Argument Type	Result Type
ALOG10[1]	REAL(4)	REAL(4)
DLOG10	REAL(8)	REAL(8)

Specific Name	Argument Type	Result Type
QLOG10	REAL(16)	REAL(16)

1. This function can also be specified as LOG10. The setting of compiler options specifying real size can affect ALOG10 and LOG10.

Examples

LOG10 (8.0) has the value 0.9030900.

LOG10 (15.0) has the value 1.176091.

LOGICAL

Description: Converts the logical value of the argument to a logical value with different kind parameters.

Syntax: result = LOGICAL (*l* [, *kind*])

Class: Elemental function; Generic

Arguments:

l Must be of type logical.

kind (opt) Must be a scalar integer initialization expression.

Results: The result type is logical. If *kind* is present, the kind parameter is that specified by *kind*; otherwise, the kind parameter is that of default logical. The result value is that of *l*.

The setting of compiler options specifying integer size can affect this function.

Examples

LOGICAL (L .OR. .NOT. L) has the value true and is of type default logical regardless of the kind parameter of logical variable L.

LOGICAL (.FALSE., 2) has the value false, with the kind parameter of INTEGER(KIND=2).

MALLOC

Description: Allocates a block of memory. This specific function cannot be passed as an actual argument.

Syntax: result = MALLOC (*i*)

Class: Elemental function; Specific

Arguments: *i* must be of type integer. This value is the size (in bytes) of memory to be allocated.

Results: The result type is INTEGER(4) on IA-32 processors; INTEGER(8) on Intel Itanium processors. The result is the starting address of the allocated memory. The memory allocated can be freed by using the FREE intrinsic function (see "FREE").

Example

Consider the following:

```
INTEGER(4) SIZE
REAL(4) STORAGE(*)
POINTER (ADDR, STORAGE)     ! ADDR will point to STORAGE
SIZE = 1024                 ! Size in bytes
ADDR = MALLOC(SIZE)         ! Allocate the memory
CALL FREE(ADDR)             ! Free it
```

MATMUL

Description: Performs matrix multiplication of numeric or logical matrices.

Syntax: result = MATMUL (*matrix_a*, *matrix_b*)

Class: Transformational function; Generic

Arguments:

matrix_a Must be an array of rank one or two. It must be of numeric (integer, real, or complex) or logical type.

matrix_b Must be an array of rank one or two. It must be of numeric type if *matrix_a* is of numeric type or logical type if *matrix_a* is logical type.

At least one argument must be of rank two. The size of the first (or only) dimension of *matrix_b* must equal the size of the last (or only) dimension of *matrix_a*.

Results: The result is an array whose type depends on the data type of the arguments, according to the rules described in "Data Type of Numeric Expressions". The rank and shape of the result depends on the rank and shapes of the arguments, as follows:

- If *matrix_a* has shape (n, m) and *matrix_b* has shape (m, k), the result is a rank-two array with shape (n, k).
- If *matrix_a* has shape (m) and *matrix_b* has shape (m, k), the result is a rank-one array with shape (k).
- If *matrix_a* has shape (n, m) and *matrix_b* has shape (m), the result is a rank-one array with shape (n).

If the arguments are of numeric type, element (i, j) of the result has the value SUM ((row i of *matrix_a*) * (column j of *matrix_b*)). If the arguments are of logical type, element (i, j) of the result has the value ANY ((row i of *matrix_a*) .AND. (column j of *matrix_b*)).

Examples

A is matrix $\begin{bmatrix} 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix}$, B is matrix $\begin{bmatrix} 2 & 3 \\ 3 & 4 \\ 4 & 5 \end{bmatrix}$, X is vector (1, 2), and Y is vector (1, 2, 3).

The result of MATMUL (A, B) is the matrix-matrix product AB with the value $\begin{bmatrix} 29 & 38 \\ 38 & 50 \end{bmatrix}$.

The result of MATMUL (X, A) is the vector-matrix product XA with the value (8, 11, 14).

The result of MATMUL (A, Y) is the matrix-vector product AY with the value (20, 26).

MAX

Description: Returns the maximum value of the arguments.

Syntax: result = MAX (*a1*, *a2* [, *a3*,...])

Class: Elemental function; Generic

Arguments: *a1*, *a2*, and *a3* (opt)
All must have the same type (integer or real) and kind parameters.

Results: For MAX0, AMAX1, DMAX1, QMAX1, IMAX0, JMAX0, and KMAX0, the result type is the same as the arguments. For MAX1, IMAX1, JMAX1, and KMAX1, the result type is integer. For AMAX0, AIMAX0, AJMAX0, and AKMAX0, the result is of type real. The value of the result is that of the largest argument.

Specific Name[1]	Argument Type	Result Type
	INTEGER(1)	INTEGER(1)
	INTEGER(1)	REAL(4)
IMAX0	INTEGER(2)	INTEGER(2)
AIMAX0	INTEGER(2)	REAL(4)
MAX0[2]	INTEGER(4)	INTEGER(4)
AMAX0[3,4]	INTEGER(4)	REAL(4)
KMAX0	INTEGER(8)	INTEGER(8)
AKMAX0	INTEGER(8)	REAL(4)
IMAX1	REAL(4)	INTEGER(2)
MAX1[4,5,6]	REAL(4)	INTEGER(4)
KMAX1	REAL(4)	INTEGER(8)
AMAX1[7]	REAL(4)	REAL(4)
DMAX1	REAL(8)	REAL(8)
QMAX1	REAL(16)	REAL(16)

1. These specific functions cannot be passed as actual arguments.
2. Or JMAX0.
3. Or AJMAX0. AMAX0 is the same as REAL(MAX).
4. In Fortran 95/90, AMAX0 and MAX1 are specific functions with no generic name. For compatibility with older versions of Fortran, these functions can also be specified as generic functions.
5. Or JMAX1. MAX1 is the same as INT(MAX).
6. The setting of compiler options specifying integer size can affect MAX1.
7. The setting of compiler options specifying real size can affect AMAX1.

Examples

MAX (2.0, –8.0, 6.0) has the value 6.0.

MAX (14, 32, –50) has the value 32.

MAXEXPONENT

Description: Returns the maximum exponent in the model representing the same type and kind parameters as the argument.

Syntax: result = MAXEXPONENT (*x*)

Class: Inquiry function; Generic

Arguments: *x* must be of type real; it can be scalar or array valued.

Results: The result is a scalar of type default integer. The result has the value e_{max}, as defined in "Model for Real Data".

Example

If X is of type REAL(4), MAXEXPONENT (X) has the value 128.

MAXLOC

Description: Returns the location of the maximum value of all elements in an array, a set of elements in an array, or elements in a specified dimension of an array.

Syntax: result = MAXLOC (*array* [, *dim*] [, *mask*] [, *kind*])

Class: Transformational function; Generic

Arguments:

array Must be an array of type integer or real.

dim (opt) Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of *array*. This argument is a Fortran 95 feature.

mask (opt) Must be a logical array that is conformable with *array*.

kind (opt) Must be a scalar integer initialization expression.

Results: The result is an array of type integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The following rules apply if *dim* is omitted:

- The array result has rank one and a size equal to the rank of *array.*
- If MAXLOC (*array*) is specified, the elements in the array result form the subscript of the location of the element with the maximum value in *array.* The *i*th subscript returned lies in the range 1 to e_i, where e_i is the extent of the *i*th dimension of *array.*
- If MAXLOC (*array*, MASK=*mask*) is specified, the elements in the array result form the subscript of the location of the element with the maximum value corresponding to the condition specified by *mask*.

The following rules apply if *dim* is specified:

- The array result has a rank that is one less than *array*, and shape (d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n), where (d_1, d_2, ..., d_n) is the shape of *array.*
- If *array* has rank one, MAXLOC (*array*, *dim* [,*mask*]) has a value equal to that of MAXLOC (*array* [,MASK = *mask*]). Otherwise, the value of element (s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n) of MAXLOC (*array*, *dim* [,*mask*]) is equal to MAXLOC (*array* (s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n) [,MASK = *mask* (s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n)]).

If more than one element has maximum value, the element whose subscripts are returned is the first such element, taken in array element order. If *array* has size zero, or every element of *mask* has the value .FALSE., the value of the result is undefined.

The setting of compiler options specifying integer size can affect this function.

Examples

The value of MAXLOC ((/3, 7, 4, 7/)) is (2), which is the subscript of the location of the first occurrence of the maximum value in the rank-one array.

A is the array $\begin{bmatrix} 4 & 0 & -3 & 2 \\ 3 & 1 & -2 & 6 \\ -1 & -4 & 5 & -5 \end{bmatrix}$.

MAXLOC (A, MASK=A .LT. 5) has the value (1, 1) because these are the subscripts of the location of the maximum value (4) that is less than 5.

MAXLOC (A, DIM=1) has the value (1, 2, 3, 2). 1 is the subscript of the location of the maximum value (4) in column 1; 2 is the subscript of the location of the maximum value (1) in column 2; and so forth.

MAXLOC (A, DIM=2) has the value (1, 4, 3). 1 is the subscript of the location of the maximum value in row 1; 4 is the subscript of the location of the maximum value in row 2; and so forth.

MAXVAL

Description: Returns the maximum value of all elements in an array, a set of elements in an array, or elements in a specified dimension of an array.

Syntax: result = MAXVAL (*array* [, *dim*] [, *mask*])

Class: Transformational function; Generic

Arguments:

array Must be an array of type integer or real.

dim (opt) Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of *array.*

mask (opt) Must be a logical array that is conformable with *array.*

Results: The result is an array or a scalar of the same data type as *array.*

The result is a scalar if *dim* is omitted or *array* has rank one.

The following rules apply if *dim* is omitted:

- If MAXVAL (*array*) is specified, the result has a value equal to the maximum value of all the elements in *array.*
- If MAXVAL (*array*, MASK=*mask*) is specified, the result has a value equal to the maximum value of the elements in *array* corresponding to the condition specified by *mask*.

The following rules apply if *dim* is specified:

- The array result has a rank that is one less than *array*, and shape (d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n), where (d_1, d_2, ..., d_n) is the shape of *array.*
- If *array* has rank one, MAXVAL (*array*, *dim* [,*mask*]) has a value equal to that of MAXVAL (*array* [,MASK = *mask*]). Otherwise, the value of element (s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n) of MAXVAL (*array*, *dim* [,*mask*]) is equal to MAXVAL (*array* (s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n) [,MASK = *mask* (s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n)]).

If *array* has size zero or if there are no true elements in *mask*, the result (if *dim* is omitted), or each element in the result array (if *dim* is specified), has the value of the negative number of the largest magnitude supported by the processor for numbers of the type and kind parameters of *array.*

Examples

The value of MAXVAL ((/2, 3, 4/)) is 4 because that is the maximum value in the rank-one array.

MAXVAL (B, MASK=B .LT. 0.0) finds the maximum value of the negative elements of B.

C is the array $\begin{bmatrix} 2 & 3 & 4 \\ 5 & 6 & 7 \end{bmatrix}$.

MAXVAL (C, DIM=1) has the value (5, 6, 7). 5 is the maximum value in column 1; 6 is the maximum value in column 2; and so forth.

MAXVAL (C, DIM=2) has the value (4, 7). 4 is the maximum value in row 1 and 7 is the maximum value in row 2.

MCLOCK

Description: Returns time accounting for a program.

Syntax: result = MCLOCK ( )

Class: Inquiry function; Specific

Arguments: None.

Results: The result type is integer. The result is the sum (in units of microseconds) of the current process's user time and the user and system time of all its child processes.

MERGE

Description: Selects between two values or between corresponding elements in two arrays, according to the condition specified by a logical mask.

Syntax: result = MERGE (*tsource*, *fsource*, *mask*)

Class: Elemental function; Generic

Arguments:

tsource Must be a scalar or array (of any data type).

fsource Must be a scalar or array of the same type and type parameters as *tsource*.

mask Must be a logical array.

Results: The result type is the same as *tsource*. The value of *mask* determines whether the result value is taken from *tsource* (if *mask* is true) or *fsource* (if *mask* is false).

Examples

For MERGE (1.0, 0.0, R < 0), R = –3 has the value 1.0, and R = 7 has the value 0.0.

TSOURCE is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$, FSOURCE is the array $\begin{bmatrix} 8 & 9 & 0 \\ 1 & 2 & 3 \end{bmatrix}$, and MASK is the array $\begin{bmatrix} F & T & T \\ T & T & F \end{bmatrix}$.

MERGE (TSOURCE, FSOURCE, MASK) produces the result: $\begin{bmatrix} 8 & 3 & 5 \\ 2 & 4 & 3 \end{bmatrix}$.

MIN

Description: Returns the minimum value of the arguments.

Syntax: result = MIN (*a1*, *a2* [, *a3*,...])

Class: Elemental function; Generic

Arguments: *a1*, *a2*, and *a3* (opt)
All must have the same type (integer or real) and kind parameters.

Results: For MIN0, AMIN1, DMIN1, QMIN1, IMIN0, JMIN0, and KMIN0, the result type is the same as the arguments. For MIN1, IMIN1, JMIN1, and KMIN1, the result type is integer. For AMIN0, AIMIN0, AJMIN0, and AKMIN0, the result is of type real. The value of the result is that of the smallest argument.

Specific Name[1]	Argument Type	Result Type
	INTEGER(1)	INTEGER(1)
	INTEGER(1)	REAL(4)
IMIN0	INTEGER(2)	INTEGER(2)
AIMIN0	INTEGER(2)	REAL(4)
MIN0[2]	INTEGER(4)	INTEGER(4)
AMIN0[3,4]	INTEGER(4)	REAL(4)
KMIN0	INTEGER(8)	INTEGER(8)
AKMIN0	INTEGER(8)	REAL(4)
IMIN1	REAL(4)	INTEGER(2)
MIN1[4,5,6]	REAL(4)	INTEGER(4)
KMIN1	REAL(4)	INTEGER(8)
AMIN1[7]	REAL(4)	REAL(4)

Specific Name[1]	Argument Type	Result Type
DMIN1	REAL(8)	REAL(8)
QMIN1	REAL(16)	REAL(16)

1. These specific functions cannot be passed as actual arguments.
2. Or JMIN0.
3. Or AJMIN0. AMIN0 is the same as REAL(MIN).
4. In Fortran 95/90, AMIN0 and MIN1 are specific functions with no generic name. For compatibility with older versions of Fortran, these functions can also be specified as generic functions.
5. Or JMIN1. MIN1 is the same as INT(MIN).
6. The setting of compiler options specifying integer size can affect MIN1.
7. The setting of compiler options specifying real size can affect AMIN1.

Examples

MIN (2.0, –8.0, 6.0) has the value –8.0.

MIN (14, 32, –50) has the value –50.

MINEXPONENT

Description: Returns the minimum exponent in the model representing the same type and kind parameters as the argument.

Syntax: result = MINEXPONENT (*x*)

Class: Inquiry function; Generic

Arguments: *x* must be of type real; it can be scalar or array valued.

Results: The result is a scalar of type default integer. The result has the value e_{min}, as defined in "Model for Real Data".

Example

If X is of type REAL(4), MINEXPONENT (X) has the value –125.

MINLOC

Description: Returns the location of the minimum value of all elements in an array, a set of elements in an array, or elements in a specified dimension of an array.

Syntax: result = MINLOC (*array* [, *dim*] [, *mask*] [, *kind*])

Class: Transformational function; Generic

Arguments:

array	Must be an array of type integer or real.
dim (opt)	Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of *array.* This argument is a Fortran 95 feature.
mask (opt)	Must be a logical array that is conformable with *array.*
kind (opt)	Must be a scalar integer initialization expression.

Results: The result is an array of type integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The following rules apply if *dim* is omitted:

- The array result has rank one and a size equal to the rank of *array.*
- If MINLOC (*array*) is specified, the elements in the array result form the subscript of the location of the element with the minimum value in *array.* The *i*th subscript returned lies in the range 1 to e_i, where e_i is the extent of the *i*th dimension of *array.*
- If MINLOC (*array*, MASK=*mask*) is specified, the elements in the array result form the subscript of the location of the element with the minimum value corresponding to the condition specified by *mask*.

The following rules apply if *dim* is specified:

- The array result has a rank that is one less than *array*, and shape (d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n), where (d_1, d_2, ..., d_n) is the shape of *array.*
- If *array* has rank one, MINLOC (*array*, *dim* [,*mask*]) has a value equal to that of MINLOC (*array* [,MASK = *mask*]). Otherwise, the value of element (s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n) of MINLOC (*array*, *dim* [,*mask*]) is equal to MINLOC (*array* (s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n) [,MASK = *mask* (s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n)]).

If more than one element has minimum value, the element whose subscripts are returned is the first such element, taken in array element order. If *array* has size zero, or every element of *mask* has the value .FALSE., the value of the result is undefined.

The setting of compiler options specifying integer size can affect this function.

Examples

The value of MINLOC ((/3, 1, 4, 1/)) is (2), which is the subscript of the location of the first occurrence of the minimum value in the rank-one array.

A is the array $\begin{bmatrix} 4 & 0 & -3 & 2 \\ 3 & 1 & -2 & 6 \\ -1 & -4 & 5 & -5 \end{bmatrix}$.

MINLOC (A, MASK=A .GT. –5) has the value (3, 2) because these are the subscripts of the location of the minimum value (-4) that is greater than –5.

MINLOC (A, DIM=1) has the value (3, 3, 1, 3). 3 is the subscript of the location of the minimum value (–1) in column 1; 3 is the subscript of the location of the minimum value (–4) in column 2; and so forth.

MINLOC (A, DIM=2) has the value (3, 3, 4). 3 is the subscript of the location of the minimum value (–3) in row 1; 3 is the subscript of the location of the minimum value (–2) in row 2; and so forth.

MINVAL

Description: Returns the minimum value of all elements in an array, a set of elements in an array, or elements in a specified dimension of an array.

Syntax: result = MINVAL (*array* [, *dim*] [, *mask*])

Class: Transformational function; Generic

Arguments:

array Must be an array of type integer or real.

dim (opt) Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of *array.*

mask (opt) Must be a logical array that is conformable with *array.*

Results: The result is an array or a scalar of the same data type as *array.*

The result is a scalar if *dim* is omitted or *array* has rank one.

The following rules apply if *dim* is omitted:

- If MINVAL (*array*) is specified, the result has a value equal to the minimum value of all the elements in *array.*
- If MINVAL (*array*, MASK=*mask*) is specified, the result has a value equal to the minimum value of the elements in *array* corresponding to the condition specified by *mask*.

The following rules apply if *dim* is specified:

- The array result has a rank that is one less than *array*, and shape (d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n), where (d_1, d_2, ..., d_n) is the shape of *array*.
- If *array* has rank one, MINVAL (*array*, *dim* [,*mask*]) has a value equal to that of MINVAL (*array* [,MASK = *mask*]). Otherwise, the value of element (s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n) of MINVAL (*array*, *dim* [,*mask*]) is equal to MINVAL (*array* (s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n) [,MASK = *mask* (s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n)]).

If *array* has size zero or if there are no true elements in *mask*, the result (if *dim* is omitted), or each element in the result array (if *dim* is specified), has the value of the positive number of the largest magnitude supported by the processor for numbers of the type and kind parameters of *array*.

Examples

The value of MINVAL ((/2, 3, 4/)) is 2 because that is the minimum value in the rank-one array.

The value of MINVAL (B, MASK=B .GT. 0.0) finds the minimum value of the positive elements of B.

C is the array $\begin{bmatrix} 2 & 3 & 4 \\ 5 & 6 & 7 \end{bmatrix}$.

MINVAL (C, DIM=1) has the value (2, 3, 4). 2 is the minimum value in column 1; 3 is the minimum value in column 2; and so forth.

MINVAL (C, DIM=2) has the value (2, 5). 2 is the minimum value in row 1 and 5 is the minimum value in row 2.

MM_PREFETCH

Description: Prefetches data from the specified address on one memory cache line.

Syntax: CALL MM_PREFETCH (*address* [, *hint*] [, *fault*] [, *exclusive*])

Class: Subroutine

Arguments:

address Is the name of a scalar or array; it can be of any type or rank. It specifies the address of the data on the cache line to prefetch.

hint (opt) Is an optional default integer constant with one of the following values:

Value	Prefetch Constant	Description
0	FOR_K_PREFETCH_T0	Prefetches into the L1 cache (and the L2 and the L3 cache). Use this for integer data.
1	FOR_K_PREFETCH_T1	Prefetches into the L2 cache (and the L3 cache); floating-point data is used from the L2 cache, not the L1 cache. Use this for real data.
2	FOR_K_PREFETCH_T2	Prefetches into the L2 cache (and the L3 cache); this line will be marked for early displacement. Use this if you are not going to reuse the cache line frequently.
3	FOR_K_PREFETCH_NTA	Prefetches into the L2 cache (but *not* the L3 cache); this line will be marked for early displacement. Use this if you are not going to reuse the cache line.

The preceding return values are defined in file `fordef.f` on Linux* systems and file `fordef.for` on Windows* systems. For information on the location of these files, see your user's guide.

If *hint* is omitted, 0 is assumed.

fault (opt) Is an optional default logical constant. If .TRUE. is specified, page faults are allowed to occur, if necessary; if .FALSE. is specified, page faults are not allowed to occur. If *fault* is omitted, .FALSE. is assumed. This argument is ignored on IA-32 processors.

exclusive (opt) Is an optional default logical constant. If .TRUE. is specified, you get exclusive ownership of the cache line because you intend to assign to it; if .FALSE. is specified, there is no exclusive ownership. If *exclusive* is omitted, .FALSE. is assumed. This argument is ignored on IA-32 processors.

Example

Consider the following:

```
subroutine spread_lf (a, b)
PARAMETER (n = 1025)

real*8 a(n,n), b(n,n), c(n)
do j = 1,n
  do i = 1,100
    a(i, j) = b(i-1, j) + b(i+1, j)
    call mm_prefetch a(i+20, j), 1)
    call mm_prefetch b(i+21, j), 1)
```

```
  enddo
enddo

print *, a(2, 567)

stop
end
```

MOD

Description: Returns the remainder when the first argument is divided by the second argument.

Syntax: result = MOD (*a*, *p*)

Class: Elemental function; Generic

Arguments:

a Must be of type integer or real.

p Must have the same type and kind parameters as *a*.

Results: The result type is the same as *a*. If *p* is not equal to zero, the value of the result is `a - INT(a/p) x p`. If *p* is equal to zero, the result is undefined.

Specific Name	Argument Type	Result Type
BMOD	INTEGER(1)	INTEGER(1)
IMOD[1]	INTEGER(2)	INTEGER(2)
MOD[2]	INTEGER(4)	INTEGER(4)
KMOD	INTEGER(8)	INTEGER(8)
AMOD[3]	REAL(4)	REAL(4)
DMOD	REAL(8)	REAL(8)
QMOD	REAL(16)	REAL(16)

1. Or HMOD.
2. Or JMOD.
3. The setting of compiler options specifying real size can affect AMOD.

Examples

MOD (7, 3) has the value 1.

MOD (9, –6) has the value 3.

MOD (–9, 6) has the value –3.

MODULO

Description: Returns the modulo of the arguments.

Syntax: result = MODULO (*a*, *p*)

Class: Elemental function; Generic

Arguments:

a Must be of type integer or real.

p Must have the same type and kind parameters as *a*.

Results: The result type is the same as *a*. The result value depends on the type of *a*, as follows:

- If *a* is of type integer and *p* is not equal to zero, the value of the result is `a - FLOOR(REAL(a)/REAL(p)) * p`.
- If *a* is of type real and P is not equal to zero, the value of the result is `a - FLOOR(a/p) * p`.

If *p* is equal to zero (regardless of the type of *a*), the result is undefined.

Examples

MODULO (7, 3) has the value 1.

MODULO (9, –6) has the value –3.

MODULO (–9, 6) has the value 3.

MULT_HIGH (i64 only)

Description: Multiplies two 64-bit unsigned integers. This specific function has no generic function associated with it and is only available on Intel Itanium processors. It cannot be passed as an actual argument.

Syntax: result = MULT_HIGH (*i*, *j*)

Class: Elemental function; Specific

Arguments:

i Must be of type INTEGER(8).

j Must be of type INTEGER(8).

Results: The result type is INTEGER(8). The result value is the upper (leftmost) 64 bits of the 128-bit unsigned result.

Example

Consider the following:

```
          INTEGER(8) I,J,K
          I=2_8**53
          J=2_8**51
          K = MULT_HIGH (I,J)
          PRINT *,I,J,K
          WRITE (6,1000)I,J,K
  1000    FORMAT (' ', 3(Z,1X))
          END
```

This example prints the following:

```
 9007199254740992      2251799813685248         1099511627776
   20000000000000           8000000000000             10000000000
```

MVBITS

Description: Copies a sequence of bits (a bit field) from one location to another.

Syntax: CALL MVBITS (*from*, *frompos*, *len*, *to*, *topos*)

Class: Elemental subroutine

Arguments: There are five arguments:[1]

from Can be of any integer type. It represents the location from which a bit field is transferred.

frompos Can be of any integer type; it must not be negative. It identifies the first bit position in the field transferred from *from*. *frompos* + *len* must be less than or equal to BIT_SIZE (*from*).[2]

len Can be of any integer type; it must not be negative. It identifies the length of the field transferred from *from*.

to Can be of any integer type, but must have the same kind parameter as *from*. It represents the location to which a bit field is transferred. *to* is set by copying the sequence of bits of length *len*, starting at position *frompos* of *from* to position *topos* of *to*. No other bits of *to* are altered.

On return, the *len* bits of *to* (starting at *topos*) are equal to the value that *len* bits of *from* (starting at *frompos*) had on entry.[2]

topos	Can be of any integer type; it must not be negative. It identifies the starting position (within *to*) for the bits being transferred. *topos* + *len* must be less than or equal to BIT_SIZE (*to*).

1. FROM, FROMPOS, LEN, and TOPOS are INTENT(IN) arguments; TO is an INTENT(INOUT) argument. For more information on INTENT, see "INTENT Attribute and Statement".
2. The model for the interpretation of an integer value as a sequence of bits is shown in "Model for Bit Data". For more information on bit functions, see "Bit Functions".

You can also use the following specific subroutines:

BMVBITS	Arguments *from* and *to* must be INTEGER(1).
HMVBITS	Arguments *from* and *to* must be INTEGER(2).
IMVBITS	All arguments must be INTEGER(2).
JMVBITS	Arguments can be INTEGER(2) or INTEGER(4); at least one must be INTEGER(4).
KMVBITS	Arguments can be INTEGER(2), INTEGER(4), or INTEGER(8); at least one must be INTEGER(8).

Example

If TO has the initial value of 6, its value after a call to MVBITS (7, 2, 2, TO, 0) is 5.

NARGS

Description: Returns the total number of command-line arguments, including the command. This function cannot be passed as an actual argument.

Syntax: result = NARGS ( )

Class: Inquiry function; Specific

Arguments: None.

Results: The result type is INTEGER(4). The result is the number of command-line arguments, including the command. For example, NARGS returns 4 for the command-line invocation of `PROG1 -g -c -a`.

Example

Consider the following:

```
INTEGER(2) result
result = RUNQQ('myprog', '-c -r')
END
```

```
!  MYPROG.F90 responds to command switches -r, -c,
!  and/or -d
INTEGER(4) count, num, i, status
CHARACTER(80) buf
REAL r1 / 0.0 /
COMPLEX c1 / (0.0,0.0) /
REAL(8) d1 / 0.0 /

num = 5
count = NARGS()
DO i = 1, count-1
   CALL GETARG(i, buf, status)
   IF (status .lt. 0) THEN
     WRITE (*,*) 'GETARG error - exiting'
     EXIT
   END IF
   IF (buf(2:status) .EQ.'r') THEN
     r1 = REAL(num)
     WRITE (*,*) 'r1 = ',r1
   ELSE IF (buf(2:status) .EQ.'c') THEN
     c1 = CMPLX(num)
     WRITE (*,*) 'c1 = ', c1
   ELSE IF (buf(2:status) .EQ.'d') THEN
     d1 = DBLE(num)
     WRITE (*,*) 'd1 = ', d1
   ELSE
     WRITE(*,*) 'Invalid command switch: ', buf (1:status)
   END IF
END DO
END
```

See Also

- "GETARG"
- "IARGC"

NEAREST

Description: Returns the nearest different number (representable on the processor) in a given direction.

Syntax: result = NEAREST (*x*, *s*)

Class: Elemental function; Generic

Arguments:

x Must be of type real.

s Must be of type real and nonzero.

Results: The result type is the same as *x*. The result has a value equal to the machine representable number that is different from and nearest to *x*, in the direction of the infinity with the same sign as *s*.

Example

If 3.0 and 2.0 are REAL(4) values, NEAREST (3.0, 2.0) has the value $3 + 2^{-22}$, which equals approximately 3.0000002. (For more information on the model for REAL(4), see "Model for Real Data".)

NINT

Description: Returns the nearest integer to the argument.

Syntax: result = NINT (*a* [, *kind*])

Class: Elemental function; Generic

Arguments:

a Must be of type real.

kind (opt) Must be a scalar integer initialization expression.

Results: The result type is integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is shown in the following table. If the processor cannot represent the result value in the kind of the result, the result is undefined.

If *a* is greater than zero, NINT (*a*) has the value INT (*a* + 0.5); if *a* is less than or equal to zero, NINT (*a*) has the value INT (*a* – 0.5).

Specific Name	Argument Type	Result Type
ININT	REAL(4)	INTEGER(2)

Specific Name	Argument Type	Result Type
NINT[1,2]	REAL(4)	INTEGER(4)
KNINT	REAL(4)	INTEGER(8)
IIDNNT	REAL(8)	INTEGER(2)
IDNINT[2,3]	REAL(8)	INTEGER(4)
KIDNNT	REAL(8)	INTEGER(8)
IIQNNT	REAL(16)	INTEGER(2)
IQNINT[2,4]	REAL(16)	INTEGER(4)
KIQNNT[5]	REAL(16)	INTEGER(8)

1. Or JNINT
2. The setting of compiler options specifying integer size can affect NINT, IDNINT, and IQNINT.
3. Or JIDNNT. For compatibility with older versions of Fortran, IDNINT can also be specified as a generic function.
4. Or JIQNNT. For compatibility with older versions of Fortran, IQNINT can also be specified as a generic function.
5. This specific function cannot be passed as an actual argument.

Examples

NINT (3.879) has the value 4.

NINT (–2.789) has the value –3.

NOT

Description: Returns the logical complement of the argument.

Syntax: result = NOT (*i*)

Class: Elemental function; Generic

Arguments: *i* must be of type integer.

Results: The result type is the same as I. The result value is obtained by complementing I bit-by-bit according to the following truth table:

I	NOT(I)
1	0
0	1

The model for the interpretation of an integer value as a sequence of bits is shown in "Model for Bit Data".

Specific Name	Argument Type	Result Type
BNOT	INTEGER(1)	INTEGER(1)

Specific Name	Argument Type	Result Type
INOT[1]	INTEGER(2)	INTEGER(2)
JNOT	INTEGER(4)	INTEGER(4)
KNOT	INTEGER(8)	INTEGER(8)

1. Or HNOT.

Example

If I has a value equal to 10101010 (base 2), NOT (I) has the value 01010101 (base 2).

NULL

Description: Initializes a pointer as disassociated when it is declared. This is a new intrinsic procedure in Fortran 95.

Syntax: result = NULL ([*mold*])

Class: Transformational function; Generic

Arguments: *mold* (opt)
Must be a pointer; it can be of any type. Its pointer association status can be associated, disassociated, or undefined. If its status is associated, the target does not have to be defined with a value.

Results: The result type is the same as *mold* (if present); otherwise, it is determined as follows:

If NULL( ) Appears:	Type is Determined From:
On the right side of pointer assignment	The pointer on the left side
As initialization for an object in a declaration	The object
As default initialization for a component	The component
In a structure constructor	The corresponding component
As an actual argument	The corresponding dummy argument
In a DATA statement	The corresponding pointer object

The result is a pointer with disassociated association status.

Example

Consider the following:

```
INTEGER, POINTER :: POINT1 => NULL()
```

This statement defines the initial association status of POINT1 to be disassociated.

PACK

Description: Takes elements from an array and packs them into a rank-one array under the control of a mask.

Syntax: result = PACK (*array*, *mask* [, *vector*])

Class: Transformational function; Generic

Arguments:

array Must be an array (of any data type).

mask Must be of type logical and conformable with *array*. It determines which elements are taken from *array*.

vector (opt) Must be a rank-one array with the same type and type parameters as *array*. Its size must be at least t, where *t* is the number of true elements in *mask*. If *mask* is a scalar with value true, *vector* must have at least as many elements as there are in *array*.

Elements in *vector* are used to fill out the result array if there are not enough elements selected by *mask*.

Results: The result is a rank-one array with the same type and type parameters as *array*. If *vector* is present, the size of the result is that of *vector*. Otherwise, the size of the result is the number of true elements in *mask*, or the number of elements in *array* (if *mask* is a scalar with value true).

Elements in *array* are processed in array element order to form the result array. Element i of the result is the element of *array* that corresponds to the ith true element of *mask*. If *vector* is present and has more elements than there are true values in *mask*, any result elements that are empty (because they were not true according to *mask*) are set to the corresponding values in *vector*.

Examples

N is the array $\begin{bmatrix} 0 & 8 & 0 \\ 0 & 0 & 0 \\ 7 & 0 & 0 \end{bmatrix}$.

PACK (N, MASK=N .NE. 0, VECTOR=(/1, 3, 5, 9, 11, 13/)) produces the result (7, 8, 5, 9, 11, 13).

PACK (N, MASK=N .NE. 0) produces the result (7, 8).

POPCNT

Description: Returns the number of 1 bits in the integer argument.

Syntax: result = POPCNT (*i*)

Class: Elemental function; Generic

Arguments: *i* must be of type integer or logical.

Results: The result type is the same as *i*. The result value is the number of 1 bits in the binary representation of the integer *i*.

The model for the interpretation of an integer value as a sequence of bits is shown in "Model for Bit Data".

Example

If the value of I is B'0...00011010110', the value of POPCNT(I) is 5.

POPPAR

Description: Returns the parity of the integer argument.

Syntax: result = POPPAR (*i*)

Class: Elemental function; Generic

Arguments: *i* must be of type integer or logical.

Results: The result type is the same as *i*. If there are an odd number of 1 bits in the binary representation of the integer *i*, the result value is 1. If there are an even number, the result value is zero.

POPPAR(*i*) is the same as 1 .AND. POPCNT(*i*).

The model for the interpretation of an integer value as a sequence of bits is shown in "Model for Bit Data".

Example

If the value of I is B'0...00011010110', the value of POPPAR(I) is 1.

PRECISION

Description: Returns the decimal precision in the model representing real numbers with the same kind parameter as the argument.

Syntax: result = PRECISION (*x*)

Class: Inquiry function; Generic

Arguments: *x* must be of type real or complex; it can be scalar or array valued.

Results: The result is a scalar of type default integer. The result has the value `INT((DIGITS(X) – 1) * LOG10(RADIX(X)))`. If `RADIX(X)` is an integral power of 10, 1 is added to the result.

Example

If X is a REAL(4) value, PRECISION (X) has the value 6. The value 6 is derived from INT ((24–1) * LOG10 (2.)) = INT (6.92...). For more information on the model for REAL(4), see "Model for Real Data".

PRESENT

Description: Returns whether or not an optional dummy argument is present (has an associated actual argument).

Syntax: result = PRESENT (*a*)

Class: Inquiry function; Generic

Arguments: *a* must be an optional argument of the current procedure.

Results: The result is a scalar of type default logical. The result is .TRUE. if *a* is present; otherwise, .FALSE..

Example

Consider the following:

```
SUBROUTINE CHECK (X, Y)
  REAL X, Z
  REAL, OPTIONAL :: Y
  ...
  IF (PRESENT (Y)) THEN
    Z = Y
  ELSE
    Z = X * 2
  END IF
END
 ...
CALL CHECK (15.0, 12.0)      ! Causes B to be set to 12.0
```

```
CALL CHECK (15.0)            ! Causes B to be set to 30.0
```

PRODUCT

Description: Returns the product of all the elements in an entire array or in a specified dimension of an array.

Syntax: result = PRODUCT (*array* [, *dim*] [, *mask*])

Class: Transformational function; Generic

Arguments:

array Must be an array of type integer or real.

dim (opt) Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of *array.*

mask (opt) Must be of type logical and conformable with *array.*

Results: The result is an array or a scalar of the same data type as *array.*

The result is a scalar if *dim* is omitted or *array* has rank one.

The following rules apply if *dim* is omitted:

- If PRODUCT (*array*) is specified, the result is the product of all elements of *array.* If *array* has size zero, the result is 1.
- If PRODUCT (*array*, MASK=*mask*) is specified, the result is the product of all elements of *array* corresponding to true elements of *mask*. If *array* has size zero, or every element of *mask* has the value .FALSE., the result is 1.

The following rules apply if *dim* is specified:

- If *array* has rank one, the value is the same as PRODUCT (*array* [,MASK=*mask*]).
- An array result has a rank that is one less than *array*, and shape (d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n), where (d_1, d_2, ..., d_n) is the shape of *array.*
- The value of element (s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n) of PRODUCT (*array*, *dim* [,*mask*]) is equal to PRODUCT (*array* (s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n) [,MASK=*mask* (s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n)]).

Examples

PRODUCT ((/2, 3, 4/)) returns the value 24 (the product of 2 * 3 * 4). PRODUCT ((/2, 3, 4/), DIM=1) returns the same result.

PRODUCT (C, MASK=C .LT. 0.0) returns the product of the negative elements of C.

A is the array $\begin{bmatrix} 1 & 4 & 7 \\ 2 & 3 & 5 \end{bmatrix}$.

PRODUCT (A, DIM=1) returns the value (2, 12, 35), which is the product of all elements in each column. 2 is the product of 1 * 2 in column 1. 12 is the product of 4 * 3 in column 2, and so forth.

PRODUCT (A, DIM=2) returns the value (28, 30), which is the product of all elements in each row. 28 is the product of 1 * 4 * 7 in row 1. 30 is the product of 2 * 3 * 5 in row 2.

QCMPLX

Description: Converts an argument to COMPLEX(16) type. This function cannot be passed as an actual argument.

Syntax: result = QCMPLX (*x* [, *y*])

Class: Elemental function; Specific

Arguments:

x Must be of type integer, real, or complex.

y (opt) Must be of type integer or real. It must not be present if *x* is of type complex.

Results: The result type is COMPLEX(16) (or COMPLEX*32).

If only one noncomplex argument appears, it is converted into the real part of the result value and zero is assigned to the imaginary part. If *y* is not specified and *x* is complex, the result value is CMPLX (REAL(X), AIMAG(X)).

If two noncomplex arguments appear, the complex value is produced by converting the first argument into the real part of the value, and converting the second argument into the imaginary part.

QCMPLX(X, Y) has the complex value whose real part is REAL(X, KIND=16) and whose imaginary part is REAL(Y, KIND=16).

Examples

QCMPLX (–3) has the value (–3.0Q0, 0.0Q0).

QCMPLX (4.1, 2.3) has the value (4.1Q0, 2.3Q0).

QEXT

Description: Converts a number to quad-precision real (REAL(16)) type.

Syntax: result = QEXT (*a*)

Class: Elemental function; Generic

Arguments: *a* must be of type integer, real, or complex.

Results: The result type is REAL(16) (REAL*16). Functions that cause conversion of one data type to another type have the same effect as the implied conversion in assignment statements.

If *a* is of type REAL(16), the result is the value of the *a* with no conversion (QEXT(*a*) = *a*).

If *a* is of type integer or real, the result has as much precision of the significant part of *a* as a REAL(16) value can contain.

If *a* is of type complex, the result has as much precision of the significant part of the real part of *a* as a REAL(16) value can contain.

Specific Name[1]	Argument Type	Result Type
	INTEGER(1)	REAL(16)
	INTEGER(2)	REAL(16)
	INTEGER(4)	REAL(16)
	INTEGER(8)	REAL(16)
QEXT	REAL(4)	REAL(16)
QEXTD	REAL(8)	REAL(16)
	REAL(16)	REAL(16)
	COMPLEX(4)	REAL(16)
	COMPLEX(8)	REAL(16)
	COMPLEX(16)	REAL(16)

1. These specific functions cannot be passed as actual arguments.

Examples

QEXT (4) has the value 4.0 (rounded; there are 32 places to the right of the decimal point).

QEXT ((3.4, 2.0)) has the value 3.4 (rounded; there are 32 places to the right of the decimal point).

QFLOAT

Description: Converts an integer to quad-precision real (REAL(16)) type.

Syntax: result = QFLOAT (*a*)

Class: Elemental function; Generic

Arguments: *a* must be of type integer.

Results: The result type is REAL(16) (REAL*16).

Functions that cause conversion of one data type to another type have the same affect as the implied conversion in assignment statements.

Example

QFLOAT (–4) has the value –4.0 (rounded; there are 32 places to the right of the decimal point).

QNUM

Description: Converts a character string to a REAL(16) value.

Syntax: result = QNUM (*i*)

Class: Elemental function; Specific

Arguments: *i* must be of type character.

Results: The result type is REAL(16). The result value is the real value represented by the character string *i*.

Example

QNUM ("–174.23") has the value –174.23 of type REAL(16).

QREAL

Description: Converts the real part of a COMPLEX(16) argument to REAL(16) type. This is a specific function that has no generic function associated with it. It cannot be passed as an actual argument.

Syntax: result = QREAL (*a*)

Class: Elemental function; Specific

Arguments: *a* must be of type COMPLEX(16) (or COMPLEX*32).

Results: The result type is quad-precision real (REAL(16) or REAL*16).

Example

QREAL ((2.0q0, 3.0q0)) has the value 2.0q0.

RADIX

Description: Returns the base of the model representing numbers of the same type and kind parameters as the argument.

Syntax: result = RADIX (*x*)

Class: Inquiry function; Generic

Arguments: *x* must be of type integer or real; it can be scalar or array valued.

Results: The result is a scalar of type default integer. For an integer argument, the result has the value r (as defined in "Model for Integer Data"). For a real argument, the result has the value b (as defined in "Model for Real Data").

Example

If X is a REAL(4) value, RADIX (X) has the value 2.

RAN

Description: Returns the next number from a sequence of pseudorandom numbers of uniform distribution over the range 0 to 1. This is a specific function that has no generic function associated with it. It cannot be passed as an actual argument.

RAN is not a pure function.

Syntax: result = RAN (*i*)

Class: Nonelemental function; Specific

Arguments: *i* is the *seed*. It must be an INTEGER(4) variable or array element.

It should initially be set to a large, odd integer value. The RAN function stores a value in the argument that is later used to calculate the next random number.

There are no restrictions on the seed, although it should be initialized with different values on separate runs to obtain different random numbers.

Results: The result type is REAL(4). The result is a floating-point number that is uniformly distributed in the range between 0.0 inclusive and 1.0 exclusive. It is set equal to the value associated with the argument *i*.

Example

In RAN (I), if variable I has the value 3, RAN has the value 4.8220158E–05.

RANDOM_NUMBER

Description: Returns one pseudorandom number or an array of such numbers.

Syntax: CALL RANDOM_NUMBER (*harvest*)

Class: Subroutine

Arguments: *harvest* must be of type real. It is an INTENT(OUT) argument (see "INTENT Attribute and Statement"), and can be scalar or an array variable.

It is set to contain pseudorandom numbers from the uniform distribution within the range $0 \leq x < 1$.

The seed for the pseudorandom number generator used by RANDOM_NUMBER can be set or queried with "RANDOM_SEED". If RANDOM_SEED is not used, the processor sets the seed for RANDOM_NUMBER to a processor-dependent value.

The RANDOM_NUMBER generator uses two separate congruential generators together to produce a period of approximately 10**18, and produces real pseudorandom results with a uniform distribution in (0,1). It accepts two integer seeds, the first of which is reduced to the range [1, 2147483562]. The second seed is reduced to the range [1, 2147483398]. This means that the generator effectively uses two 31-bit seeds.

For more information on the algorithm, see the following:

- Communications of the ACM vol 31 num 6 June 1988, titled: Efficient and Portable Combined Random Number Generators by Pierre L'ecuyer.
- Springer-Verlag New York, N. Y. 2nd ed. 1987, titled: A Guide to Simulation by Bratley, P., Fox, B. L., and Schrage, L. E.

Example

Consider the following:

```
REAL Y, Z (5, 5)
! Initialize Y with a pseudorandom number
CALL RANDOM_NUMBER (HARVEST = Y)
CALL RANDOM_NUMBER (Z)
```

Y and Z contain uniformly distributed random numbers.

The following shows another example:

```
REAL x, array1 (5, 5)
CALL RANDOM_SEED()
CALL RANDOM_NUMBER(x)
CALL RANDOM_NUMBER(array1)
```

Consider also the following:

```
program testrand
  intrinsic random_seed, random_number
  integer size, seed(2), gseed(2), hiseed(2), zseed(2)
  real harvest(10)
  data seed /123456789, 987654321/
  data hiseed /-1, -1/
  data zseed /0, 0/
  call random_seed(SIZE=size)
  print *,"size ",size
  call random_seed(PUT=hiseed(1:size))
  call random_seed(GET=gseed(1:size))
  print *,"hiseed gseed", hiseed, gseed
  call random_seed(PUT=zseed(1:size))
  call random_seed(GET=gseed(1:size))
  print *,"zseed gseed ", zseed, gseed
  call random_seed(PUT=seed(1:size))
  call random_seed(GET=gseed(1:size))
  call random_number(HARVEST=harvest)
  print *, "seed gseed ", seed, gseed
  print *, "harvest"
  print *, harvest
  call random_seed(GET=gseed(1:size))
  print *,"gseed after harvest ", gseed
end program testrand
```

RANDOM_SEED

Description: Changes or queries the seed (starting point) for the pseudorandom number generator used by RANDOM_NUMBER.

Syntax: CALL RANDOM_SEED ([*size*] [, *put*] [, *get*])

Class: Subroutine

Arguments: No more than one argument can be specified. If no argument is specified, a random number based on the date and time is assigned to the seed. The three optional arguments follow:[1]

size (opt)	Must be scalar and of type default integer. It is set to the number of integers (N) that the processor uses to hold the value of the seed.
put (opt)	Must be a default integer array of rank one and size ≥ N. It is used to reset the seed value.
get (opt)	Must be a default integer array of rank one and size ≥ N. It is set to the current value of the seed.

1. SIZE and GET are INTENT(OUT) arguments; PUT is an INTENT(IN) argument. For more information on INTENT, see "INTENT Attribute and Statement".

You can determine the size of the array the processor uses to store the seed by calling RANDOM_SEED with the *size* argument (see the second example below).

The setting of compiler options specifying integer size can affect this subroutine.

Example

Consider the following:

```
CALL RANDOM_SEED                      ! Processor initializes the seed
                                    !   randomly from the date and time
CALL RANDOM_SEED (SIZE = M)             ! Sets M to N
CALL RANDOM_SEED (PUT = SEED (1 : M))  ! Sets user seed
CALL RANDOM_SEED (GET = OLD  (1 : M))  ! Reads current seed
```

The following shows another example:

```
INTEGER I
INTEGER, ALLOCATABLE :: new (:), old(:)
CALL RANDOM_SEED ( )    ! Processor reinitializes the seed
                        !   randomly from the date and time
CALL RANDOM_SEED (SIZE = I)    ! I is set to the size of
                               !   the seed array
ALLOCATE (new(I))
ALLOCATE (old(I))
CALL RANDOM_SEED (GET=old(1:I))   ! Gets the current seed
WRITE(*,*) old
new = 5
CALL RANDOM_SEED (PUT=new(1:I))   ! Sets seed from array
                                  !   new
END
```

RANDU

Description: Computes a pseudorandom number as a single-precision value.

Syntax: CALL RANDU (*i1*, *i2*, *x*)

Class: Subroutine

Arguments:

i1, *i2* INTEGER(2) variables or array elements that contain the *seed* for computing the random number. These values are updated during the computation so that they contain the updated seed.

x A REAL(4) variable or array element where the computed random number is returned.

The result is returned in *x*, which must be of type REAL(4). The result value is a pseudorandom number in the range 0.0 to 1.0. The algorithm for computing the random number value is based on the values for *i1* and *i2*.

If *i1*=0 and *i2*=0, the generator base is set as follows:

```
X(n + 1) = 2**16 + 3
```

Otherwise, it is set as follows:

```
X(n + 1) = (2**16 + 3) * X(n) mod 2**32
```

The generator base `X(n + 1)` is stored in *i1*, *i2*. The result is `X(n + 1)` scaled to a real value `Y(n + 1)`, for `0.0` ≤ `Y(n + 1)` < `1`.

Example

Consider the following:

```
REAL X
INTEGER(2) I, J
...
CALL RANDU (I, J, X)
```

If I and J are values 4 and 6, X stores the value 5.4932479E–04.

RANGE

Description: Returns the decimal exponent range in the model representing numbers with the same kind parameter as the argument.

Syntax: result = RANGE (*x*)

Class: Inquiry function; Generic

Arguments: *x* must be of type integer, real, or complex; it can be scalar or array valued.

Results: The result is a scalar of type default integer.

For an integer argument, the result has the value INT (LOG10 (HUGE(X))). For information on the integer model, see "Model for Integer Data"; on HUGE, see "HUGE".

For a real or complex argument, the result has the value INT(MIN (LOG10(HUGE(X)), –LOG10(TINY(X)))). For information on the real model, see "Model for Real Data"; on TINY, see "TINY".

Example

If X is a REAL(4) value, RANGE (X) has the value 37. (HUGE(X) = $(1 - 2^{-24}) \times 2^{128}$ and TINY(X) = 2^{-126}.)

REAL

Description: Converts a value to real type.

Syntax: result = REAL (*a* [, *kind*])

Class: Elemental function; Generic

Arguments:

a Must be of type integer, real, or complex.

kind (opt) Must be a scalar integer initialization expression.

Results: The result type is real. If *kind* is present, the kind parameter is that specified by *kind*; otherwise, the kind parameter of the result is shown in the following table. If the processor cannot represent the result value in the kind of the result, the result is undefined.

Functions that cause conversion of one data type to another type have the same affect as the implied conversion in assignment statements.

If *a* is integer or real, the result is equal to an approximation of *a*. If *a* is complex, the result is equal to an approximation of the real part of *a*.

Specific Name[1]	Argument Type	Result Type
	INTEGER(1)	REAL(4)
FLOATI	INTEGER(2)	REAL(4)
FLOAT[2,3]	INTEGER(4)	REAL(4)

Specific Name[1]	Argument Type	Result Type
REAL[2]	INTEGER(4)	REAL(4)
FLOATK	INTEGER(8)	REAL(4)
	REAL(4)	REAL(4)
SNGL[2,4]	REAL(8)	REAL(4)
SNGLQ	REAL(16)	REAL(4)
	COMPLEX(4)	REAL(4)
	COMPLEX(8)	REAL(8)

1. These specific functions cannot be passed as actual arguments.
2. The setting of compiler options specifying real size can affect FLOAT, REAL, and SNGL.
3. Or FLOATJ. For compatibility with older versions of Fortran, FLOAT can also be specified as a generic function.
4. For compatibility with older versions of Fortran, SNGL can also be specified as a generic function. The generic SNGL includes specific function REAL, which takes a REAL(4) argument and produces a REAL(4) result.

Examples

REAL (–4) has the value –4.0.

REAL (Y) has the same kind parameter and value as the real part of complex variable Y.

REPEAT

Description: Concatenates several copies of a string.

Syntax: result = REPEAT (*string*, *ncopies*)

Class: Transformational function; Generic

Arguments:

string Must be scalar and of type character.

ncopies Must be scalar and of type integer. It must not be negative.

Results: The result is a scalar of type character and length *ncopies* x LEN(*string*). The kind parameter is the same as *string*. The value of the result is the concatenation of *ncopies* copies of *string*.

Examples

REPEAT ('S', 3) has the value SSS.

REPEAT ('ABC', 0) has the value of a zero-length string.

RESHAPE

Description: Constructs an array with a different shape from the argument array.

Syntax: result = RESHAPE (*source*, *shape* [, *pad*] [, *order*])

Class: Transformational function; Generic

Arguments:

source Must be an array (of any data type). It supplies the elements for the result array. Its size must be greater than or equal to PRODUCT(*shape*) if *pad* is omitted or has size zero.

shape Must be an integer array of up to 7 elements, with rank one and constant size. It defines the shape of the result array. Its size must be positive; its elements must not have negative values.

pad (opt) Must be an array with the same type and kind parameters as *source*. It is used to fill in extra values if the result array is larger than *source*.

order (opt) Must be an integer array with the same shape as *shape*. Its elements must be a permutation of (1,2,...,n), where *n* is the size of *shape*. If *order* is omitted, it is assumed to be (1,2,...,n).

Results: The result is an array of shape *shape* with the same type and kind parameters as *source*. The size of the result is the product of the values of the elements of *shape*.

In the result array, the array elements of *source* are placed in the order of dimensions specified by *order*. If *order* is omitted, the array elements are placed in normal array element order.

The array elements of *source* are followed (if necessary) by the array elements of *pad* in array element order. If necessary, additional copies of *pad* follow until all the elements of the result array have values.

Examples

RESHAPE ((/3, 4, 5, 6, 7, 8/), (/2, 3/)) has the value $\begin{bmatrix} 3 & 5 & 7 \\ 4 & 6 & 8 \end{bmatrix}$.

RESHAPE ((/3, 4, 5, 6, 7, 8/), (/2, 4/), (/1, 1/), (/2, 1/)) has the value $\begin{bmatrix} 3 & 4 & 5 & 6 \\ 7 & 8 & 1 & 1 \end{bmatrix}$.

RNUM

Description: Converts a character string to a REAL(4) value.

Syntax: result = RNUM (*i*)

Class: Elemental function; Specific

Arguments: *i* must be of type character.

Results: The result type is REAL(4). The result value is the real value represented by the character string *i*.

Example

RNUM ("821.003") has the value 821.003 of type REAL(4)

RRSPACING

Description: Returns the reciprocal of the relative spacing of model numbers near the argument value.

Syntax: result = RRSPACING (*x*)

Class: Elemental function; Generic

Arguments: *x* must be of type real.

Results: The result type is the same as *x*. The result has the value $| x * b^{-e} | \times b^{p}$. Parameters *b*, *e*, and *p* are defined in "Model for Real Data".

Example

If –3.0 is a REAL(4) value, RRSPACING (–3.0) has the value 0.75×2^{24}.

SCALE

Description: Returns the value of the exponent part (of the model for the argument) changed by a specified value.

Syntax: result = SCALE (*x*, *i*)

Class: Elemental function; Generic

Arguments:

x Must be of type real.

i Must be of type integer.

Results: The result type is the same as *x*. The result has the value $x \times b^i$. Parameter *b* is defined in "Model for Real Data".

Example

If 3.0 is a REAL(4) value, SCALE (3.0, 2) has the value 12.0 and SCALE (3.0, 3) has the value 24.0.

SCAN

Description: Scans a string for any character in a set of characters.

Syntax: result = SCAN (*string*, *set* [, *back*] [, *kind*])

Class: Elemental function; Generic

Arguments:

string Must be of type character.

set Must be of type character with the same kind parameter as *string*.

back (opt) Must be of type logical.

kind (opt) Must be a scalar integer initialization expression.

Results: The result is of type integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

If *back* is omitted (or is present with the value false) and *string* has at least one character that is in *set*, the value of the result is the position of the leftmost character of *string* that is in *set*.

If *back* is present with the value true and *string* has at least one character that is in *set*, the value of the result is the position of the rightmost character of *string* that is in *set*.

If no character of *string* is in *set* or the length of *string* or *set* is zero, the value of the result is zero.

The setting of compiler options specifying integer size can affect this function.

Examples

SCAN ('ASTRING', 'ST') has the value 2.

SCAN ('ASTRING', 'ST', BACK=.TRUE.) has the value 3.

SCAN ('ASTRING', 'CD') has the value zero.

SECNDS

Description: Provides the system time of day, or elapsed time, as a floating-point value in seconds. This is a specific function that has no generic function associated with it. It cannot be passed as an actual argument.

SECNDS is not a pure function, so it cannot be referenced inside a FORALL construct.

Syntax: result = SECNDS (*x*)

Class: Elemental function; Specific

Arguments: *x* must be of type REAL(4).

Results: The result type is the same as *x*. The result value is the time in seconds since midnight – *x*. (The function also produces correct results for time intervals that span midnight.)

The value of SECNDS is accurate to 0.01 second, which is the resolution of the system clock.

The 24 bits of precision provide accuracy to the resolution of the system clock for about one day. However, loss of significance can occur if you attempt to compute very small elapsed times late in the day.

Example

The following shows how to use SECNDS to perform elapsed-time computations:

```
C     START OF TIMED SEQUENCE
      T1 = SECNDS(0.0)
C     CODE TO BE TIMED
      ...
      DELTA = SECNDS(T1)      ! DELTA gives the elapsed time
```

SELECTED_INT_KIND

Description: Returns the value of the kind parameter of an integer data type.

Syntax: result = SELECTED_INT_KIND (*p*)

Class: Transformational function; Generic

Arguments: *p* must be scalar and of type integer.

Results: The result is a scalar of type default integer. The result has a value equal to the value of the kind parameter of the integer data type that represents all values *n* in the range of values *n* with $-10^p < n < 10^p$.

If no such kind type parameter is available on the processor, the result is –1. If more than one kind type parameter meets the criteria, the value returned is the one with the smallest decimal exponent range. (For information on the integer model, see "Model for Integer Data".)

Example

SELECTED_INT_KIND (6) = 4

SELECTED_REAL_KIND

Description: Returns the value of the kind parameter of a real data type.

Syntax: result = SELECTED_REAL_KIND ([*p*] [, *r*])

Class: Transformational function; Generic

Arguments:

p (opt) Must be scalar and of type integer.

r (opt) Must be scalar and of type integer.

Results: The result is a scalar of type default integer. If both arguments are absent, the result is zero. Otherwise, the result has a value equal to a value of the kind parameter of a real data type with decimal precision, as returned by the function PRECISION, of at least *p* digits and a decimal exponent range, as returned by the function RANGE, of at least *r*.

If no such kind type parameter is available on the processor, the result is as follows:

```
-1 if the precision is not available
-2 if the exponent range is not available
-3 if neither is available
```

If more than one kind type parameter value meets the criteria, the value returned is the one with the smallest decimal precision. (For information on the real model, see "Model for Real Data".)

Example

SELECTED_REAL_KIND (6, 70) = 8

SET_EXPONENT

Description: Returns the value of the exponent part (of the model for the argument) set to a specified value.

Syntax: result = SET_EXPONENT (*x*, *i*)

Class: Elemental function; Generic

Arguments:

x Must be of type real.

i Must be of type integer.

Results: The result type is the same as *x*. The result has the value $x \times b^{i-e}$. Parameters *b* and *e* are defined in "Model for Real Data". If *x* has the value zero, the result is zero.

Example

If 3.0 is a REAL(4) value, SET_EXPONENT (3.0, 1) has the value 1.5.

SHAPE

Description: Returns the shape of an array or scalar argument.

Syntax: result = SHAPE (*source* [, *kind*])

Class: Inquiry function; Generic

Arguments:

source Is a scalar or array (of any data type). It must not be an assumed-size array, a disassociated pointer, or an allocatable array that is not allocated.

kind (opt) Must be a scalar integer initialization expression.

Results: The result is a rank-one integer array whose size is equal to the rank of *source*. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The value of the result is the shape of *source*.

The setting of compiler options specifying integer size can affect this function.

Example

SHAPE (2) has the value of a rank-one array of size zero.

If B is declared as B(2:4, –3:1), then SHAPE (B) has the value (3, 5).

SHIFTL

Description: Arithmetically shifts an integer left by a specified number of bits.

Syntax: result = SHIFTL (*ivalue*, *ishift*)

Class: Elemental function; Specific

Arguments:

ivalue Must be of type INTEGER(4). This is the value to be shifted.

ishift Must be of type INTEGER(4) and positive. This value is the number of positions to shift.

Results: The result type is INTEGER(4). The result is the value of *ivalue* shifted left by *ishift* bit positions. Bits shifted off the left end are lost; zeros are shifted in from the opposite end.

SHIFTL (i, j) is the same as ISHA (i, j).

SHIFTR

Description: Arithmetically shifts an integer right by a specified number of bits.

Syntax: result = SHIFTR (*ivalue*, *ishift*)

Class: Elemental function; Specific

Arguments:

ivalue Must be of type INTEGER(4). This is the value to be shifted.

ishift Must be of type INTEGER(4) and positive. This value is the number of positions to shift.

Results: The result type is INTEGER(4). The result is the value of *ivalue* shifted right by *ishift* bit positions. Bits shifted off the right end are lost; zeros are shifted in from the opposite end.

SHIFTR (i, j) is the same as ISHA (i, –j).

SIGN

Description: Returns the absolute value of *a* times the sign of *b*.

Syntax: result = SIGN (*a*, *b*)

Class: Elemental function; Generic

Arguments:

a Must be of type integer or real.

b Must have the same type and kind parameters as *a*.

Results: The result type is the same as *a*. The value of the result is | *a* | if *b* ≥ zero and –| *a* | if *b* < zero.

If *b* is of type real and zero, the value of the result is | *a* |. However, if the processor can distinguish between positive and negative real zero and the appropriate compiler option is specified, the following occurs:

- If *b* is positive real zero, the value of the result is | *a* |.
- If *b* is negative real zero, the value of the result is –| *a* |.

Specific Name	Argument Type	Result Type
BSIGN	INTEGER(1)	INTEGER(1)
IISIGN[1]	INTEGER(2)	INTEGER(2)
ISIGN[2]	INTEGER(4)	INTEGER(4)
KISIGN	INTEGER(8)	INTEGER(8)
SIGN	REAL(4)	REAL(4)
DSIGN	REAL(8)	REAL(8)
QSIGN	REAL(16)	REAL(16)

1. Or HSIGN.
2. Or JISIGN. For compatibility with older versions of Fortran, ISIGN can also be specified as a generic function.

Examples

SIGN (4.0, –6.0) has the value –4.0.

SIGN (–5.0, 2.0) has the value 5.0.

SIN

Description: Produces the sine of *x*.

Syntax: result = SIN (*x*)

Class: Elemental function; Generic

Arguments: *x* must be of type real or complex. It must be in radians and is treated as modulo 2*π.

If *x* is of type complex, its real part is regarded as a value in radians.

Results: The result type is the same as *x*.

Specific Name	Argument Type	Result Type
SIN	REAL(4)	REAL(4)
DSIN	REAL(8)	REAL(8)
QSIN	REAL(16)	REAL(16)
CSIN[1]	COMPLEX(4)	COMPLEX(4)
CDSIN[2]	COMPLEX(8)	COMPLEX(8)
CQSIN	COMPLEX(16)	COMPLEX(16)

1. The setting of compiler options specifying real size can affect CSIN.
2. This function can also be specified as ZSIN.

Examples

SIN (2.0) has the value 0.9092974.

SIN (0.8) has the value 0.7173561.

SIND

Description: Produces the sine of *x*.

Syntax: result = SIND (*x*)

Class: Elemental function; Generic

Arguments: *x* must be of type real. It must be in degrees and is treated as modulo 360.

Results: The result type is the same as *x*.

Specific Name	Argument Type	Result Type
SIND	REAL(4)	REAL(4)

Specific Name	Argument Type	Result Type
DSIND	REAL(8)	REAL(8)
QSIND	REAL(16)	REAL(16)

Examples

SIND (2.0) has the value 3.4899496E–02.

SIND (0.8) has the value 1.3962180E–02.

SINH

Description: Produces a hyperbolic sine.

Syntax: result = SINH (*x*)

Class: Elemental function; Generic

Arguments: *x* must be of type real.

Results: The result type is the same as *x*.

Specific Name	Argument Type	Result Type
SINH	REAL(4)	REAL(4)
DSINH	REAL(8)	REAL(8)
QSINH	REAL(16)	REAL(16)

Examples

SINH (2.0) has the value 3.626860.

SINH (0.8) has the value 0.8881060.

SIZE

Description: Returns the total number of elements in an array, or the extent of an array along a specified dimension.

Syntax: result = SIZE (*array* [, *dim*] [, *kind*])

Class: Inquiry function; Generic

Arguments:

array	Must be an array (of any data type). It must not be a disassociated pointer or an allocatable array that is not allocated. It can be an assumed-size array if *dim* is present with a value less than the rank of *array.*
dim (opt)	Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of *array.*
kind (opt)	Must be a scalar integer initialization expression.
Results:	The result is a scalar of type integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined. If *dim* is present, the result is the extent of dimension *dim* in *array*; otherwise, the result is the total number of elements in *array.* The setting of compiler options specifying integer size can affect this function.

Example

If B is declared as B(2:4, –3:1), then SIZE (B, DIM=2) has the value 5 and SIZE (B) has the value 15.

SIZEOF

Description:	Returns the number of bytes of storage used by the argument. It cannot be passed as an actual argument.
Syntax:	result = SIZEOF (*x*)
Class:	Inquiry function; Generic
Arguments:	*x* is a scalar or array (of any data type). It must *not* be an assumed-size array.
Results:	The result type is INTEGER(4) on IA-32 processors; INTEGER(8) on Intel Itanium processors. The result value is the number of bytes of storage used by *x*.

Examples

SIZEOF (3.44) has the value 4.

SIZEOF ('SIZE') has the value 4.

SPACING

Description: Returns the absolute spacing of model numbers near the argument value.

Syntax: result = SPACING (*x*)

Class: Elemental function; Generic

Arguments: *x* must be of type real.

Results: The result type is the same as *x*. The result has the value b^{e-p}. Parameters *b*, *e*, and *p* are defined in "Model for Real Data".

If the result value is outside of the real model range, the result is TINY(X). (For information on TINY, see "TINY".)

Example

If 3.0 is a REAL(4) value, SPACING (3.0) has the value 2^{-22}.

SPREAD

Description: Creates a replicated array with an added dimension by making copies of existing elements along a specified dimension.

Syntax: result = SPREAD (*source*, *dim*, *ncopies*)

Class: Transformational function; Generic

Arguments:

source Must be a scalar or array (of any data type). The rank must be less than 7.

dim Must be scalar and of type integer. It must have a value in the range 1 to n + 1 (inclusive), where *n* is the rank of *source*.

ncopies Must be scalar and of type integer. It becomes the extent of the additional dimension in the result.

Results: The result is an array of the same type as *source* and of rank that is one greater than *source*.

If *source* is an array, each array element in dimension *dim* of the result is equal to the corresponding array element in *source*.

If *source* is a scalar, the result is a rank-one array with *ncopies* elements, each with the value *source*.

If *ncopies* ≤ zero, the result is an array of size zero.

Examples

SPREAD ("B", 1, 4) is the character array (/"B", "B", "B", "B"/).

B is the array (3, 4, 5) and NC has the value 4.

SPREAD (B, DIM=1, NCOPIES=NC) produces the array $\begin{bmatrix} 3 & 4 & 5 \\ 3 & 4 & 5 \\ 3 & 4 & 5 \\ 3 & 4 & 5 \end{bmatrix}$.

SPREAD (B, DIM=2, NCOPIES=NC) produces the array $\begin{bmatrix} 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 \\ 5 & 5 & 5 & 5 \end{bmatrix}$.

SQRT

Description: Produces the square root of the argument.

Syntax: result = SQRT (*x*)

Class: Elemental function; Generic

Arguments: *x* must be of type real or complex. If *x* is type real, its value must be greater than or equal to zero.

Results: The result type is the same as *x*. The result has a value equal to the square root of *x*. A result of type complex is the principal value, with the real part greater than or equal to zero. When the real part of the result is zero, the imaginary part is greater than or equal to zero.

Specific Name	Argument Type	Result Type
SQRT	REAL(4)	REAL(4)
DSQRT	REAL(8)	REAL(8)
QSQRT	REAL(16)	REAL(16)
CSQRT[1]	COMPLEX(4)	COMPLEX(4)
CDSQRT[2]	COMPLEX(8)	COMPLEX(8)
CQSQRT	COMPLEX(16)	COMPLEX(16)

1. The setting of compiler options specifying real size can affect CSQRT.
2. This function can also be specified as ZSQRT.

Examples

SQRT (16.0) has the value 4.0.

SQRT (3.0) has the value 1.732051.

SUM

Description: Returns the sum of all the elements in an entire array or in a specified dimension of an array.

Syntax: result = SUM (*array* [, *dim*] [, *mask*])

Class: Transformational function; Generic

Arguments:

array Must be an array of type integer, real, or complex.

dim (opt) Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of *array.*

mask (opt) Must be of type logical and conformable with *array.*

Results: The result is an array or a scalar of the same type as *array.*

The result is a scalar if *dim* is omitted or *array* has rank one.

The following rules apply if *dim* is omitted:

- If SUM (*array*) is specified, the result is the sum of all elements of *array.* If *array* has size zero, the result is zero.
- If SUM (*array*, MASK=*mask*) is specified, the result is the sum of all elements of *array* corresponding to true elements of *mask*. If *array* has size zero, or every element of *mask* has the value .FALSE., the result is zero.

The following rules apply if *dim* is specified:

- If *array* has rank one, the value is the same as SUM (*array* [,MASK=*mask*]).
- An array result has a rank that is one less than *array*, and shape (d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n), where (d_1, d_2, ..., d_n) is the shape of *array.*
- The value of element (s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n) of SUM (*array*, *dim* [,*mask*]) is equal to SUM (*array* (s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1} ..., s_n) [,MASK = *mask* (s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n)].

Examples

SUM ((/2, 3, 4/)) returns the value 9 (sum of 2 + 3 + 4). SUM ((/2, 3, 4/), DIM=1) returns the same result.

SUM (B, MASK=B .LT. 0.0) returns the arithmetic sum of the negative elements of B.

C is the array $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$.

SUM (C, DIM=1) returns the value (5, 7, 9), which is the sum of all elements in each column. 5 is the sum of 1 + 4 in column 1. 7 is the sum of 2 + 5 in column 2, and so forth. SUM (C, DIM=2) returns the value (6, 15), which is the sum of all elements in each row. 6 is the sum of 1 + 2 + 3 in row 1. 15 is the sum of 4 + 5 + 6 in row 2.

SYSTEM_CLOCK

Description: Returns integer data from a real-time clock.

SYSTEM_CLOCK returns the number of seconds from 00:00 Coordinated Universal Time (CUT) on 1 JAN 1970. The number is returned with no bias. To get the elapsed time, you must call SYSTEM_CLOCK twice, and subtract the starting time value from the ending time value.

Syntax: CALL SYSTEM_CLOCK ([*count*] [, *count_rate*] [, *count_max*])

Class: Subroutine

Arguments: There are three optional arguments:[1]

count (opt) Must be scalar and of type integer. It is set to a value based on the current value of the processor clock. The value is increased by one for each clock count until the value *count_max* is reached, and is reset to zero at the next count. (*count* lies in the range 0 to *count_max*.)

count_rate (opt) Must be scalar and of type integer. It is set to the number of processor clock counts per second.

If the type is INTEGER(2), *count_rate* is 1000. If the type is INTEGER(4), *count_rate* is 10000. If the type is INTEGER(8), *count_rate* is 1000000.

count_max (opt) Must be scalar and of type integer. It is set to the maximum value that *count* can have, HUGE(0). For more information on HUGE, see "HUGE".

1. All are INTENT(OUT) arguments. (See "INTENT Attribute and Statement".)

All arguments used must have the same integer kind parameter.

Example

Consider the following:

```
integer(2) :: ic2, crate2, cmax2
integer(4) :: ic4, crate4, cmax4
```

```
call system_clock(count=ic2, count_rate=crate2, count_max=cmax2)
call system_clock(count=ic4, count_rate=crate4, count_max=cmax4)
print *, ic2, crate2, cmax2
print *, ic4, crate4, cmax4
end
```

This program was run on Thursday Dec 11, 1997 at 14:23:55 EST and produced the following output:

```
13880   1000  32767
1129498807       10000  2147483647
```

TAN

Description: Produces the tangent of *x*.

Syntax: result = TAN (*x*)

Class: Elemental function; Generic

Arguments: *x* must be of type real or complex. It must be in radians and is treated as modulo 2 * π.

If *x* is of type complex, its real part is regarded as a value in radians.

Results: The result type is the same as *x*.

Specific Name	Argument Type	Result Type
TAN	REAL(4)	REAL(4)
DTAN	REAL(8)	REAL(8)
QTAN	REAL(16)	REAL(16)
CTAN[1]	COMPLEX(4)	COMPLEX(4)
CDTAN[2]	COMPLEX(8)	COMPLEX(8)
CQTAN	COMPLEX(16)	COMPLEX(16)

1. The setting of compiler options specifying real size can affect CTAN.
2. This function can also be specified as ZTAN.

Examples

TAN (2.0) has the value –2.185040.

TAN (0.8) has the value 1.029639.

TAND

Description: Produces the tangent of *x*.

Syntax: result = TAND (*x*)

Class: Elemental function; Generic

Arguments: *x* must be of type real. It must be in degrees and is treated as modulo 360.

Results: The result type is the same as *x*.

Specific Name	Argument Type	Result Type
TAND	REAL(4)	REAL(4)
DTAND	REAL(8)	REAL(8)
QTAND	REAL(16)	REAL(16)

Examples

TAND (2.0) has the value 3.4920771E–02.

TAND (0.8) has the value 1.3963542E–02.

TANH

Description: Produces a hyperbolic tangent.

Syntax: result = TANH (*x*)

Class: Elemental function; Generic

Arguments: *x m*ust be of type real.

Results: The result type is the same as *x*.

Specific Name	Argument Type	Result Type
TANH	REAL(4)	REAL(4)
DTANH	REAL(8)	REAL(8)
QTANH	REAL(16)	REAL(16)

Examples

TANH (2.0) has the value 0.9640276.

TANH (0.8) has the value 0.6640368.

TIME

Description: Returns the current time as set within the system.

Syntax: CALL TIME (*buf*)

Class: Subroutine

Arguments: *buf* is an 8-byte variable, array, array element, or character substring.

The date is returned as an 8-byte ASCII character string taking the form hh:mm:ss, where:

hh	is the 2-digit hour
mm	is the 2-digit minute
ss	is the 2-digit second

If *buf* is of numeric type and smaller than 8 bytes, data corruption can occur.

If *buf* is of character type, its associated length is passed to the subroutine. If *buf* is smaller than 8 bytes, the subroutine truncates the date to fit in the specified length. If a CHARACTER array is passed, the subroutine stores the date in the first array element, using the element length, not the length of the entire array.

Examples

An example of a value returned from a call to TIME is 13:45:23 (a 24-hour clock is used).

Consider the following:

```
CHARACTER*1 HOUR(8)
...
CALL TIME (HOUR)
```

The length of the first array element in CHARACTER array HOUR is passed to the TIME subroutine. The subroutine then truncates the time to fit into the 1-character element, producing an incorrect result.

TINY

Description: Returns the smallest number in the model representing the same type and kind parameters as the argument.

Syntax: result = TINY (*x*)

Class: Inquiry function; Generic

Arguments: *x* must be of type real; it can be scalar or array valued.

Results: The result is a scalar with the same type and kind parameters as *x*. The result has the value b^{emin-1}. Parameters *b* and e_{min} are defined in "Model for Real Data".

Example

If X is of type REAL(4), TINY (X) has the value 2^{-126}.

TRAILZ

Description: Returns the number of trailing zero bits in an integer.

Syntax: result = TRAILZ (*i*)

Class: Elemental function; Generic

Arguments: *i* must be of type integer or logical.

Results: The result type is the same as *i*. The result value is the number of trailing zeros in the binary representation of the integer *i*.

The model for the interpretation of an integer value as a sequence of bits is shown in "Model for Bit Data".

Example

Consider the following:

```
INTEGER*8 J, TWO
PARAMETER (TWO=2)
DO J= -1, 40
 TYPE *, TRAILZ(TWO**J)  ! Prints 64, then 0 up to
ENDDO                    !  40 (trailing zeros)
END
```

TRANSFER

Description: Converts the bit pattern of *source* according to the type and kind parameters of *mold*.

Syntax: result = TRANSFER (*source*, *mold* [, *size*])

Class: Transformational function; Generic

Arguments:

source Must be a scalar or array (of any data type).

mold Must be a scalar or array (of any data type). It provides the type characteristics (not a value) for the result.

size (opt) Must be scalar and of type integer. It provides the number of elements for the output result.

Results: The result has the same type and type parameters as *mold*.

If *mold* is a scalar and *size* is omitted, the result is a scalar.

If *mold* is an array and *size* is omitted, the result is a rank-one array. Its size is the smallest that is possible to hold all of *source*.

If *size* is present, the result is a rank-one array of size *size*.

If the physical representation of the result is larger than *source*, the result contains *source*'s bit pattern in its right-most bits; the left-most bits of the result are undefined.

If the physical representation of the result is smaller than *source*, the result contains the right-most bits of *source*'s bit pattern.

Examples

TRANSFER (1082130432, 0.0) has the value 4.0 (on processors that represent the values 4.0 and 1082130432 as the string of binary digits 0100 0000 1000 0000 0000 0000 0000 0000).

TRANSFER ((/2.2, 3.3, 4.4/), ((0.0, 0.0))) results in a scalar whose value is (2.2, 3.3).

TRANSFER ((/2.2, 3.3, 4.4/), (/(0.0, 0.0)/)) results in a complex rank-one array of length 2. Its first element is (2.2,3.3) and its second element has a real part with the value 4.4 and an undefined imaginary part.

TRANSFER ((/2.2, 3.3, 4.4/), (/(0.0, 0.0)/), 1) results in a complex rank-one array having one element with the value (2.2, 3.3).

TRANSPOSE

Description: Transposes an array of rank two.

Syntax: result = TRANSPOSE (*matrix*)

Class: Transformational function; Generic

Arguments: *matrix* must be a rank-two array (of any data type).

Results: The result is a rank-two array with the same type and kind parameters as *matrix*. Its shape is (n, m), where (m, n) is the shape of *matrix*. For example, if the shape of *matrix* is (4,6), the shape of the result is (6,4).

Element (i, j) of the result has the value *matrix* (j, i), where *i* is in the range 1 to n, and *j* is in the range 1 to m.

Examples

B is the array $\begin{bmatrix} 2 & 3 & 4 \\ 5 & 6 & 7 \\ 8 & 9 & 1 \end{bmatrix}$. TRANSPOSE (B) has the value $\begin{bmatrix} 2 & 5 & 8 \\ 3 & 6 & 9 \\ 4 & 7 & 1 \end{bmatrix}$.

TRIM

Description: Returns the argument with trailing blanks removed.

Syntax: result = TRIM (*string*)

Class: Transformational function; Generic

Arguments: *string* must be a scalar of type character.

Results: The result type is character with the same kind parameter as *string*. Its length is the length of *string* minus the number of trailing blanks in *string*.

The value of the result is the same as *string*, except any trailing blanks are removed. If *string* contains only blank characters, the result has zero length.

Examples

TRIM ('ΔΔ NAMEΔΔΔΔ') has the value 'ΔΔ NAME'.

TRIM ('ΔΔ CΔΔ DΔΔΔΔΔ') has the value 'ΔΔ CΔΔ D'.

UBOUND

Description: Returns the upper bounds for all dimensions of an array, or the upper bound for a specified dimension.

Syntax: result = UBOUND (*array* [, *dim*] [, *kind*])

Class: Inquiry function; Generic

Arguments:

array Must be an array (of any data type). It must not be an allocatable array that is not allocated, or a disassociated pointer. It can be an assumed-size array if *dim* is present with a value less than the rank of *array.*

dim (opt) Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of *array.*

kind (opt) Must be a scalar integer initialization expression.

Results: The result type is integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

If *dim* is present, the result is a scalar. Otherwise, the result is a rank-one array with one element for each dimension of *array*. Each element in the result corresponds to a dimension of *array*.

If *array* is an array section or an array expression that is not a whole array or array structure component, UBOUND (*array*, *dim*) has a value equal to the number of elements in the given dimension.

If *array* is a whole array or array structure component, UBOUND (*array*, *dim*) has a value equal to the upper bound for subscript *dim* of *array*, if *dim* is nonzero. If *dim* has size zero, the corresponding element of the result has the value zero.

The setting of compiler options specifying integer size can affect this function.

Examples

Consider the following:

```
REAL ARRAY_A (1:3, 5:8)
REAL ARRAY_B (2:8, -3:20)
```

UBOUND (ARRAY_A) is (3, 8). UBOUND (ARRAY_A, DIM=2) is 8.

UBOUND (ARRAY_B) is (8, 20). UBOUND (ARRAY_B (5:8, :)) is (4,24) because the number of elements is significant for array section arguments.

UNPACK

Description: Takes elements from a rank-one array and unpacks them into another (possibly larger) array under the control of a mask.

Syntax: result = UNPACK (*vector*, *mask*, *field*)

Class: Transformational function; Generic

Arguments:

vector Must be a rank-one array (of any data type). Its size must be at least t, where *t* is the number of true elements in *mask*.

mask Must be a logical array. It determines where elements of *vector* are placed when they are unpacked.

field Must be of the same type and type parameters as *vector* and conformable with *mask*. Elements in *field* are inserted into the result array when the corresponding *mask* element has the value false.

Results: The result is an array with the same shape as *mask*, and the same type and type parameters as *vector*.

Elements in the result array are filled in array element order. If element i of *mask* is true, the corresponding element of the result is filled by the next element in *vector*. Otherwise, it is filled by *field* (if *field* is scalar) or the ith element of *field* (if *field* is an array).

Examples

N is the array $\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$, P is the array (2, 3, 4, 5), and Q is the array $\begin{bmatrix} T & F & F \\ F & T & F \\ T & T & F \end{bmatrix}$.

UNPACK (P, MASK=Q, FIELD=N) produces the result $\begin{bmatrix} 2 & 0 & 1 \\ 1 & 4 & 1 \\ 3 & 5 & 0 \end{bmatrix}$.

UNPACK (P, MASK=Q, FIELD=1) produces the result $\begin{bmatrix} 2 & 1 & 1 \\ 1 & 4 & 1 \\ 3 & 5 & 1 \end{bmatrix}$.

VERIFY

Description: Verifies that a set of characters contains all the characters in a string by identifying the first character in the string that is not in the set.

Syntax: result = VERIFY (*string*, *set* [, *back*] [, *kind*])

Class: Elemental function; Generic

Arguments:

string Must be of type character.

set Must be of type character with the same kind parameter as *string*.

back (opt) Must be of type logical.

kind (opt) Must be a scalar integer initialization expression.

Results: The result type is integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

If *back* is omitted (or is present with the value false) and *string* has at least one character that is not in *set*, the value of the result is the position of the leftmost character of *string* that is not in *set*.

If *back* is present with the value true and *string* has at least one character that is not in *set*, the value of the result is the position of the rightmost character of *string* that is not in *set*.

If each character of *string* is in *set* or the length of *string* is zero, the value of the result is zero.

The setting of compiler options specifying integer size can affect this function.

Examples

VERIFY ('CDDDC', 'C') has the value 2.

VERIFY ('CDDDC', 'C', BACK=.TRUE.) has the value 4.

VERIFY ('CDDDC', 'CD') has the value zero.

ZEXT

Description: Extends an argument with zeros. This function is used primarily for bit-oriented operations. It cannot be passed as an actual argument.

Syntax: result = ZEXT (*x* [, *kind*])

Class: Elemental function; Generic

Arguments: Must be of type logical or integer.

x

kind (opt) Must be a scalar integer initialization expression.

Results: The result type is integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The result value is *x* extended with zeros and treated as an unsigned value.

The storage requirements for integer constants are never less than two bytes. Integer constants within the range of constants that can be represented by a single byte still require two bytes of storage.

The setting of compiler options specifying integer size can affect this function.

Specific Name[1]	Argument Type	Result Type
IZEXT	LOGICAL(1)	INTEGER(2)
	LOGICAL(2)	INTEGER(2)
	INTEGER(1)	INTEGER(2)
	INTEGER(2)	INTEGER(2)
JZEXT	LOGICAL(1)	INTEGER(4)
	LOGICAL(2)	INTEGER(4)
	LOGICAL(4)	INTEGER(4)
	INTEGER(1)	INTEGER(4)
	INTEGER(2)	INTEGER(4)
	INTEGER(4)	INTEGER(4)
KZEXT	LOGICAL(1)	INTEGER(8)
	LOGICAL(2)	INTEGER(8)
	LOGICAL(4)	INTEGER(8)
	LOGICAL(8)	INTEGER(8)
	INTEGER(1)	INTEGER(8)
	INTEGER(2)	INTEGER(8)
	INTEGER(4)	INTEGER(8)
	INTEGER(8)	INTEGER(8)

1. These specific functions cannot be passed as actual arguments.

Examples

Consider the following:

```
INTEGER(2) W_VAR /'FFFF'X/
INTEGER(4) L_VAR
L_VAR = ZEXT(W_VAR)
```

This example stores an INTEGER(2) quantity in the low-order 16 bits of an INTEGER(4) quantity, with the resulting value of L_VAR being '0000FFFF'X. If the ZEXT function had not been used, the resulting value would have been 'FFFFFFFF'X, because W_VAR would have been converted to the left-hand operand's data type by sign extension.

Data Transfer I/O Statements

10

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

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

- An overview of "Records and Files"
- "Components of Data Transfer Statements"
- Data transfer input statements:
  - "READ Statements"
  - "ACCEPT Statement"
- Data transfer output statements:
  - "WRITE Statements"
  - "PRINT and TYPE Statements"[1]
  - "REWRITE Statement"

File connection, file inquiry, and file positioning I/O statements are discussed in Chapter 12, "File Operation I/O Statements".

Records and Files

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

- Formatted

  A record containing formatted data that requires translation from internal to external form. Formatted I/O statements have explicit format specifiers (which can specify list-directed formatting) or namelist specifiers (for namelist formatting). Only formatted I/O statements can read formatted data.

1. TYPE statements are language extensions.

- Unformatted

  A record containing unformatted data that is not translated from internal form. An unformatted record can also contain no data. The internal representation of unformatted data is processor-dependent. Only unformatted I/O statements can read unformatted data.
- Endfile

  The last record of a file. An endfile record can be explicitly written to a sequential file by an ENDFILE statement (see "ENDFILE Statement").

A file is a sequence of records. There are two types of Fortran files, as follows:

- External

  A file that exists in a medium (such as computer disks) external to the executable program.

  Records in an external file must be either all formatted or all unformatted. There are two ways to access records in external files: sequential and direct access.

  In sequential access, records are processed in the order in which they appear in the file. In direct access, records are selected by record number, so they can be processed in any order.
- Internal

  Memory (internal storage) that behaves like a file. This type of file provides a way to transfer and convert data in memory from one format to another. The contents of these files are stored as scalar character variables.

See Also

Your user's guide for details on formatted and unformatted data transfers and external file access methods

Components of Data Transfer Statements

Data transfer statements take one of the following forms:

io-keyword (*io-control-list*) [*io-list*]

io-keyword format [, *io-list*]

io-keyword

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

io-control-list

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

[UNIT=]*io-unit*	[NML=]*group*	END	ERR	REC
[FMT=]*format*	ADVANCE	EOR	IOSTAT	SIZE

io-list

Is an I/O list, which can contain variables (except for assumed-size arrays) or implied-DO lists. Output statements can contain constants or expressions.

format

Is the nonkeyword form of a control-list format specifier (no FMT=).

If a format specifier ([FMT=]format) or namelist specifier ([NML=]group) is present, the data transfer statement is called a formatted I/O statement; otherwise, it is an unformatted I/O statement.

If a record specifier (REC) is present, the data transfer statement is a direct-access I/O statement; otherwise, it is a sequential-access I/O statement.

If an error, end-of-record, or end-of-file condition occurs during data transfer, file positioning and execution are affected, and certain control-list specifiers (if present) become defined. (For more information, see "Branch Specifiers".)

The following sections discuss the "I/O Control List" and "I/O Lists".

I/O Control List

The I/O control list specifies one or more of the following:

- The I/O unit to act upon ([UNIT=]*io-unit*)
  This specifier *must* be present; the rest are optional.
- The format (explicit or list-directed) to use for data editing; if explicit, the keyword form must appear ([FMT=]*format*)
- The namelist group name to act upon ([NML=]*group*)
- The number of a record to access (REC)
- The name of a variable that contains the completion status of an I/O operation (IOSTAT)
- The label of the statement that receives control if an error (ERR), end-of-file (END), or end-of-record (EOR) condition occurs
- Whether you want to use advancing or nonadvancing I/O (ADVANCE)
- The number of characters read from a record (SIZE) by a nonadvancing READ statement

No control specifier can appear more than once, and the list must not contain both a format specifier and namelist group name specifier.

Control specifiers can take any of the following forms:

- Keyword form
- When the keyword form (for example, UNIT=*io-unit*) is used for all control-list specifiers in an I/O statement, the specifiers can appear in any order.

- Nonkeyword form

  When the nonkeyword form (for example, *io-unit*) is used for all control-list specifiers in an I/O statement, the io-unit specifier must be the first item in the control list. If a format specifier or namelist group name specifier is used, it must immediately follow the io-unit specifier.
- Mixed form

  When a mix of keyword and nonkeyword forms is used for control-list specifiers in an I/O statement, the nonkeyword values must appear first. Once a keyword form of a specifier is used, all specifiers to the right must also be keyword forms.

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

Unit Specifier

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

[UNIT=]*io-unit*

io-unit

For external files, it identifies a logical unit and is one of the following:

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

  Units 5, 6, and 0 are associated with preconnected units.
- An asterisk (*). This is the default (or implicit) external unit, which is preconnected for formatted sequential access.

For internal files, *io-unit* identifies a scalar or array character variable that is an internal file. An internal file is designated internal storage space (a variable buffer) that is used with formatted (including list-directed) sequential READ and WRITE statements.

The *io-unit* must be specified in a control list. If the keyword UNIT is omitted, the *io-unit* must be first in the control list.

A unit number is assigned either explicitly through an OPEN statement or implicitly by the system. If a READ statement implicitly opens a file, the file's status is STATUS='OLD'. If a WRITE statement implicitly opens a file, the file's status is STATUS='UNKNOWN'.

If the internal file is a *scalar* character variable, the file has only one record; its length is equal to that of the variable.

If the internal file is an *array* character variable, the file has a record for each element in the array; each record's length is equal to one array element.

An internal file can be read only if the variable has been defined and a value assigned to each record in the file. If the variable representing the internal file is a pointer, it must be associated; if the variable is an allocatable array, it must be currently allocated.

Before data transfer, an internal file is always positioned at the beginning of the first character of the first record.

See Also

- "OPEN Statement"
- Your user's guide for details on implicit logical assignments, preconnected units, and using internal files

Format Specifier

The format specifier indicates the format to use for data editing. It takes the following form:

[FMT=]*format*

format

Is one of the following:

- The statement label of a FORMAT statement

  The FORMAT statement must be in the same scoping unit as the data transfer statement.
- An asterisk (*), indicating list-directed formatting
- A scalar default integer variable that has been assigned the label of a FORMAT statement (through an ASSIGN statement)

  The FORMAT statement must be in the same scoping unit as the data transfer statement.
- A character expression (which can be an array or character constant) containing the run-time format

  A default character expression must evaluate to a valid format specification. If the expression is an array, it is treated as if all the elements of the array were specified in array element order and were concatenated.
- The name of a numeric array (or array element) containing the format

If the keyword FMT is omitted, the format specifier must be the second specifier in the control list; the io-unit specifier must be first.

If a format specifier appears in a control list, a namelist group specifier must not appear.

See Also

- "Interaction Between Format Specifications and I/O Lists"
- "Format Specifications" for details on FORMAT statements
- "Rules for List-Directed Sequential READ Statements" for details on list-directed input

- "Rules for List-Directed Sequential WRITE Statements" for details on list-directed output

Namelist Specifier

The namelist specifier indicates namelist formatting and identifies the namelist group for data transfer. It takes the following form:

[NML=]*group*

group

Is the name of a namelist group previously declared in a NAMELIST statement.

If the keyword NML is omitted, the namelist specifier must be the second specifier in the control list; the io-unit specifier must be first.

If a namelist specifier appears in a control list, a format specifier must not appear.

See Also

- "Rules for Namelist Sequential READ Statements" for details on namelist input
- "Rules for Namelist Sequential WRITE Statements" for details on namelist output

Record Specifier

The record specifier identifies the number of the record for data transfer in a file connected for direct access. It takes the following form:

REC=*r*

r

Is a scalar numeric expression indicating the record number. The value of the expression must be greater than or equal to 1, and less than or equal to the maximum number of records allowed in the file.

If necessary, the value is converted to integer data type before use.

If REC is present, no END specifier, * format specifier, or namelist group name can appear in the same control list.

See Also

"Alternative Syntax for a Record Specifier"

I/O Status Specifier

The I/O status specifier designates a variable to store a value indicating the status of a data transfer operation. It takes the following form:

IOSTAT=*i-var*

i-var

Is a scalar integer variable. When a data transfer statement is executed, *i-var* is set to one of the following values:

A positive integer	Indicating an error condition occurred.
A negative integer	Indicating an end-of-file or end-of-record condition occurred. The negative integers differ depending on which condition occurred.
Zero	Indicating no error, end-of-file, or end-of-record condition occurred.

Execution continues with the statement following the data transfer statement, or the statement identified by a branch specifier (if any).

An end-of-file condition occurs only during execution of a sequential READ statement; an end-of-record condition occurs only during execution of a nonadvancing READ statement.

See Also

Your user's guide for details on the error numbers returned by IOSTAT

Branch Specifiers

A branch specifier identifies a branch target statement that receives control if an error, end-of-file, or end-of-record condition occurs. There are three branch specifiers, taking the following forms:

ERR=*label*
END=*label*
EOR=*label*

label

Is the label of the branch target statement that receives control when the specified condition occurs.

The branch target statement must be in the same scoping unit as the data transfer statement.

The following rules apply to these specifiers:

- ERR

  The error specifier can appear in a sequential access READ or WRITE statement, a direct-access READ statement, or a REWRITE statement.

  If an error condition occurs, the position of the file is indeterminate, and execution of the statement terminates.

  If IOSTAT was specified, the IOSTAT variable becomes defined as a positive integer value. If SIZE was specified (in a nonadvancing READ statement), the SIZE variable becomes defined as an integer value. If an ERR=*label* was specified, execution continues with the labeled statement.

- END

  The end-of-file specifier can appear only in a sequential access READ statement.

  An end-of-file condition occurs when no more records exist in a file during a sequential read, or when an end-of-file record produced by the ENDFILE statement is encountered. End-of-file conditions do not occur in direct-access READ statements.

  If an end-of-file condition occurs, the file is positioned after the end-of-file record, and execution of the statement terminates.

  If IOSTAT was specified, the IOSTAT variable becomes defined as a negative integer value. If an END=*label* was specified, execution continues with the labeled statement.
- EOR

  The end-of-record specifier can appear only in a formatted, sequential access READ statement that has the specifier ADVANCE='NO'(nonadvancing input).

  An end-of-record condition occurs when a nonadvancing READ statement tries to transfer data from a position after the end of a record.

  If an end-of-record condition occurs, the file is positioned after the current record, and execution of the statement terminates.

  If IOSTAT was specified, the IOSTAT variable becomes defined as a negative integer value. If PAD='YES' was specified for file connection, the record is padded with blanks (as necessary) to satisfy the input item list and the corresponding data edit descriptor. If SIZE was specified, the SIZE variable becomes defined as an integer value. If an EOR=*label* was specified, execution continues with the labeled statement.

If one of the conditions occurs, no branch specifier appears in the control list, but an IOSTAT specifier appears, execution continues with the statement following the I/O statement. If neither a branch specifier nor an IOSTAT specifier appears, the program terminates.

See Also

- "Branch Statements" for details on branch target statements
- "I/O Status Specifier" for details on the IOSTAT specifier
- Your user's guide for details on error processing

Advance Specifier

The advance specifier determines whether nonadvancing I/O occurs for a data transfer statement. It takes the following form:

ADVANCE=*c-expr*

c-expr

Is a scalar character expression that evaluates to 'YES' for advancing I/O or 'NO' for nonadvancing I/O. The default value is 'YES'.

Trailing blanks in the expression are ignored.

The ADVANCE specifier can appear only in a formatted, sequential data transfer statement that specifies an external unit. It must not be specified for list-directed or namelist data transfer.

Advancing I/O always positions a file at the end of a record, unless an error condition occurs. Nonadvancing I/O can position a file at a character position within the current record.

See Also

Your user's guide for details on advancing and nonadvancing I/O

Character Count Specifier

The character count specifier defines a variable to contain the count of how many characters are read when a nonadvancing READ statement terminates. It takes the following form:

SIZE=*i-var*

i-var

Is a scalar integer variable.

If PAD='YES' was specified for file connection, blanks inserted as padding are not counted.

The SIZE specifier can appear only in a formatted, sequential READ statement that has the specifier ADVANCE='NO' (nonadvancing input). It must not be specified for list-directed or namelist data transfer.

I/O Lists

In a data transfer statement, the I/O list specifies the entities whose values will be transferred. The I/O list is either an implied-DO list or a simple list of variables (except for assumed-size arrays).

In input statements, the I/O list cannot contain constants and expressions because these do not specify named memory locations that can be referenced later in the program.

However, constants and expressions can appear in the I/O lists for output statements because the compiler can use temporary memory locations to hold these values during the execution of the I/O statement.

If an input item is a pointer, it must be currently associated with a definable target; data is transferred from the file to the associated target. If an output item is a pointer, it must be currently associated with a target; data is transferred from the target to the file.

If an input or output item is an array, it is treated as if the elements (if any) were specified in array element order. For example, if ARRAY_A is an array of shape (2,1), the following input statements are equivalent:

```
READ *, ARRAY_A
READ *, ARRAY_A(1,1), ARRAY_A(2,1)
```

However, no element of that array can affect the value of any expression in the input list, nor can any element appear more than once in an input list. For example, the following input statements are invalid:

```
INTEGER  B(50)
...
READ *, B(B)
READ *, B(B(1):B(10))
```

If an input or output item is an allocatable array, it must be currently allocated.

If an input or output item is a derived type, the following rules apply:

- Any derived-type component must be in the scoping unit containing the I/O statement.
- The derived type must not have a pointer component.
- In a formatted I/O statement, a derived type is treated as if all of the components of the structure were specified in the same order as in the derived-type definition.
- In an unformatted I/O statement, a derived type is treated as a single object.

The following sections describe simple list items in I/O lists, and implied-DO lists in I/O lists.

Simple List Items in I/O Lists

In a data transfer statement, a simple list of items takes the following form:

item [, *item*]...

item

Is one of the following:

- For input statements: a variable name

  The variable must not be an assumed-size array, unless one of the following appears in the last dimension: a subscript, a vector subscript, or a section subscript specifying an upper bound.
- For output statements: a variable name, expression, or constant

  Any expression must not attempt further I/O operations on the same logical unit. For example, it must not refer to a function subprogram that performs I/O on the same logical unit.

The data transfer statement assigns values to (or transfers values from) the list items in the order in which the items appear, from left to right.

When multiple array names are used in the I/O list of an unformatted input or output statement, only one record is read or written, regardless of how many array name references appear in the list.

Examples

The following example shows a simple I/O list:

```
WRITE (6,10) J, K(3), 4, (L+4)/2, N
```

When you use an array name reference in an I/O list, an input statement reads enough data to fill every item of the array. An output statement writes all of the values in the array.

Data transfer begins with the initial item of the array and proceeds in the order of subscript progression, with the leftmost subscript varying most rapidly. The following statement defines a two-dimensional array:

```
DIMENSION ARRAY(3,3)
```

If the name ARRAY appears with no subscripts in a READ statement, that statement assigns values from the input record(s) to ARRAY(1,1), ARRAY(2,1), ARRAY(3,1), ARRAY(1,2), and so on through ARRAY(3,3).

An input record contains the following values:

```
  1,3,721.73
```

The following example shows how variables in the I/O list can be used in array subscripts later in the list:

```
DIMENSION ARRAY(3,3)
...
READ (1,30) J, K, ARRAY(J,K)
```

When the READ statement is executed, the first input value is assigned to J and the second to K, establishing the subscript values for ARRAY(J,K). The value 721.73 is then assigned to ARRAY(1,3). Note that the variables must appear before their use as array subscripts.

Consider the following derived-type definition and structure declaration:

```
TYPE EMPLOYEE
  INTEGER ID
  CHARACTER(LEN=40) NAME
END TYPE EMPLOYEE
...
TYPE(EMPLOYEE) :: CONTRACT   ! A structure of type EMPLOYEE
```

The following statements are equivalent:

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

See Also

"I/O Lists"

implied-DO Lists in I/O Lists

In a data transfer statement, an implied-DO list acts as though it were a part of an I/O statement within a DO loop. It takes the following form:

(*list*, *do-var* = *expr*1, *expr*2 [, *expr*3])

list

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

do-var

Is the name of a scalar integer or real variable. The variable must not be one of the input items in *list*.

expr

Are scalar numeric expressions of type integer or real. They do not all have to be the same type, or the same type as the DO variable.

The implied-DO loop is initiated, executed, and terminated in the same way as a DO construct.

The *list* is the range of the implied-DO loop. Items in that list can refer to *do-var*, but they must not change the value of *do-var*.

Two nested implied-DO lists must not have the same (or an associated) DO variable.

Use an implied-DO list to do the following:

- Specify iteration of part of an I/O list
- Transfer part of an array
- Transfer array items in a sequence different from the order of subscript progression

If the I/O statement containing an implied-DO list terminates abnormally (with an END, EOR, or ERR branch or with an IOSTAT value other than zero), the DO variable becomes undefined.

Examples

The following two output statements are equivalent:

```
WRITE (3,200) (A,B,C, I=1,3)          ! An implied-DO list
WRITE (3,200) A,B,C,A,B,C,A,B,C       ! A simple item list
```

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

```
WRITE (6,150) ((FORM(K,L), L=1,10), K=1,10,2)
```

The inner DO loop is executed 10 times for each iteration of the outer loop; the second subscript (L) advances from 1 through 10 for each increment of the first subscript (K). This is the reverse of the normal array element order. Note that K is incremented by 2, so only the odd-numbered rows of the array are output.

In the following example, the entire list of the implied-DO list (P(1), Q(1,1), Q(1,2)...,Q(1,10)) are read before I is incremented to 2:

```
READ (5,999) (P(I), (Q(I,J), J=1,10), I=1,5)
```

The following example uses fixed subscripts and subscripts that vary according to the implied-DO list:

```
READ (3,5555) (BOX(1,J), J=1,10)
```

Input values are assigned to BOX(1,1) through BOX(1,10), but other elements of the array are not affected.

The following example shows how a DO variable can be output directly:

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

Integers 1 through 20 are written.

See Also

- "I/O Lists"
- "DO Constructs"

READ Statements

The READ statement is a data transfer input statement. Data can be input from external sequential or direct-access records, or from internal records.

Forms for Sequential READ Statements

Sequential READ statements transfer input data from external sequential-access records. The statements can be formatted with format specifiers (which can use list-directed formatting) or namelist specifiers (for namelist formatting), or they can be unformatted.

Sequential READ statements take one of the following forms:

Formatted:

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

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

Formatted - List-Directed:

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

READ * [, *io-list*]

Formatted - Namelist:

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

READ *nml*

Unformatted:

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

eunit

Is an external unit specifier ([UNIT=]*io-unit*).

format

Is a format specifier ([FMT=]*format*).

advance

Is an advance specifier (ADVANCE=*c-expr*). If the value of *c-expr* is 'YES', the statement uses advancing input; if the value is 'NO', the statement uses nonadvancing input. The default value is 'YES'.

size

Is a character count specifier (SIZE=*i-var*). It can only be specified for nonadvancing READ statements.

iostat

Is a status specifier (IOSTAT=*i-var*).

err, end, eor

Are branch specifiers if an error (ERR=*label*), end-of-file (END=*label*), or end-of-record (EOR=*label*) condition occurs.

EOR can only be specified for nonadvancing READ statements.

io-list

Is an I/O list.

form

Is the nonkeyword form of a format specifier (no FMT=).

*

Is the format specifier indicating list-directed formatting. (It can also be specified as FMT=*.)

nml-group

Is a namelist specifier ([NML=]group) indicating namelist formatting.

nml

Is the nonkeyword form of a namelist specifier (no NML=) indicating namelist formatting.

See Also

- "I/O Control List" for details on I/O control-list specifiers

- "I/O Lists" for details on the general rules for I/O lists
- "Advance Specifier" and your user's guide for details on advancing I/O
- Your user's guide for details on file sharing

Rules for Formatted Sequential READ Statements

Formatted, sequential READ statements translate data from character to binary form by using format specifications for editing (if any). The translated data is assigned to the entities in the I/O list in the order in which the entities appear, from left to right.

Values can be transferred to objects of intrinsic or derived types. For derived types, values of intrinsic types are transferred to the components of intrinsic types that ultimately make up these structured objects.

For data transfer, the file must be positioned so that the record read is a formatted record or an end-of-file record.

If the number of I/O list items is *less* than the number of fields in an input record, the statement ignores the excess fields.

If the number of I/O list items is *greater* than the number of fields in an input record, the input record is padded with blanks. However, if PAD='NO' was specified for file connection, the input list and file specification must not require more characters from the record than it contains. If more characters are required and nonadvancing input is in effect, an end-of-record condition occurs.

If the file is connected for unformatted I/O, formatted data transfer is prohibited.

Example

The following example shows formatted, sequential READ statements:

```
READ (*, '(B)', ADVANCE='NO') C
READ (FMT="(E2.4)", UNIT=6, IOSTAT=IO_STATUS) A, B, C
```

Rules for List-Directed Sequential READ Statements

List-directed, sequential READ statements translate data from character to binary form by using the data types of the corresponding I/O list item to determine the form of the data. The translated data is then assigned to the entities in the I/O list in the order in which they appear, from left to right.

If a slash (/) is encountered during execution, the READ statement is terminated, and any remaining input list items are unchanged.

If the file is connected for unformatted I/O, list-directed data transfer is prohibited.

List-Directed Records

A list-directed external record consists of a sequence of values and value separators. A value can be any of the following:

- A constant

  Each constant must be a literal constant of type integer, real, complex, logical, or character; or a nondelimited character string. Binary, octal, hexadecimal, Hollerith, and named constants are not permitted.

  In general, the form of the constant must be acceptable for the type of the list item. The data type of the constant determines the data type of the value and the translation from external to internal form. The following rules also apply:

  — A numeric list item can correspond only to a numeric constant, and a character list item can correspond only to a character constant. If the data types of a numeric list element and its corresponding numeric constant do not match, conversion is performed according to the rules for arithmetic assignment (see Table 4-2).

  — A complex constant has the form of a pair of real or integer constants separated by a comma and enclosed in parentheses. Blanks can appear between the opening parenthesis and the first constant, before and after the separating comma, and between the second constant and the closing parenthesis.

  — A logical constant represents true values (.TRUE. or any value beginning with T, .T, t, or .t) or false values (.FALSE. or any value beginning with F, .F, f, or .f).

  A character string does not need delimiting apostrophes or quotation marks if the corresponding I/O list item is of type default character, and the following is true:

  — The character string does not contain a blank, comma, or slash.

  — The character string is not continued across a record boundary.

  — The first nonblank character in the string is not an apostrophe or a quotation mark.

  — The leading character is not a string of digits followed by an asterisk.

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

- A null value

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

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

- A repetition of a null value (r*) or a constant (r*constant), where *r* is an unsigned, nonzero, integer literal constant with no kind parameter, and no embedded blanks.

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

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

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

Example

Suppose the following statements are specified:

```
CHARACTER*14 C
DOUBLE PRECISION T
COMPLEX D,E
LOGICAL L,M
READ (1,*) I,R,D,E,L,M,J,K,S,T,C,A,B
```

Then suppose the following external record is read:

```
4 6.3 (3.4,4.2), (3, 2 ) , T,F,,3*14.6 ,'ABC,DEF/GHI''JK'/
```

The following values are assigned to the I/O list items:

I/O List Item	Value Assigned
I	4
R	6.3
D	(3.4,4.2)
E	(3.0,2.0)
L	.TRUE.
M	.FALSE.
J	Unchanged
K	14
S	14.6
T	14.6D0
C	ABC,DEF/GHI' JK
A	Unchanged

I/O List Item	Value Assigned
B	Unchanged

See Also

- "Rules for Formatted Sequential READ Statements"
- "Intrinsic Data Types" for details on the literal constant forms of intrinsic data types
- "Rules for List-Directed Sequential WRITE Statements" for details on list-directed output

Rules for Namelist Sequential READ Statements

Namelist, sequential READ statements translate data from external to internal form by using the data types of the objects in the corresponding NAMELIST statement to determine the form of the data. The translated data is assigned to the specified objects in the namelist group in the order in which they appear, from left to right.

If a slash (/) is encountered during execution, the READ statement is terminated, and any remaining input list items are unchanged.

If the file is connected for unformatted I/O, namelist data transfer is prohibited.

Namelist Records

A namelist external record takes the following form:

&*group-name object* = *value* [, *object* = *value*].../

group-name

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

object

Is the name (or subobject designator) of an entity defined in the NAMELIST declaration of the group name. The object name must not contain embedded blanks except within the parentheses of a subscript or substring specifier. Each object must be contained in a single record.

value

Is any of the following:

- A constant

  Each constant must be a literal constant of type integer, real, complex, logical, or character; or a nondelimited character string. Binary, octal, hexadecimal, Hollerith, and named constants are not permitted.

  In general, the form of the constant must be acceptable for the type of the list item. The data type of the constant determines the data type of the value and the translation from external to internal form. The following rules also apply:

  — A numeric list item can correspond only to a numeric constant, and a character list item can correspond only to a character constant. If the data types of a numeric list element and its corresponding numeric constant do not match, conversion is performed according to the rules for arithmetic assignment (see Table 4-2).

  — A complex constant has the form of a pair of real or integer constants separated by a comma and enclosed in parentheses. Blanks can appear between the opening parenthesis and the first constant, before and after the separating comma, and between the second constant and the closing parenthesis.

  — A logical constant represents true values (.TRUE. or any value beginning with T, .T, t, or .t) or false values (.FALSE. or any value beginning with F, .F, f, or .f).

  A character string does not need delimiting apostrophes or quotation marks if the corresponding NAMELIST item is of type default character, and the following is true:

  — The character string does not contain a blank, comma (,), slash (/), exclamation point(!), ampersand (&), dollar sign ($), left parenthesis, equal sign (=), percent sign (%), or period (.).

  — The character string is not continued across a record boundary.

  — The first nonblank character in the string is not an apostrophe or a quotation mark.

  — The leading characters are not a string of digits followed by an asterisk.

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

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

- A null value

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

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

- A repetition of a null value (r*) or a constant (r*constant), where *r* is an unsigned, nonzero, integer literal constant with no kind parameter, and no embedded blanks.

Blanks can precede or follow the beginning ampersand (&), follow the group name, precede or follow the equal sign, or precede the terminating slash.

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

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

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

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

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

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

Prompting for Namelist Group Information

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

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

For example, consider the following statements:

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

READ (5,NML=NLIST)
WRITE (6,NML=NLIST)
END
```

During execution, if a blank followed by ? is entered on a terminal device, the following values are displayed:

```
 &NLIST
   A
   B
   C
 /
```

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

```
 &NLIST
   A  =   1.500000    ,
   B  =           2,
   C  = ABCDE
 /
```

Examples

Suppose the following statements are specified:

```
NAMELIST /CONTROL/ TITLE, RESET, START, STOP, INTERVAL
CHARACTER*10 TITLE
REAL(KIND=8) START, STOP
LOGICAL(KIND=4) RESET
INTEGER(KIND=4) INTERVAL
READ (UNIT=1, NML=CONTROL)
```

The NAMELIST statement associates the group name CONTROL with a list of five objects. The corresponding READ statement reads the following input data from unit 1:

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

The following values are assigned to objects in group CONTROL:

Namelist Object	Value Assigned
TITLE	TESTT002AA
RESET	T

Namelist Object	Value Assigned
START	10.2
STOP	14.5
INTERVAL	1

It is not necessary to assign values to all of the objects declared in the corresponding NAMELIST group. If a namelist object does not appear in the input statement, its value (if any) is unchanged.

Similarly, when character substrings and array elements are specified, only the values of the specified variable substrings and array elements are changed. For example, suppose the following input is read:

```
&CONTROL TITLE(9:10)='BB' /
```

The new value for TITLE is TESTT002BB; only the last two characters in the variable change.

The following example shows an array as an object:

```
DIMENSION ARRAY_A(20)
NAMELIST /ELEM/ ARRAY_A
READ (UNIT=1,NML=ELEM)
```

Suppose the following input is read:

```
&ELEM
ARRAY_A=1.1, 1.2, , 1.4
/
```

The following values are assigned to the ARRAY_A elements:

Array Element	Value Assigned
ARRAY_A(1)	1.1
ARRAY_A(2)	1.2
ARRAY_A(3)	Unchanged
ARRAY_A(4)	1.4
ARRAY_A(5)...ARRAY_A(20)	Unchanged

When a list of values is assigned to an array element, the assignment begins with the specified array element, rather than with the first element of the array. For example, suppose the following input is read:

```
&ELEM
ARRAY_A(3)=34.54, 45.34, 87.63, 3*20.00
/
```

New values are assigned only to array ARRAY_A elements 3 through 8. The other element values are unchanged.

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

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

The output file NMLTEST.DAT will contain:

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

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

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

The result is the following:

```
CHARR read in >AAA<  >   <  >   <  >   <
```

See Also

- "Alternative Form for Namelist External Records"
- "Rules for Formatted Sequential READ Statements"
- "NAMELIST Statement" for the rules for objects in a namelist group
- "Rules for Namelist Sequential WRITE Statements" for details on namelist output

Rules for Unformatted Sequential READ Statements

Unformatted, sequential READ statements transfer binary data (without translation) between the current record and the entities specified in the I/O list. Only one record is read.

Objects of intrinsic or derived types can be transferred.

For data transfer, the file must be positioned so that the record read is an unformatted record or an end-of-file record.

The unformatted, sequential READ statement reads a single record. Each value in the record must be of the same type as the corresponding entity in the input list, unless the value is real or complex.

If the value is real or complex, one complex value can correspond to two real list entities, or two real values can correspond to one complex list entity. The corresponding values and entities must have the same kind parameter.

If the number of I/O list items is *less* than the number of fields in an input record, the statement ignores the excess fields. If the number of I/O list items is *greater* than the number of fields in an input record, an error occurs.

If a statement contains no I/O list, it skips over one full record, positioning the file to read the following record on the next execution of a READ statement.

If the file is connected for formatted, list-directed, or namelist I/O, unformatted data transfer is prohibited.

Example

The following example shows an unformatted, sequential READ statement:

```
READ (UNIT=6, IOSTAT=IO_STATUS) A, B, C
```

Forms for Direct-Access READ Statements

Direct-access READ statements transfer input data from external records with direct access. (The attributes of a direct-access file are established by the OPEN statement.)

A direct-access READ statement can be formatted or unformatted, and takes one of the following forms:

Formatted:

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

Unformatted:

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

eunit

Is an external unit specifier ([UNIT=]*io-unit*).

format

Is a format specifier ([FMT=]*format*). It must not be an asterisk (*).

rec

Is a record specifier (REC=*r*).

iostat

Is a status specifier (IOSTAT=*i-var*).

err

Is a branch specifier (ERR=*label*) if an error condition occurs.

io-list

Is an I/O list.

See Also

- "I/O Control List" for details on I/O control-list specifiers
- "I/O Lists" for the general rules for I/O lists
- Your user's guide for details on file sharing

Rules for Formatted Direct-Access READ Statements

Formatted, direct-access READ statements translate data from character to binary form by using format specifications for editing (if any). The translated data is assigned to the entities in the I/O list in the order in which the entities appear, from left to right.

Values can be transferred to objects of intrinsic or derived types. For derived types, values of intrinsic types are transferred to the components of intrinsic types that ultimately make up these structured objects.

For data transfer, the file must be positioned so that the record read is a formatted record or an end-of-file record.

If the number of I/O list items is *less* than the number of fields in an input record, the statement ignores the excess fields.

If the number of I/O list items is *greater* than the number of fields in an input record, the input record is padded with blanks. However, if PAD='NO' was specified for file connection, the input list and file specification must not require more characters from the record than it contains. If more characters are required and nonadvancing input is in effect, an end-of-record condition occurs.

If the format specification specifies another record, the record number is increased by one as each subsequent record is read by that input statement.

Example

The following example shows a formatted, direct-access READ statement:

```
READ (2, REC=35, FMT=10) (NUM(K), K=1,10)
```

Rules for Unformatted Direct-Access READ Statements

Unformatted, direct-access READ statements transfer binary data (without translation) between the current record and the entities specified in the I/O list. Only one record is read.

Objects of intrinsic or derived types can be transferred.

For data transfer, the file must be positioned so that the record read is an unformatted record or an end-of-file record.

The unformatted, direct-access READ statement reads a single record. Each value in the record must be of the same type as the corresponding entity in the input list, unless the value is real or complex.

If the value is real or complex, one complex value can correspond to two real list entities, or two real values can correspond to one complex list entity. The corresponding values and entities must have the same kind parameter.

If the number of I/O list items is less than the number of fields in an input record, the statement ignores the excess fields. If the number of I/O list items is greater than the number of fields in an input record, an error occurs.

If the file is connected for formatted, list-directed, or namelist I/O, unformatted data transfer is prohibited.

Example

The following example shows unformatted, direct-access READ statements:

```
READ (1, REC=10) LIST(1), LIST(8)
READ (4, REC=58, IOSTAT=K, ERR=500) (RHO(N), N=1,5)
```

Forms and Rules for Internal READ Statements

Internal READ statements transfer input data from an internal file.

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

An internal READ statement takes the following form:

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

iunit

Is an internal unit specifier ([UNIT=]*io-unit*). It must be a character variable. It must not be an array section with a vector subscript.

format

Is a format specifier ([FMT=]*format*). An asterisk (*) indicates list-directed formatting.

iostat

Is a status specifier (IOSTAT=*i-var*).

err, end

Are branch specifiers if an error (ERR=*label*) or end-of-file (END=*label*) condition occurs.

io-list

Is an I/O list.

Formatted, internal READ statements translate data from character to binary form by using format specifications for editing (if any). The translated data is assigned to the entities in the I/O list in the order in which the entities appear, from left to right.

This form of READ statement behaves as if the format begins with a BN edit descriptor. (You can override this behavior by explicitly specifying the BZ edit descriptor.)

Values can be transferred to objects of intrinsic or derived types. For derived types, values of intrinsic types are transferred to the components of intrinsic types that ultimately make up these structured objects.

Before data transfer occurs, the file is positioned at the beginning of the first record. This record becomes the current record.

If the number of I/O list items is *less* than the number of fields in an input record, the statement ignores the excess fields.

If the number of I/O list items is *greater* than the number of fields in an input record, the input record is padded with blanks. However, if PAD='NO' was specified for file connection, the input list and file specification must not require more characters from the record than it contains.

In list-directed formatting, character strings have no delimiters.

Example

The following program segment reads a record and examines the first character to determine whether the remaining data should be interpreted as decimal, octal, or hexadecimal. It then uses internal READ statements to make appropriate conversions from character string representations to binary.

```
INTEGER IVAL
CHARACTER TYPE, RECORD*80
CHARACTER*(*) AFMT, IFMT, OFMT, ZFMT
PARAMETER (AFMT='(Q,A)', IFMT= '(I10)', OFMT= '(O11)',        &
           ZFMT= '(Z8)')
ACCEPT AFMT, ILEN, RECORD
TYPE = RECORD(1:1)
IF (TYPE .EQ. 'D') THEN
```

```
    READ (RECORD(2:MIN(ILEN, 11)), IFMT) IVAL
ELSE IF (TYPE .EQ. 'O') THEN
    READ (RECORD(2:MIN(ILEN, 12)), OFMT) IVAL
ELSE IF (TYPE .EQ. 'X') THEN
    READ (RECORD(2:MIN(ILEN, 9)),ZFMT) IVAL
ELSE
    PRINT *, 'ERROR'
END IF
END
```

See Also

- "I/O Control List" for details on I/O control-list specifiers
- "I/O Lists" for the general rules for I/O lists
- "Rules for List-Directed Sequential READ Statements" for details on list-directed input
- Your user's guide for details on using internal files

ACCEPT Statement

The ACCEPT statement is a data transfer input statement. This statement is the same as a formatted, sequential READ statement, except that an ACCEPT statement must never be connected to user-specified I/O units.

An ACCEPT statement takes one of the following forms:

Formatted:

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

Formatted - List-Directed:

ACCEPT * [, *io-list*]

Formatted - Namelist:

ACCEPT *nml*

form

Is the nonkeyword form of a format specifier (no FMT=).

io-list

Is an I/O list.

*

Is the format specifier indicating list-directed formatting.

nml

Is the nonkeyword form of a namelist specifier (no NML=) indicating namelist formatting.

Example

In the following example, character data is read from the implicit unit and binary values are assigned to each of the five elements of array CHARAR:

```
      CHARACTER*10 CHARAR(5)
      ACCEPT 200, CHARAR
200   FORMAT (5A10)
```

See Also

- "Forms for Sequential READ Statements"
- "I/O Control List" for details on I/O control-list specifiers
- "I/O Lists" for the general rules for I/O lists
- "Rules for List-Directed Sequential READ Statements" for details on list-directed input
- "Rules for Namelist Sequential READ Statements" for details on namelist input
- Your user's guide for details on formatted data and data transfers

WRITE Statements

The WRITE statement is a data transfer output statement. Data can be output to external sequential or direct-access records, or to internal records.

Forms for Sequential WRITE Statements

Sequential WRITE statements transfer output data to external sequential access records. The statements can be formatted by using format specifiers (which can use list-directed formatting) or namelist specifiers (for namelist formatting), or they can be unformatted.

A sequential WRITE statement takes one of the following forms:

Formatted:

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

Formatted - List-Directed:

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

Formatted - Namelist:

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

Unformatted:

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

eunit

Is an external unit specifier ([UNIT=]*io-unit*).

format

Is a format specifier ([FMT=]*format*).

advance

Is an advance specifier (ADVANCE=*c-expr*). If the value of *c-expr* is 'YES', the statement uses advancing output; if the value is 'NO', the statement uses nonadvancing output. The default value is 'YES'.

iostat

Is a status specifier (IOSTAT=*i-var*).

err

Is a branch specifier (ERR=*label*) if an error condition occurs.

io-list

Is an I/O list.

*

Is the format specifier indicating list-directed formatting. (It can also be specified as FMT=*.)

nml-group

Is a namelist specifier ([NML=]*group*) indicating namelist formatting.

See Also

- "I/O Control List" for details on I/O control-list specifiers
- "I/O Lists" for the general rules for I/O lists
- "Advance Specifier" for details on advancing I/O

Rules for Formatted Sequential WRITE Statements

Formatted, sequential WRITE statements translate data from binary to character form by using format specifications for editing (if any). The translated data is written to an external file that is connected for sequential access.

Values can be transferred from objects of intrinsic or derived types. For derived types, values of intrinsic types are transferred from the components of intrinsic types that ultimately make up these structured objects.

The output list and format specification must not specify more characters for a record than the record size. (Record size is specified by RECL in an OPEN statement.)

If the file is connected for unformatted I/O, formatted data transfer is prohibited.

Example

The following example shows formatted, sequential WRITE statements:

```
WRITE (UNIT=8, FMT='(B)', ADVANCE='NO') C
WRITE (*, &quot;(F6.5)&quot;, ERR=25, IOSTAT=IO_STATUS) A, B, C
```

Rules for List-Directed Sequential WRITE Statements

List-directed, sequential WRITE statements transfer data from binary to character form by using the data types of the corresponding I/O list item to determine the form of the data. The translated data is then written to an external file.

In general, values transferred as output have the same forms as values transferred as input.

Table 10-1 shows the default output formats for each intrinsic data type.

Table 10-1 Default Formats for List-Directed Output

Data Type	Output Format
BYTE	I5
LOGICAL(1)	L2
LOGICAL(2)	L2
LOGICAL(4)	L2
LOGICAL(8)	L2
INTEGER(1)	I5
INTEGER(2)	I7
INTEGER(4)	I12
INTEGER(8)	I22
REAL(4)	1PG15.7E2
REAL(8) T_floating	1PG24.15E3
REAL(8) D_floating	1PG24.16E2
REAL(8) G_floating	1PG24.15E3
REAL(16)	1PG43.33E4
COMPLEX(4)	'(',1PG14.7E2,',',1PG14.7E2,')'
COMPLEX(8) T_floating	'(',1PG23.15E3,',',1PG23.15E3,')'
COMPLEX((8) D_floating	'(',1PG23.16E2,',',1PG23.16E2,')'

Table 10-1 Default Formats for List-Directed Output

Data Type	Output Format
COMPLEX((8) G_floating	'(',1PG23.15E3,',',1PG23.15E3,')'
COMPLEX(16)	'(',1PG42.33E4,',',1PG42.33E4,')'
CHARACTER	A*w*[1]

1. Where *w* is the length of the character expression.

By default, character constants are not delimited by apostrophes or quotation marks, and each internal apostrophe or quotation mark is represented externally by one apostrophe or quotation mark.

This behavior can be changed by the DELIM specifier (in an OPEN statement) as follows:

- If the file is opened with the DELIM='QUOTE' specifier, character constants are delimited by quotation marks and each internal quotation mark is represented externally by two consecutive quotation marks.
- If the file is opened with the DELIM='APOSTROPHE' specifier, character constants are delimited by apostrophes and each internal apostrophe is represented externally by two consecutive apostrophes.

Each output statement writes one or more complete records.

A literal character constant or complex constant can be longer than an entire record. For complex constants, the end of the record can occur between the comma and the imaginary part, if the imaginary part and closing right parenthesis cannot fit in the current record. For literal constants that are longer than an entire record, the constant is continued onto as many records as necessary.

Each output record begins with a blank character for carriage control.

Slashes, octal values, null values, and repeated forms of values are not output.

If the file is connected for unformatted I/O, list-directed data transfer is prohibited.

Example

Suppose the following statements are specified:

```
DIMENSION A(4)
DATA A/4*3.4/
WRITE (1,*) 'ARRAY VALUES FOLLOW'
WRITE (1,*) A,4
```

The following records are then written to external unit 1:

```
ARRAY VALUES FOLLOW
   3.400000      3.400000      3.400000      3.400000      4
```

See Also

- "Rules for Formatted Sequential WRITE Statements"
- "Rules for List-Directed Sequential READ Statements" for details on list-directed input

Rules for Namelist Sequential WRITE Statements

Namelist, sequential WRITE statements translate data from internal to external form by using the data types of the objects in the corresponding NAMELIST statement to determine the form of the data. The translated data is then written to an external file.

In general, values transferred as output have the same forms as values transferred as input.

By default, character constants are not delimited by apostrophes or quotation marks, and each internal apostrophe or quotation mark is represented externally by one apostrophe or quotation mark.

This behavior can be changed by the DELIM specifier (in an OPEN statement) as follows:

- If the file is opened with the DELIM='QUOTE' specifier, character constants are delimited by quotation marks and each internal quotation mark is represented externally by two consecutive quotation marks.
- If the file is opened with the DELIM='APOSTROPHE' specifier, character constants are delimited by apostrophes and each internal apostrophe is represented externally by two consecutive apostrophes.

Each output statement writes one or more complete records.

A literal character constant or complex constant can be longer than an entire record. In the case of complex constants, the end of the record can occur between the comma and the imaginary part, if the imaginary part and closing right parenthesis cannot fit in the current record.

Each output record begins with a blank character for carriage control, except for literal character constants that are continued from the previous record.

Slashes, octal values, null values, and repeated forms of values are not output.

If the file is connected for unformatted I/O, namelist data transfer is prohibited.

Example

Consider the following statements:

```
CHARACTER*19 NAME(2)/2*' '/
REAL PITCH, ROLL, YAW, POSITION(3)
LOGICAL DIAGNOSTICS
INTEGER ITERATIONS
NAMELIST /PARAM/ NAME, PITCH, ROLL, YAW, POSITION,        &
         DIAGNOSTICS, ITERATIONS
```

```
...
READ (UNIT=1,NML=PARAM)
WRITE (UNIT=2,NML=PARAM)
```

Suppose the following input is read:

```
&PARAM
    NAME(2)(10:)='HEISENBERG',
    PITCH=5.0, YAW=0.0, ROLL=5.0,
    DIAGNOSTICS=.TRUE.
    ITERATIONS=10
/
```

The following is then written to the file connected to unit 2:

```
&PARAM
NAME   = '                          ', '        HEISENBERG',
PITCH  =    5.000000    ,
ROLL   =    5.000000    ,
YAW    =   0.0000000E+00,
POSITION        = 3*0.0000000E+00,
DIAGNOSTICS     = T,
ITERATIONS      =                10
/
```

Note that character values are not enclosed in apostrophes unless the output file is opened with DELIM='APOSTROPHE'. The value of POSITION is not defined in the namelist input, so the current value of POSITION is written.

See Also

- "Rules for Formatted Sequential WRITE Statements"
- "Rules for Namelist Sequential READ Statements" for details on namelist input

Rules for Unformatted Sequential WRITE Statements

Unformatted, sequential WRITE statements transfer binary data (without translation) between the entities specified in the I/O list and the current record. Only one record is written.

Objects of intrinsic or derived types can be transferred.

This form of WRITE statement writes exactly one record. If there is no I/O item list, the statement writes one null record.

If the file is connected for formatted, list-directed, or namelist I/O, unformatted data transfer is prohibited.

Example

The following example shows an unformatted, sequential WRITE statement:

```
WRITE (UNIT=6, IOSTAT=IO_STATUS) A, B, C
```

Forms for Direct-Access WRITE Statements

Direct-access WRITE statements transfer output data to external records with direct access. (The attributes of a direct-access file are established by the OPEN statement.)

A direct-access WRITE statement can be formatted or unformatted, and takes one of the following forms:

Formatted:

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

Unformatted:

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

eunit

Is an external unit specifier ([UNIT=]*io-unit*).

format

Is a format specifier ([FMT=]*format*). It must not be an asterisk (*).

rec

Is a record specifier (REC=*r*).

iostat

Is a status specifier (IOSTAT=*i-var*).

err

Is a branch specifier (ERR=*label*) if an error condition occurs.

io-list

Is an I/O list.

See Also

- "I/O Control List" for details on I/O control-list specifiers
- "I/O Lists" for the general rules for I/O lists

Rules for Formatted Direct-Access WRITE Statements

Formatted, direct-access WRITE statements translate data from binary to character form by using format specifications for editing (if any). The translated data is written to an external file that is connected for direct access.

Values can be transferred from objects of intrinsic or derived types. For derived types, values of intrinsic types are transferred from the components of intrinsic types that ultimately make up these structured objects.

If the values specified by the I/O list do not fill a record, blank characters are added to fill the record. If the I/O list specifies too many characters for the record, an error occurs.

If the format specification specifies another record, the record number is increased by one as each subsequent record is written by that output statement.

Example

The following example shows a formatted, direct-access WRITE statement:

```
WRITE (2, REC=35, FMT=10) (NUM(K), K=1,10)
```

Rules for Unformatted Direct-Access WRITE Statements

Unformatted, direct-access WRITE statements transfer binary data (without translation) between the entities specified in the I/O list and the current record. Only one record is written.

Objects of intrinsic or derived types can be transferred.

If the values specified by the I/O list do not fill a record, blank characters are added to fill the record. If the I/O list specifies too many characters for the record, an error occurs.

If the file is connected for formatted, list-directed, or namelist I/O, unformatted data transfer is prohibited.

Example

The following example shows unformatted, direct-access WRITE statements:

```
WRITE (1, REC=10) LIST(1), LIST(8)
WRITE (4, REC=58, IOSTAT=K, ERR=500) (RHO(N), N=1,5)
```

Forms and Rules for Internal WRITE Statements

Internal WRITE statements transfer output data to an internal file.

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

An internal WRITE statement takes the following form:

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

iunit

Is an internal unit specifier ([UNIT=]*io-unit*). It must be a default character variable. It must not be an array section with a vector subscript.

format

Is a format specifier ([FMT=]*format*). An asterisk (*) indicates list-directed formatting.

iostat

Is a status specifier (IOSTAT=*i-var*).

err

Is a branch specifier (ERR=*label*) if an error condition occurs.

io-list

Is an I/O list.

Formatted, internal WRITE statements translate data from binary to character form by using format specifications for editing (if any). The translated data is written to an internal file.

Values can be transferred from objects of intrinsic or derived types. For derived types, values of intrinsic types are transferred from the components of intrinsic types that ultimately make up these structured objects.

f the number of characters written in a record is less than the length of the record, the rest of the record is filled with blanks. The number of characters to be written must not exceed the length of the record.

Character constants are not delimited by apostrophes or quotation marks, and each internal apostrophe or quotation mark is represented externally by one apostrophe or quotation mark.

Example

The following example shows an internal WRITE statement:

```
INTEGER J, K, STAT_VALUE
CHARACTER*50 CHAR_50
...
WRITE (FMT=*, UNIT=CHAR_50, IOSTAT=STAT_VALUE) J, K
```

See Also

- "I/O Control List" for details on I/O control-list specifiers
- "I/O Lists" for the general rules for I/O lists
- "Rules for List-Directed Sequential WRITE Statements" for details on list-directed output
- Your user's guide for details on using internal files

PRINT and TYPE Statements

The PRINT statement is a data transfer output statement. TYPE is a synonym for PRINT. All forms and rules for the PRINT statement also apply to the TYPE statement.

The PRINT statement is the same as a formatted, sequential WRITE statement, except that the PRINT statement must never transfer data to user-specified I/O units.

A PRINT statement takes one of the following forms:

Formatted:

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

Formatted - List-Directed:

PRINT * [, *io-list*]

Formatted - Namelist:

PRINT *nml*

form

Is the nonkeyword form of a format specifier (no FMT=).

io-list

Is an I/O list.

*

Is the format specifier indicating list-directed formatting.

nml

Is the nonkeyword form of a namelist specifier (no NML=) indicating namelist formatting.

Example

In the following example, one record (containing four fields of data) is printed to the implicit output device:

```
      CHARACTER*16 NAME, JOB
      PRINT 400, NAME, JOB
400   FORMAT ('NAME=', A, 'JOB=', A)
```

See Also

- "Rules for Formatted Sequential WRITE Statements"
- "I/O Lists" for the general rules for I/O lists
- "Rules for List-Directed Sequential WRITE Statements" for details on list-directed output
- "Rules for Namelist Sequential WRITE Statements" for details on namelist output

- Your user's guide for details on formatted data and data transfers

REWRITE Statement

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

A REWRITE statement can be formatted or unformatted, and takes one of the following forms:

Formatted:

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

Unformatted:

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

eunit

Is an external unit specifier ([UNIT=]*io-unit*).

format

Is a format specifier ([FMT=]*format*).

iostat

Is a status specifier (IOSTAT=*i-var*).

err

Is a branch specifier (ERR=*label*) if an error condition occurs.

io-list

Is an I/O list.

In the REWRITE statement, data (translated if formatted; untranslated if unformatted) is written to the current (existing) record in a file with direct access.

The current record is the last record accessed by a preceding, successful sequential or direct-access READ statement.

Between a READ and REWRITE statement, you should not specify any other I/O statement (except INQUIRE) on that logical unit. Execution of any other I/O statement on the logical unit destroys the current-record context and causes the current record to become undefined.

Only one record can be rewritten in a single REWRITE statement operation.

The output list (and format specification, if any) must not specify more characters for a record than the record size. (Record size is specified by RECL in an OPEN statement.)

If the number of characters specified by the I/O list (and format, if any) do not fill a record, blank characters are added to fill the record.

Example

In the following example, the current record (contained in the relative organization file connected to logical unit 3) is updated with the values represented by NAME, AGE, and BIRTH:

```
      REWRITE (3, 10, ERR=99) NAME, AGE, BIRTH
10    FORMAT (A16, I2, A8)
```

See Also

- "I/O Control List" for details on I/O control-list specifiers
- "I/O Lists" for the general rules for I/O lists
- "RECL Specifier" for details on using the specifier in OPEN statements
- Your user's guide for details on formatted data and data transfers

I/O Formatting

11

A format appearing in an input or output (I/O) statement specifies the form of data being transferred and the data conversion (editing) required to achieve that form. The format specified can be explicit or implicit.

Explicit format is indicated in a format specification that appears in a FORMAT statement or a character expression (the expression must evaluate to a valid format specification).

The format specification contains edit descriptors, which can be data edit descriptors, control edit descriptors, or string edit descriptors.

Implicit format is determined by the processor and is specified using list-directed or namelist formatting.

List-directed formatting is specified with an asterisk (*); namelist formatting is specified with a namelist group name.

List-directed formatting can be specified for advancing sequential files and internal files. Namelist formatting can be specified only for advancing sequential files.

This chapter contains information on the following topics:

- "Format Specifications"
- "Data Edit Descriptors"
- "Control Edit Descriptors"
- "Character String Edit Descriptors"
- "Nested and Group Repeat Specifications"
- "Variable Format Expressions"
- "Printing of Formatted Records"
- "Interaction Between Format Specifications and I/O Lists"

See Also

- "Rules for List-Directed Sequential READ Statements" for details on list-directed input

- "Rules for List-Directed Sequential WRITE Statements" for details on list-directed output
- "Rules for Namelist Sequential READ Statements" for details on namelist input
- "Rules for Namelist Sequential WRITE Statements" for details on namelist output

Format Specifications

A format specification can appear in a FORMAT statement or character expression. In a FORMAT statement, it is preceded by the keyword FORMAT. A format specification takes the following form:

(*format-list*)

format-list

Is a list of one or more of the following edit descriptors, separated by commas or slashes (/):

Data edit descriptors:	I, B, O, Z, F, E, EN, ES, D, G, L, and A
Control edit descriptors:	T, TL, TR, X, S, SP, SS, BN, BZ, P, :, /, $, \, and Q
String edit descriptors:	H, 'c', and "c", where *c* is a character constant

A comma can be omitted in the following cases:

- Between a P edit descriptor and an immediately following F, E, EN, ES, D, or G edit descriptor
- Before a slash (/) edit descriptor when the optional repeat specification is not present
- After a slash (/) edit descriptor
- Before or after a colon (:) edit descriptor

Edit descriptors can be nested and a *repeat specification* can precede data edit descriptors, the slash edit descriptor, or a parenthesized list of edit descriptors.

Rules and Behavior

A FORMAT statement must be labeled.

Named constants are not permitted in format specifications.

If the associated I/O statement contains an I/O list, the format specification must contain at least one data edit descriptor or the control edit descriptor Q.

Blank characters can precede the initial left parenthesis, and additional blanks can appear anywhere within the format specification. These blanks have no meaning unless they are within a character string edit descriptor.

When a formatted input statement is executed, the setting of the BLANK specifier (for the relevant logical unit) determines the interpretation of blanks within the specification. If the BN or BZ edit descriptors are specified for a formatted input statement, they supersede the default interpretation of blanks. (For more information on BLANK defaults, see "BLANK Specifier".)

For formatted input, use the comma as an external field separator. The comma terminates the input of fields (for noncharacter data types) that are shorter than the number of characters expected. It can also designate null (zero-length) fields.

The first character of a record transmitted to a line printer or terminal is typically used for carriage control; it is not printed. The first character of such a record should be a blank, 0, 1, $, +, or ASCII NUL. Any other character is treated as a blank.

A format specification cannot specify more output characters than the external record can contain. For example, a line printer record cannot contain more than 133 characters, including the carriage control character.

Table 11-1 summarizes the edit descriptors that can be used in format specifications.

Table 11-1 Summary of Edit Descriptors

Code	Form	Effect
A	A[w]	Transfers character or Hollerith values. See "Character Editing (A)".
B	Bw[.m]	Transfers binary values. See "B Editing".
BN	BN	Ignores embedded and trailing blanks in a numeric input field. See "BN Editing".
BZ	BZ	Treats embedded and trailing blanks in a numeric input field as zeros. See "BZ Editing".
D	Dw.d	Transfers real values with D exponents. See "E and D Editing".
E	Ew.d[Ee]	Transfers real values with E exponents. See "E and D Editing".
EN	ENw.d[Ee]	Transfers real values with engineering notation. See "EN Editing".
ES	ESw.d[Ee]	Transfers real values with scientific notation. See "ES Editing".
F	Fw.d	Transfers real values with no exponent. See "F Editing".
G	Gw.d[Ee]	Transfers values of all intrinsic types. See "G Editing".
H	nHch[ch...]	Transfers characters following the H edit descriptor to an output record. See "H Editing".
I	Iw[.m]	Transfers decimal integer values. See "I Editing".
L	Lw	Transfers logical values: on input, transfers characters; on output, transfers T or F. See "Logical Editing (L)".
O	Ow[.m]	Transfers octal values. See "O Editing".
P	kP	Interprets certain real numbers with a specified scale factor. See "Scale Factor Editing (P)".

Table 11-1 Summary of Edit Descriptors

Code	Form	Effect
Q	Q	Returns the number of characters remaining in an input record. See "Character Count Editing (Q)".
S	S	Reinvokes optional plus sign (+) in numeric output fields; counters the action of SP and SS. See "S Editing".
SP	SP	Writes optional plus sign (+) into numeric output fields. See "SP Editing".
SS	SS	Suppresses optional plus sign (+) in numeric output fields. See "SS Editing".
T	Tn	Tabs to specified position. See "T Editing".
TL	TLn	Tabs left the specified number of positions. See "TL Editing".
TR	TRn	Tabs right the specified number of positions. See "TR Editing".
X	nX	Skips the specified number of positions. See "X Editing".
Z	Zw[.m]	Transfers hexadecimal values. See "Z Editing".
$	$	Suppresses trailing carriage return during interactive I/O. See "Dollar Sign ($) and Backslash (\) Editing".
:	:	Terminates format control if there are no more items in the I/O list. See "Colon Editing (:)".
/	[r]/	Terminates the current record and moves to the next record. See "Slash Editing (/)".
\	\	Continues the same record; same as $. See "Dollar Sign ($) and Backslash (\) Editing".
'c'[1]	'c'	Transfers the character literal constant (between the delimiters) to an output record. See "Character Constant Editing".

1. These delimiters can also be quotation marks (").

Character Format Specifications

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

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

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

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

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

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

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

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

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

The following shows another character format specification:

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

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

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

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

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

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

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

See Also

- "Data Edit Descriptors"
- "Control Edit Descriptors"
- "Character String Edit Descriptors"
- "Nested and Group Repeat Specifications"
- "Printing of Formatted Records"

Data Edit Descriptors

A data edit descriptor causes the transfer or conversion of data to or from its internal representation.

The part of a record that is input or output and formatted with data edit descriptors (or character string edit descriptors) is called a *field*.

This section describes the forms for data edit descriptors and the individual descriptors, themselves. It also describes general rules for numeric editing and default widths for data edit descriptors.

Forms for Data Edit Descriptors

A data edit descriptor takes one of the following forms:

[*r*]*c*

[*r*]*cw*

[*r*]*cw*.*m*

[*r*]*cw*.*d*

[*r*]*cw*.*d*[E*e*]

r

Is a repeat specification. The range of *r* is 1 through 2147483647 (2**31–1). If *r* is omitted, it is assumed to be 1.

c

Is one of the following format codes: I, B, O, Z, F, E, EN, ES, D, G, L, or A.

w

Is the total number of digits in the field (the field width). If omitted, the system applies default values (see "Default Widths for Data Edit Descriptors"). The range of *w* is 1 through 2147483647 (2**31–1) on Intel® Itanium® processors; 1 through 32767 (2**15–1) on IA-32 processors. For I, B, O, Z, and F, the range can start at zero.

m

Is the minimum number of digits that must be in the field (including leading zeros). The range of m is 0 through 32767 (2**15–1) on Intel Itanium processors; 0 through 255 (2**8–1) on IA-32 processors.

d

Is the number of digits to the right of the decimal point (the significant digits). The range of *d* is 0 through 32767 (2**15–1) on Intel Itanium processors; 0 through 255 (2**8–1) on IA-32 processors.

The number of significant digits is affected if a scale factor is specified for the data edit descriptor.

E

Identifies an exponent field.

e

Is the number of digits in the exponent. The range of e is 1 through 32767 (2**15–1) on Intel Itanium processors; 1 through 255 (2**8–1) on IA-32 processors.

Rules and Behavior

Fortran 95/90 (and the previous standard) allows the field width to be omitted only for the A descriptor. However, Intel® Fortran allows the field width to be omitted for any data edit descriptor.

The *r*, *w*, *m*, *d*, and *e* must all be positive, unsigned, integer literal constants; or variable format expressions; no kind parameter can be specified. They must not be named constants.

Actual useful ranges for *r*, *w*, *m*, *d*, and *e* may be constrained by record sizes (RECL) and the file system.

The data edit descriptors have the following specific forms:

Integer:	Iw[.m], Bw[.m], Ow[.m], and Zw[.m]
Real and complex:	Fw.d, Ew.d[Ee], ENw.d[Ee], ESw.d[Ee], Dw.d, and Gw.d[Ee]
Logical:	Lw
Character:	A[w]

The *d* must be specified with F, E, D, and G field descriptors even if *d* is zero. The decimal point is also required. You must specify both *w* and *d*, or omit them both.

A repeat specification can simplify formatting. For example, the following two statements are equivalent:

```
20  FORMAT (E12.4,E12.4,E12.4,I5,I5,I5,I5)
20  FORMAT (3E12.4,4I5)
```

See Also

- "General Rules for Numeric Editing"
- "Nested and Group Repeat Specifications"

General Rules for Numeric Editing

The following rules apply to input and output data for numeric editing (data edit descriptors I, B, O, Z, F, E, EN, ES, D, and G).

Rules for Input Processing

Leading blanks in the external field are ignored. If BLANK='NULL' is in effect (or the BN edit descriptor has been specified) embedded and trailing blanks are ignored; otherwise, they are treated as zeros. An all-blank field is treated as a value of zero.

The following table shows how blanks are interpreted by default:

Type of Unit of File	Default
An explicitly OPENed unit	BLANK='NULL'
An internal file	BLANK='NULL'
A preconnected file[1]	BLANK='NULL'

1. For interactive input from preconnected files, you should explicitly specify the BN or BZ edit descriptor to ensure desired behavior.

A minus sign must precede a negative value in an external field; a plus sign is optional before a positive value.

In input records, constants can include any valid kind parameter. Named constants are not permitted.

If the data field in a record contains fewer than *w* characters, an input statement will read characters from the next data field in the record. You can prevent this by padding the short field with blanks or zeros, or by using commas to separate the input data. The comma terminates the data field, and can also be used to designate null (zero-length) fields. For more information, see "Terminating Short Fields of Input Data".

Rules for Output Processing

The field width *w* must be large enough to include any leading plus or minus sign, and any decimal point or exponent. For example, the field width for an E data edit descriptor must be large enough to contain the following:

- For positive numbers: *d*+5 or *d*+*e*+3 characters
- For negative numbers: *d*+6 or *d*+*e*+4 characters

A positive or zero value (zero is allowed for I, B, O, Z, and F descriptors) can have a plus sign, depending on which sign edit descriptor is in effect. If a value is negative, the leftmost nonblank character is a minus sign.

If the value is smaller than the field width specified, leading blanks are inserted (the value is right-justified). If the value is too large for the field width specified, the entire output field is filled with asterisks (*).

When the value of the field width is zero, the compiler selects the smallest possible positive actual field width that does not result in the field being filled with asterisks.

See Also

- "Format Specifications"
- "Forms for Data Edit Descriptors"
- Your user's guide for details on compiler options

Integer Editing

Integer editing is controlled by the I (decimal), B (binary), O (octal), and Z (hexadecimal) data edit descriptors.

I Editing

The I edit descriptor transfers decimal integer values. It takes the following form:

I*w*[.*m*]

The value of *m* (the minimum number of digits in the constant) must not exceed the value of *w* (the field width). The *m* has no effect on input, only output.

The specified I/O list item must be of type integer or logical.

The G edit descriptor can be used to edit integer data; it follows the same rules as I*w*.

Rules for Input Processing

On input, the I data edit descriptor transfers *w* characters from an external field and assigns their integer value to the corresponding I/O list item. The external field data must be an integer constant.

If the value exceeds the range of the corresponding input list item, an error occurs.

The following shows input using the I edit descriptor:

Format	Input	Value
I4	2788	2788
I3	–26	–26
I9	ΔΔΔΔΔΔ312	312

Rules for Output Processing

On output, the I data edit descriptor transfers the value of the corresponding I/O list item, right-justified, to an external field that is *w* characters long.

The field consists of zero or more blanks, followed by a sign (a plus sign is optional for positive values, a minus sign is required for negative values), followed by an unsigned integer constant with no leading zeros.

If *m* is specified, the unsigned integer constant must have at least *m* digits. If necessary, it is padded with leading zeros.

If *m* is zero, and the output list item has the value zero, the external field is filled with blanks.

The following shows output using the I edit descriptor (the symbol ^ represents a nonprinting blank character):

Format	Value	Output
I3	284	284
I4	–284	–284
I4	0	ΔΔΔ0
I5	174	ΔΔ174
I2	3274	**

Format	Value	Output
I3	–473	***
I7	29.812	An error; the decimal point is invalid
I4.0	0	ΔΔΔΔ
I4.2	1	ΔΔ 01
I4.4	1	00001

See Also

- "Forms for Data Edit Descriptors"
- "General Rules for Numeric Editing"

B Editing

The B data edit descriptor transfers binary (base 2) values. It takes the following form:

B*w*[.*m*]

The value of *m* (the minimum number of digits in the constant) must not exceed the value of *w* (the field width). The *m* has no effect on input, only output.

The specified I/O list item can be of type integer, real, or logical.

Rules for Input Processing

On input, the B data edit descriptor transfers *w* characters from an external field and assigns their binary value to the corresponding I/O list item. The external field must contain only binary digits (0 or 1) or blanks.

If the value exceeds the range of the corresponding input list item, an error occurs.

The following shows input using the B edit descriptor:

Format	Input	Value
B4	1001	9
B1	1	1
B2	0	0

Rules for Output Processing

On output, the B data edit descriptor transfers the binary value of the corresponding I/O list item, right-justified, to an external field that is *w* characters long.

The field consists of zero or more blanks, followed by an unsigned integer constant (consisting of binary digits) with no leading zeros. A negative value is transferred in internal form.

If *m* is specified, the unsigned integer constant must have at least *m* digits. If necessary, it is padded with leading zeros.

If *m* is zero, and the output list item has the value zero, the external field is filled with blanks.

The following shows output using the B edit descriptor:

Format	Value	Output
B4	9	1001
B2	0	Δ0

See Also

- "Forms for Data Edit Descriptors"
- "General Rules for Numeric Editing"

O Editing

The O data edit descriptor transfers octal (base 8) values. It takes the following form:

O*w*[.*m*]

The value of *m* (the minimum number of digits in the constant) must not exceed the value of *w* (the field width). The *m* has no effect on input, only output.

The specified I/O list item can be of type integer, real, or logical.

Rules for Input Processing

On input, the O data edit descriptor transfers *w* characters from an external field and assigns their octal value to the corresponding I/O list item. The external field must contain only octal digits (0 through 7) or blanks.

If the value exceeds the range of the corresponding input list item, an error occurs.

The following shows input using the O edit descriptor:

Format	Input	Value
O5	32767	32767
O4	16234	1623
O3	97Δ	An error; the 9 is invalid in octal notation

Rules for Output Processing

On output, the O data edit descriptor transfers the octal value of the corresponding I/O list item, right-justified, to an external field that is *w* characters long.

The field consists of zero or more blanks, followed by an unsigned integer constant (consisting of octal digits) with no leading zeros. A negative value is transferred in internal form without a leading minus sign.

If *m* is specified, the unsigned integer constant must have at least *m* digits. If necessary, it is padded with leading zeros.

If *m* is zero, and the output list item has the value zero, the external field is filled with blanks.

The following shows output using the O edit descriptor:

Format	Value	Output
O6	32767	Δ 77777
O12	–32767	Δ 37777700001
O2	14261	**
O4	27	ΔΔ 33
O5	10.5	41050
O4.2	7	ΔΔ 07
O4.4	7	0007

See Also

- "Forms for Data Edit Descriptors"
- "General Rules for Numeric Editing"

Z Editing

The Z data edit descriptor transfers hexadecimal (base 16) values. It takes the following form:

Z*w*[.*m*]

The value of *m* (the minimum number of digits in the constant) must not exceed the value of *w* (the field width). The *m* has no effect on input, only output.

The specified I/O list item can be of type integer, real, or logical.

Rules for Input Processing

On input, the Z data edit descriptor transfers *w* characters from an external field and assigns their hexadecimal value to the corresponding I/O list item. The external field must contain only hexadecimal digits (0 though 9 and A (a) through F(f)) or blanks.

If the value exceeds the range of the corresponding input list item, an error occurs.

The following shows input using the Z edit descriptor:

Format	Input	Value
Z3	A94	A94
Z5	A23DEF	A23DE
Z5	95.AF2	An error; the decimal point is invalid

Rules for Output Processing

On output, the Z data edit descriptor transfers the hexadecimal value of the corresponding I/O list item, right-justified, to an external field that is *w* characters long.

The field consists of zero or more blanks, followed by an unsigned integer constant (consisting of hexadecimal digits) with no leading zeros. A negative value is transferred in internal form without a leading minus sign.

If *m* is specified, the unsigned integer constant must have at least *m* digits. If necessary, it is padded with leading zeros.

If *m* is zero, and the output list item has the value zero, the external field is filled with blanks.

The following shows output using the Z edit descriptor:

Format	Value	Output
Z4	32767	7FFF
Z9	–32767	∆FFFF8001
Z2	16	10
Z4	–10.5	****
Z3.3	2708	A94
Z6.4	2708	∆∆0A94

See Also

- "Forms for Data Edit Descriptors"
- "General Rules for Numeric Editing"

Real and Complex Editing

Real and complex editing is controlled by the F, E, D, EN, ES, and G data edit descriptors.

If no field width (*w*) is specified for a real data edit descriptor, the system supplies default values.

Real data edit descriptors can be affected by specified scale factors.

NOTE. *Do not use the real data edit descriptors when attempting to parse textual input. These descriptors accept some forms that are purely textual as valid numeric input values. For example, input values T and F are treated as values –1.0 and 0.0, respectively, for* .TRUE. *and* .FALSE..

See Also

- "Scale Factor Editing (P)"
- "Forms for Data Edit Descriptors"
- "General Rules for Numeric Editing"
- "Default Widths for Data Edit Descriptors" for details on system default values for data edit descriptors

F Editing

The F data edit descriptor transfers real values. It takes the following form:

F*w*.*d*

The value of *d* (the number of places after the decimal point) must not exceed the value of *w* (the field width).

The specified I/O list item must be of type real, or it must be the real or imaginary part of a complex type.

Rules for Input Processing

On input, the F data edit descriptor transfers *w* characters from an external field and assigns their real value to the corresponding I/O list item. The external field data must be an integer or real constant.

If the input field contains only an exponent letter or decimal point, it is treated as a zero value.

If the input field does not contain a decimal point or an exponent, it is treated as a real number of *w* digits, with *d* digits to the right of the decimal point. (Leading zeros are added, if necessary.)

If the input field contains a decimal point, the location of that decimal point overrides the location specified by the F descriptor.

If the field contains an exponent, that exponent is used to establish the magnitude of the value before it is assigned to the list element.

The following shows input using the F edit descriptor:

Format	Input	Value
F8.5	123456789	123.45678
F8.5	–1234.567	–1234.56
F8.5	24.77E+2	2477.0
F5.2	1234567.89	123.45

Rules for Output Processing

On output, the F data edit descriptor transfers the real value of the corresponding I/O list item, right-justified and rounded to *d* decimal positions, to an external field that is *w* characters long.

The *w* must be greater than or equal to *d*+3 to allow for the following:

- A sign (optional if the value is positive and descriptor SP is not in effect)
- At least one digit to the left of the decimal point
- The decimal point
- The *d* digits to the right of the decimal point

The following shows output using the F edit descriptor:

Format	Value	Output
F8.5	2.3547188	Δ 2.35472
F9.3	8789.7361	Δ 8789.736
F2.1	51.44	**
F10.4	–23.24352	ΔΔ –23.2435
F5.2	325.013	******
F5.2	–.2	–0.20

See Also

- "Forms for Data Edit Descriptors"
- "General Rules for Numeric Editing"

E and D Editing

The E and D data edit descriptors transfer real values in exponential form. They take the following form:

E*w*.*d*[E*e*]

D*w*.*d*

For the E edit descriptor, the value of *d* (the number of places after the decimal point) plus *e* (the number of digits in the exponent) must not exceed the value of *w* (the field width).

For the D edit descriptor, the value of *d* must not exceed the value of *w*.

The specified I/O list item must be of type real, or it must be the real or imaginary part of a complex type.

Rules for Input Processing

On input, the E and D data edit descriptors transfer *w* characters from an external field and assigns their real value to the corresponding I/O list item. The E and D descriptors interpret and assign input data in the same way as the F data edit descriptor (see "F Editing").

The following shows input using the E and D edit descriptors:

Format	Input	Value
E9.3	734.432E3	734432.0
E12.4	ΔΔ1022.43E–6	1022.43E–6
E15.3	52.3759663ΔΔΔΔΔ	52.3759663
E12.5	210.5271D+10[1]	210.5271E10
BZ,D10.2	12345ΔΔΔΔΔ	12345000.0D0
D10.2	ΔΔ123.45ΔΔ	123.45D0
D15.3	367.4981763D+04	3.674981763D+06

1. If the I/O list item is single-precision real, the E edit descriptor treats the D exponent indicator as an E indicator.

Rules for Output Processing

On output, the E and D data edit descriptors transfer the real value of the corresponding I/O list item, right-justified and rounded to *d* decimal positions, to an external field that is *w* characters long.

The *w* should be greater than or equal to *d*+7 to allow for the following:

- A sign (optional if the value is positive and descriptor SP is not in effect)
- An optional zero to the left of the decimal point
- The decimal point
- The *d* digits to the right of the decimal point
- The exponent

The exponent takes one of the following forms:

Edit Descriptor	Absolute Value of Exponent	Positive Form of Exponent	Negative Form of Exponents
Ew.d	$\lvert exp \rvert \le 99$	E+nn	E–nn
	$99 < \lvert exp \rvert \le 999$	+nnn	–nnn
Ew.dEe	$\lvert exp \rvert \le 10^e - 1$	$E+n_1n_2...n_e$	$E–n_1n_2...n_e$
Dw.d	$\lvert exp \rvert \le 99$	D+nn or E+nn	D–nn or E–nn
	$99 < \lvert exp \rvert \le 999$	+nnn	–nnn

If the exponent value is too large to be converted into one of these forms, an error occurs.

The exponent field width (*e*) is optional for the E edit descriptor; if omitted, the default value is 2. If *e* is specified, the *w* should be greater than or equal to *d*+*e*+5.

NOTE. *The w can be as small as d+5 or d+e+3, if the optional fields for the sign and the zero are omitted.*

The following shows output using the E and D edit descriptors:

Format	Value	Output
E11.2	475867.222	ΔΔΔ 0.48E+06
E11.5	475867.222	0.47587E+06
E12.3	0.00069	ΔΔΔ 0.690E–03
E10.3	–0.5555	–0.556E+00
E5.3	56.12	*****
E14.5E4	–1.001	–1.10010E+0001
E13.3E6	0.000123	0.123E–000003
D14.3	0.0363	ΔΔΔΔΔ 0.363D–1
D23.12	5413.87625793	ΔΔΔΔΔ 0.541387625793D+04
D9.6	1.2	*********

See Also

- "Forms for Data Edit Descriptors"
- "General Rules for Numeric Editing"
- "Scale Factor Editing (P)"

EN Editing

The EN data edit descriptor transfers values by using engineering notation. It takes the following form:

EN*w*.*d*[E*e*]

The value of *d* (the number of places after the decimal point) plus *e* (the number of digits in the exponent) must not exceed the value of *w* (the field width).

The specified I/O list item must be of type real, or it must be the real or imaginary part of a complex type.

Rules for Input Processing

On input, the EN data edit descriptor transfers *w* characters from an external field and assigns their real value to the corresponding I/O list item. The EN descriptor interprets and assigns input data in the same way as the F data edit descriptor (see "F Editing").

The following shows input using the EN edit descriptor:

Format	Input	Value
EN11.3	ΔΔ 5.321E+00	5.32100
EN11.3	–600.00E–03	–.60000
EN12.3	ΔΔΔ 3.150E–03	.00315
EN12.3	ΔΔΔ 3.829E+03	3829.0

Rules for Output Processing

On output, the EN data edit descriptor transfers the real value of the corresponding I/O list item, right-justified and rounded to *d* decimal positions, to an external field that is *w* characters long. The real value is output in engineering notation, where the decimal exponent is divisible by 3 and the absolute value of the significand is greater than or equal to 1 and less than 1000 (unless the output value is zero).

The *w* should be greater than or equal to *d*+9 to allow for the following:

- A sign (optional if the value is positive and descriptor SP is not in effect)
- One to three digits to the left of the decimal point
- The decimal point
- The *d* digits to the right of the decimal point
- The exponent

The exponent takes one of the following forms:

Edit Descriptor	Absolute Value of Exponent	Positive Form of Exponent	Negative Form of Exponents
ENw.d	$\lvert exp \rvert \le 99$	E+nn	E–nn
	$99 < \lvert exp \rvert \le 999$	+nnn	–nnn
ENw.dEe	$\lvert exp \rvert \le 10^e - 1$	$E+n_1n_2...n_e$	$E–n_1n_2...n_e$

If the exponent value is too large to be converted into one of these forms, an error occurs.

The exponent field width (*e*) is optional; if omitted, the default value is 2. If *e* is specified, the *w* should be greater than or equal to *d*+*e*+5.

The following shows output using the EN edit descriptor:

Format	Value	Output
EN11.2	475867.222	Δ475.87E+03
EN11.5	475867.222	**********
EN12.3	0.00069	Δ690.000E–06
EN10.3	–0.5555	**********
EN11.2	0.0	Δ000.00E–03

See Also

- "Forms for Data Edit Descriptors"
- "General Rules for Numeric Editing"

ES Editing

The ES data edit descriptor transfers values by using scientific notation. It takes the following form:

ES*w*.*d*[E*e*]

The value of *d* (the number of places after the decimal point) plus *e* (the number of digits in the exponent) must not exceed the value of *w* (the field width).

The specified I/O list item must be of type real, or it must be the real or imaginary part of a complex type.

Rules for Input Processing

On input, the ES data edit descriptor transfers *w* characters from an external field and assigns their real value to the corresponding I/O list item. The ES descriptor interprets and assigns input data in the same way as the F data edit descriptor (see "F Editing").

The following shows input using the ES edit descriptor:

Format	Input	Value
ES11.3	ΔΔ 5.321E+00	5.32100
ES11.3	–6.000E–03	–.60000
ES12.3	ΔΔΔ 3.150E–03	.00315
ES12.3	ΔΔΔ 3.829E+03	3829.0

Rules for Output Processing

On output, the ES data edit descriptor transfers the real value of the corresponding I/O list item, right-justified and rounded to *d* decimal positions, to an external field that is *w* characters long. The real value is output in scientific notation, where the absolute value of the significand is greater than or equal to 1 and less than 10 (unless the output value is zero).

The *w* should be greater than or equal to *d*+7 to allow for the following:

- A sign (optional if the value is positive and descriptor SP is not in effect)
- One digit to the left of the decimal point
- The decimal point
- The *d* digits to the right of the decimal point
- The exponent

The exponent takes one of the following forms:

Edit Descriptor	Absolute Value of Exponent	Positive Form of Exponent	Negative Form of Exponents
ESw.d	$\lvert exp \rvert \le 99$	E+nn	E–nn
	$99 < \lvert exp \rvert \le 999$	+nnn	–nnn
ESw.dEe	$\lvert exp \rvert \le 10^e - 1$	$E+n_1n_2...n_e$	$E–n_1n_2...n_e$

If the exponent value is too large to be converted into one of these forms, an error occurs.

The exponent field width (*e*) is optional; if omitted, the default value is 2. If *e* is specified, the *w* should be greater than or equal to *d*+*e*+5.

The following shows output using the ES edit descriptor:

Format	Value	Output
ES11.2	473214.356	ΔΔΔ 4.73E+05
ES11.5	473214.356	4.73214E+05
ES12.3	0.00069	ΔΔΔ 6.900E–04
ES10.3	–0.5555	–5.555E–01
ES11.2	0.0	Δ 0.000E+00

See Also

- "Forms for Data Edit Descriptors"
- "General Rules for Numeric Editing"

G Editing

The G data edit descriptor generally transfers values of real type, but it can be used to transfer values of any intrinsic type. It takes the following form:

G*w*.*d*[E*e*]

The value of *d* (the number of places after the decimal point) plus *e* (the number of digits in the exponent) must not exceed the value of *w* (the field width).

The specified I/O list item can be of any intrinsic type.

When used to specify I/O for integer, logical, or character data, the edit descriptor follows the same rules as I*w*, L*w*, and A*w*, respectively, and *d* and *e* have no effect.

Rules for Real Input Processing

On input, the G data edit descriptor transfers *w* characters from an external field and assigns their real value to the corresponding I/O list item. The G descriptor interprets and assigns input data in the same way as the F data edit descriptor (see "F Editing").

Rules for Real Output Processing

On output, the G data edit descriptor transfers the real value of the corresponding I/O list item, right-justified and rounded to *d* decimal positions, to an external field that is *w* characters long.

The form in which the value is written is a function of the magnitude of the value, as described in Table 11-2.

Table 11-2 Effect of Data Magnitude on G Format Conversions

Data Magnitude	Equivalent Conversion
$0 < m < 0.1 - 0.5 \times 10^{-d-1}$	Ew.d[Ee]
$m = 0$	F(w–n).(d–1), n('b')
$0.1 - 0.5 \times 10^{-d-1} \le m < 1 - 0.5 \times 10^{-d}$	F(w–n).d, n('b')
$1 - 0.5 \times 10^{-d} \le m < 10 - 0.5 \times 10^{-d+1}$	F(w–n).(d–1), n('b')
$10 - 0.5 \times 10^{-d+1} \le m < 100 - 0.5 \times 10^{-d+2}$	F(w–n).(d–2), n('b')
.	.
.	.
.	.
$10^{d-2} - 0.5 \times 10^{-2} \le m < 10^{d-1} - 0.5 \times 10^{-1}$	F(w–n).1, n('b')
$10^{d-1} - 0.5 \times 10^{-1} \le m < 10^{d} - 0.5$	F(w–n).0, n('b')
$m \ge 10^{d} - 0.5$	Ew.d[Ee]

The 'b' is a blank following the numeric data representation. For Gw.d, n('b') is 4 blanks. For Gw.dEe, n('b') is *e*+2 blanks.

The *w* should be greater than or equal to *d*+7 to allow for the following:

- A sign (optional if the value is positive and descriptor SP is not in effect)
- One digit to the left of the decimal point
- The decimal point
- The *d* digits to the right of the decimal point
- The 4-digit or *e*+2-digit exponent

If *e* is specified, the *w* should be greater than or equal to *d*+*e*+5.

The following shows output using the G edit descriptor and compares it to output using equivalent F editing:

Value	Format	Output with G	Format	Output with F
0.01234567	G13.6	Δ 0.123457E–01	F13.6	ΔΔΔΔΔ 0.012346
–0.12345678	G13.6	–0.123457ΔΔΔΔ	F13.6	ΔΔΔΔ –0.123457
1.23456789	G13.6	ΔΔ 1.23457ΔΔΔΔ	F13.6	ΔΔΔΔΔ 1.234568
12.34567890	G13.6	ΔΔ 12.3457ΔΔΔΔ	F13.6	ΔΔΔΔ 12.345679
123.45678901	G13.6	ΔΔ 123.457ΔΔΔΔ	F13.6	ΔΔΔ 123.456789
–1234.56789012	G13.6	Δ –1234.57ΔΔΔΔ	F13.6	Δ –1234.567890

Value	Format	Output with G	Format	Output with F
12345.67890123	G13.6	ΔΔ 12345.7ΔΔΔΔ	F13.6	Δ 12345.678901
123456.78901234	G13.6	ΔΔ 123457.ΔΔΔΔ	F13.6	123456.789012
–1234567.89012345	G13.6	–0.123457E+07	F13.6	**********

See Also

- "Forms for Data Edit Descriptors"
- "General Rules for Numeric Editing"
- "Scale Factor Editing (P)"
- "I Editing" for details on the I data edit descriptor
- "Logical Editing (L)" for details on the L data edit descriptor
- "Character Editing (A)" for details on the A data edit descriptor

Complex Editing

A complex value is an ordered pair of real values. Complex editing is specified by a pair of real edit descriptors, using any combination of the forms: Fw.d, Ew.d[Ee], Dw.d, ENw.d[Ee], ESw.d[Ee], or Gw.d[Ee].

Rules for Input Processing

On input, the two successive fields are read and assigned to the corresponding complex I/O list item as its real and imaginary part, respectively.

The following shows input using complex editing:

Format	Input	Value
F8.5,F8.5	1234567812345.67	123.45678, 12345.67
E9.1,F9.3	734.432E8123456789	734.432E8, 123456.789

Rules for Output Processing

On output, the two parts of the complex value are transferred under the control of repeated or successive real edit descriptors. The two parts are transferred consecutively without punctuation or blanks, unless control or character string edit descriptors are specified between the pair of real edit descriptors.

The following shows output using complex editing:

Format	Value	Output
2F8.5	2.3547188, 3.456732	Δ 2.35472 Δ 3.45673

Format	Value	Output
E9.2,'ΔΔ',E5.3	47587.222, 56.123	Δ0.48E+06Δ,Δ*****

See Also

- "Forms for Data Edit Descriptors"
- "General Rules for Numeric Editing"
- "General Rules for Complex Constants"

Logical Editing (L)

The L data edit descriptor transfers logical values. It takes the following form:

L*w*

The specified I/O list item must be of type logical or integer.

The G edit descriptor can be used to edit logical data; it follows the same rules as L*w*.

Rules for Input Processing

On input, the L data edit descriptor transfers *w* characters from an external field and assigns their logical value to the corresponding I/O list item. The value assigned depends on the external field data, as follows:

- .TRUE. is assigned if the first nonblank character is .T, T, .t, or t. The logical constant .TRUE. is an acceptable input form.
- .FALSE. is assigned if the first nonblank character is .F, F. .f, or f, or the entire field is filled with blanks. The logical constant .FALSE. is an acceptable input form.

If an other value appears in the external field, an error occurs.

Rules for Output Processing

On output, the L data edit descriptor transfers the following to an external field that is *w* characters long: w – 1 blanks, followed by a T or F (if the value is .TRUE. or .FALSE., respectively).

The following shows output using the L edit descriptor:

Format	Value	Output
L5	.TRUE.	ΔΔΔΔT
L1	.FALSE.	F

See Also

"Forms for Data Edit Descriptors"

Character Editing (A)

The A data edit descriptor transfers character or Hollerith values. It takes the following form:

A[*w*]

If the corresponding I/O list item is of type character, character data is transferred. If the list item is of any other type, Hollerith data is transferred.

The G edit descriptor can be used to edit character data; it follows the same rules as A*w*.

Rules for Input Processing

On input, the A data edit descriptor transfers *w* characters from an external field and assigns them to the corresponding I/O list item.

The maximum number of characters that can be stored depends on the size of the I/O list item, as follows:

- For character data, the maximum size is the length of the corresponding I/O list item.
- For noncharacter data, the maximum size depends on the data type, as shown in Table 11-3.

Table 11-3 Size Limits for Noncharacter Data Using A Editing

I/O List Element	Maximum Number of Character
BYTE	1
LOGICAL(1) or LOGICAL*1	1
LOGICAL(2) or LOGICAL*2	2
LOGICAL(4) or LOGICAL*4	4
LOGICAL(8) or LOGICAL*8	8
INTEGER(1) or INTEGER*1	1
INTEGER(2) or INTEGER*2	2
INTEGER(4) or INTEGER*4	4
INTEGER(8) or INTEGER*8	8
REAL(4) or REAL*4	4
DOUBLE PRECISION	8
REAL(8) or REAL*8	8
REAL(16) or REAL*16	16
COMPLEX(4) or COMPLEX*8[1]	8
DOUBLE COMPLEX[1]	16
COMPLEX(8) or COMPLEX*16[1]	16

Table 11-3 Size Limits for Noncharacter Data Using A Editing

I/O List Element	Maximum Number of Character
COMPLEX(16) or COMPLEX*32[1]	32

1. Complex values are treated as pairs of real numbers, so complex editing requires a pair of edit descriptors. (See "Complex Editing".)

If *w* is equal to or greater than the length (*len*) of the input item, the rightmost characters are assigned to that item. The leftmost excess characters are ignored.

If *w* is less than *len*, or less than the number of characters that can be stored, *w* characters are assigned to the list item, left-justified, and followed by trailing blanks.

The following shows input using the A edit descriptor:

Format	Input	Value	Data Type
A6	PAGE∆#	#	CHARACTER(LEN=1)
A6	PAGE∆#	E∆#	CHARACTER(LEN=3)
A6	PAGE∆#	PAGE∆#	CHARACTER(LEN=6)
A6	PAGE∆#	PAGE∆#∆∆	CHARACTER(LEN=8)
A6	PAGE∆#	#	LOGICAL(1)
A6	PAGE∆#	∆#	INTEGER(2)
A6	PAGE∆#	GE∆#	REAL(4)
A6	PAGE∆#	PAGE∆#∆∆	REAL(8)

Rules for Output Processing

On output, the A data edit descriptor transfers the contents of the corresponding I/O list item to an external field that is *w* characters long.

If *w* is greater than the size of the list item, the data is transferred to the output field, right-justified, with leading blanks. If *w* is less than or equal to the size of the list item, the leftmost *w* characters are transferred.

The following shows output using the A edit descriptor:

Format	Value	Output
A5	OHMS	∆OHMS
A5	VOLTS	VOLTS
A5	AMPERES	AMPER

Default Widths for Data Edit Descriptors

If *w* (the field width) is omitted for the data edit descriptors, the system applies default values. For the real data edit descriptors, the system also applies default values for *d* (the number of characters to the right of the decimal point), and *e* (the number of characters in the exponent).

These defaults are based on the data type of the I/O list item, and are listed in Table 11-4.

Table 11-4 Default Widths for Data Edit Descriptors

Edit Descriptor	Data Type of I/O List Item	w:
I, B, O, Z, G	BYTE	7
	INTEGER(1), LOGICAL(1)	7
	INTEGER(2), LOGICAL(2)	7
	INTEGER(4), LOGICAL(4)	12
	INTEGER(8), LOGICAL(8)	23
O, Z	REAL(4)	12
	REAL(8)	23
	REAL(16)	44
	CHARACTER*len	MAX(7, 3*len)
L, G	LOGICAL(1), LOGICAL(2)	2
	LOGICAL(4), LOGICAL(8)	2
F, E, EN, ES, G, D	REAL(4), COMPLEX(4)	15 **d**: 7 **e**: 2
	REAL(8), COMPLEX(8)	25 **d**: 16 **e**: 2
	REAL(16), COMPLEX(16)	42 **d**: 33 **e**: 3
A[1], G	LOGICAL(1)	1
	LOGICAL(2), INTEGER(2)	2
	LOGICAL(4), INTEGER(4)	4
	LOGICAL(8), INTEGER(8)	8
	REAL(4), COMPLEX(4)	4
	REAL(8), COMPLEX(8)	8
	REAL(16), COMPLEX(16)	16
	CHARACTER*len	len

1. The default is the actual length of the corresponding I/O list item.

Terminating Short Fields of Input Data

On input, an edit descriptor such as Fw.d specifies that *w* characters (the field width) are to be read from the external field.

If the field contains fewer than *w* characters, the input statement will read characters from the next data field in the record. You can prevent this by padding the short field with blanks or zeros, or by using commas to separate the input data.

Padding Short Fields

You can use the OPEN statement specifier PAD='YES' to indicate blank padding for short fields of input data. However, blanks can be interpreted as blanks *or* zeros, depending on which default behavior is in effect at the time. Consider the following:

```
READ (2, '(I5)') J
```

If 3 is input for J, the value of J will be 30000 or 3 depending on which default behavior is in effect (BLANK='NULL' or BLANK='ZERO'). This can give unexpected results.

To ensure that the desired behavior is in effect, explicitly specify the BN or BZ edit descriptor. For example, the following ensures that blanks are interpreted as blanks (and not as zeros):

```
READ (2, '(BN, I5)') J
```

Using Commas to Separate Input Data

You can use a comma to terminate a short data field. The comma has no effect on the *d* part (the number of characters to the right of the decimal point) of the specification.

The comma overrides the *w* specified for the I, B, O, Z, F, E, D, EN, ES, G, and L edit descriptors. For example, suppose the following statements are executed:

```
     READ (5,100) I,J,A,B
100  FORMAT (2I6,2F10.2)
```

Suppose a record containing the following values is read:

```
1, -2, 1.0, 35
```

The following assignments occur:

```
I = 1
J = -2
A = 1.0
B = 0.35
```

A comma can only terminate fields less than *w* characters long. If a comma follows a field of *w* or more characters, the comma is considered part of the next field.

A null (zero-length) field is designated by two successive commas, or by a comma after a field of *w* characters. Depending on the field descriptor specified, the resulting value assigned is 0, 0.0, 0.0D0, 0.0Q0, or .FALSE.

See Also

"General Rules for Numeric Editing" for details on input processing

Control Edit Descriptors

A control edit descriptor either directly determines how text is displayed or affects the conversions performed by subsequent data edit descriptors.

This section describes the forms for control edit descriptors and the individual descriptors themselves.

Forms for Control Edit Descriptors

A control edit descriptor takes one of the following forms:

c
cn
nc

c

Is one of the following format codes: T, TL, TR, X, S, SP, SS, BN, BZ, P, :, /, \, $, and Q.

n

Is a number of character positions. It must be a positive integer literal constant or a variable format expression. No kind parameter can be specified. It cannot be a named constant.

The range of n is 1 through 2147483647 (2**31–1) on Intel Itanium processors; 1 through 32767 (2**15–1) on IA-32 processors. Actual useful ranges may be constrained by record sizes (RECL) and the file system.

Rules and Behavior

In general, control edit descriptors are nonrepeatable. The only exception is the slash (/) edit descriptor, which can be preceded by a repeat specification.

The control edit descriptors have the following specific forms:

Positional:	Tn, TLn, TRn, and nX
Sign:	S, SP, and SS

Blank interpretation:	BN and BZ
Scale factor:	kP
Miscellaneous:	:, /, \, $, and Q

The P edit descriptor is an exception to the general control edit descriptor syntax. It is preceded by a scale factor, rather than a character position specifier.

Control edit descriptors can be grouped in parentheses and preceded by a group repeat specification.

See Also

- "Format Specifications"
- "Nested and Group Repeat Specifications"

Positional Editing

The T, TL, TR, and X edit descriptors specify the position where the next character is transferred to or from a record.

On output, these descriptors do not themselves cause characters to be transferred and do not affect the length of the record. If characters are transferred to positions at or after the position specified by one of these descriptors, positions skipped and not previously filled are filled with blanks. The result is as if the entire record was initially filled with blanks.

The TR and X edit descriptors produce the same results.

T Editing

The T edit descriptor specifies a character position in an I/O record. It takes the following form:

T*n*

The *n* is a positive integer literal constant (with no kind parameter) indicating the character position of the record, relative to the left tab limit.

On input, the T descriptor positions the external record at the character position specified by *n*. On output, the T descriptor indicates that data transfer begins at the *n*th character position of the external record.

Examples

Suppose a file has a record containing the value `ABC∆∆∆XYZ`, and the following statements are executed:

```
     READ (11,10) VALUE1, VALUE2
10   FORMAT (T7,A3,T1,A3)
```

The values read first are XYZ, then ABC.

Suppose the following statements are executed:

```
      PRINT 25
25    FORMAT (T51,'COLUMN 2',T21,'COLUMN 1')
```

The following line is printed at the positions indicated:

```
Position 20                     Position 50
   |                               |
   COLUMN 1                        COLUMN 2
```

Note that the first character of the record printed was reserved as a control character. (For more information, see "Printing of Formatted Records".)

TL Editing

The TL edit descriptor specifies a character position to the *left* of the current position in an I/O record. It takes the following form:

TL*n*

The *n* is a positive integer literal constant (with no kind parameter) indicating the *n*th character position to the left of the current character.

If *n* is greater than or equal to the current position, the next character accessed is the first character of the record.

TR Editing

The TR edit descriptor specifies a character position to the *right* of the current position in an I/O record. It takes the following form:

TR*n*

The *n* is a positive integer literal constant (with no kind parameter) indicating the *n*th character position to the right of the current character.

X Editing

The X edit descriptor specifies a character position to the right of the current position in an I/O record. It takes the following form:

*n*X

The *n* is a positive integer literal constant (with no kind parameter) indicating the *n*th character position to the right of the current character.

On output, the X edit descriptor does not output any characters when it appears at the end of a format specification; for example:

```
      WRITE (6,99) K
99    FORMAT ('∆K=',I6,5X)
```

This example writes a record of only 9 characters. To cause *n* trailing blanks to be output at the end of a record, specify a format of `n('∆')`.

Sign Editing

The S, SP, and SS edit descriptors control the output of the optional plus (+) sign within numeric output fields. These descriptors have no effect during execution of input statements.

Within a format specification, a sign editing descriptor affects all subsequent I, F, E, EN, ES, D, and G descriptors until another sign editing descriptor occurs.

SP Editing

The SP edit descriptor causes the processor to *produce* a plus sign in any subsequent position where it would be otherwise optional. It takes the following form:

SP

SS Editing

The SS edit descriptor causes the processor to *suppress* a plus sign in any subsequent position where it would be otherwise optional. It takes the following form:

SS

S Editing

The S edit descriptor restores the plus sign as optional for all subsequent positive numeric fields. It takes the following form:

S

The S edit descriptor restores to the processor the discretion of producing plus characters on an optional basis.

Blank Editing

The BN and BZ descriptors control the interpretation of embedded and trailing blanks within numeric input fields. These descriptors have no effect during execution of output statements.

Within a format specification, a blank editing descriptor affects all subsequent I, B, O, Z, F, E, EN, ES, D, and G descriptors until another blank editing descriptor occurs.

The blank editing descriptors override the effect of the BLANK specifier during execution of a particular input data transfer statement. (For more information on the BLANK specifier in OPEN statements, see "BLANK Specifier".)

BN Editing

The BN edit descriptor causes the processor to *ignore* all embedded and trailing blanks in numeric input fields. It takes the following form:

BN

The input field is treated as if all blanks have been removed and the remainder of the field is right-justified. An all-blank field is treated as zero.

BZ Editing

The BZ edit descriptor causes the processor to *interpret* all embedded and trailing blanks in numeric input fields as zeros. It takes the following form:

BZ

Scale Factor Editing (P)

The P edit descriptor specifies a scale factor, which moves the location of the decimal point in real values and the two real parts of complex values. It takes the following form:

*k*P

The *k* is a signed (sign is optional if positive), integer literal constant specifying the number of positions, to the left or right, that the decimal point is to move (the scale factor). The range of *k* is –128 to 127.

At the beginning of a formatted I/O statement, the value of the scale factor is zero. If a scale editing descriptor is specified, the scale factor is set to the new value, which affects all subsequent real edit descriptors until another scale editing descriptor occurs.

To reinstate a scale factor of zero, you must explicitly specify 0P.

Format reversion does not affect the scale factor. (For more information on format reversion, see "Interaction Between Format Specifications and I/O Lists".)

Rules for Input Processing

On input, a positive scale factor moves the decimal point to the left, and a negative scale factor moves the decimal point to the right. (On output, the effect is the reverse.)

On input, when an input field using an F, E, D, EN, ES, or G real edit descriptor contains an explicit exponent, the scale factor has no effect. Otherwise, the internal value of the corresponding I/O list item is equal to the external field data multiplied by 10^{-k}. For example, a 2P scale factor multiplies an input value by .01, moving the decimal point two places to the left. A –2P scale factor multiplies an input value by 100, moving the decimal point two places to the right.

The following shows input using the P edit descriptor:

Format	Input	Value
3PE10.5	ΔΔΔ 37.614Δ	.037614
3PE10.5	ΔΔ 37.614E2	3761.4
–3PE10.5	ΔΔΔΔ 37.614	37614.0

The scale factor must precede the first real edit descriptor associated with it, but it need not immediately precede the descriptor. For example, the following all have the same effect:

```
(3P, I6, F6.3, E8.1)
(I6, 3P, F6.3, E8.1)
(I6, 3PF6.3, E8.1)
```

Note that if the scale factor immediately precedes the associated real edit descriptor, the comma separator is optional.

Rules for Output Processing

On output, a positive scale factor moves the decimal point to the right, and a negative scale factor moves the decimal point to the left. (On input, the effect is the reverse.)

On output, the effect of the scale factor depends on which kind of real editing is associated with it, as follows:

- For F editing, the external value equals the internal value of the I/O list item multiplied by 10^k. This changes the magnitude of the data.
- For E and D editing, the external decimal field of the I/O list item is multiplied by 10^k, and *k* is subtracted from the exponent. This changes the form of the data.

  A positive scale factor decreases the exponent; a negative scale factor increases the exponent.

  For a positive scale factor, *k* must be less than *d* + 2 or an output conversion error occurs.
- For G editing, the scale factor has no effect if the magnitude of the data to be output is within the effective range of the descriptor (the G descriptor supplies its own scaling).

  If the magnitude of the data field is outside G descriptor range, E editing is used, and the scale factor has the same effect as E output editing.
- For EN and ES editing, the scale factor has no effect.

The following shows output using the P edit descriptor:

Format	Value	Output
1PE12.3	–270.139	ΔΔ–2.701E+02
1P,E12.2	–270.139	ΔΔΔ–2.70E+02
–1PE12.2	–270.139	ΔΔΔ–0.03E+04

The following shows a FORMAT statement containing a scale factor:

```
      DIMENSION A(6)
      DO 10 I=1,6
10    A(I) = 25.
      WRITE (6, 100) A
100   FORMAT(' ', F8.2, 2PF8.2, F8.2)
```

The preceding statements produce the following results:

```
    25.00  2500.00  2500.00
  2500.00  2500.00  2500.00
```

Slash Editing (/)

The slash edit descriptor terminates data transfer for the current record and starts data transfer for a new record. It takes the following form:

[*r*]/

The *r* is a repeat specification. It must be a positive default integer literal constant; no kind parameter can be specified.

The range of *r* is 1 through 2147483647 (2**31–1) on Intel Itanium processors; 1 through 32767 (2**15–1) on IA-32 processors. If *r* is omitted, it is assumed to be 1.

Multiple slashes cause the system to skip input records or to output blank records, as follows:

- When *n* consecutive slashes appear between two edit descriptors, *n* – 1 records are skipped on input, or *n* – 1 blank records are output. The first slash terminates the current record. The second slash terminates the first skipped or blank record, and so on.
- When *n* consecutive slashes appear at the beginning or end of a format specification, *n* records are skipped or *n* blank records are output, because the opening and closing parentheses of the format specification are themselves a record initiator and terminator, respectively. For example, suppose the following statements are specified:

```
      WRITE (6,99)
99    FORMAT ('1',T51,'HEADING LINE'//T51,'SUBHEADING LINE'//)
```

The following lines are written:

```
                 Column 50, top of page
                 |
                 HEADING LINE
(blank line)
                 SUBHEADING LINE
(blank line)
(blank line)
```

Note that the first character of the record printed was reserved as a control character (see "Printing of Formatted Records").

Colon Editing (:)

The colon edit descriptor terminates format control if no more items are in the I/O list. For example, suppose the following statements are specified:

```
     PRINT 1,3
     PRINT 2,13
1    FORMAT (' I=',I2,' J=',I2)
2    FORMAT (' K=',I2,:,' L=',I2)
```

The following lines are written:

```
I=△3△J=
K=13
```

If I/O list items remain, the colon edit descriptor has no effect.

Dollar Sign ($) and Backslash (\) Editing

The dollar sign and backslash edit descriptors modify the output of carriage control specified by the first character of the record. They only affect carriage control for formatted files, and have no effect on input.

If the first character of the record is a blank or a plus sign (+), the dollar sign and backslash descriptors suppress carriage return (after printing the record).

For terminal device I/O, when this trailing carriage return is suppressed, a response follows output on the same line. For example, suppose the following statements are specified:

```
     TYPE 100
100  FORMAT (' ENTER RADIUS VALUE ',$)
     ACCEPT 200, RADIUS
200  FORMAT (F6.2)
```

The following prompt is displayed:

```
ENTER RADIUS VALUE
```

Any response (for example, "12.") is then displayed on the same line:

```
ENTER RADIUS VALUE    12.
```

If the first character of the record is 0, 1, or ASCII NUL, the dollar sign and backslash descriptors have no effect.

Consider the following:

```
      CHARACTER(20) MYNAME
      WRITE (*,9000)
9000  FORMAT ('0Please type your name:',\)
      READ  (*,9001) MYNAME
9001  FORMAT (A20)
      WRITE (*,9002) ' ',MYNAME
9002  FORMAT (1X,A20)
```

This example advances two lines, prompts for input, awaits input on the same line as the prompt, and prints the input.

Character Count Editing (Q)

The character count edit descriptor returns the remaining number of characters in the current input record.

The corresponding I/O list item must be of type integer or logical. For example, suppose the following statements are specified:

```
      READ (4,1000) XRAY, KK, NCHRS, (ICHR(I), I=1,NCHRS)
1000  FORMAT (E15.7,I4,Q,(80A1))
```

Two fields are read into variables XRAY and KK. The number of characters remaining in the record is stored in NCHRS, and exactly that many characters are read into the array ICHR. (This instruction can fail if the record is longer than 80 characters.)

If you place the character count descriptor first in a format specification, you can determine the length of an input record.

On output, the character count edit descriptor causes the corresponding I/O list item to be skipped.

Character String Edit Descriptors

Character string edit descriptors control the output of character strings. The character string edit descriptors are the character constant and H edit descriptor.

Although no string edit descriptor can be preceded by a repeat specification, a parenthesized group of string edit descriptors can be preceded by a repeat specification (see "Nested and Group Repeat Specifications").

Character Constant Editing

The character constant edit descriptor causes a character string to be output to an external record. It takes one of the following forms:

'*string*'

"*string*"

The *string* is a character literal constant; no kind parameter can be specified. Its length is the number of characters between the delimiters; two consecutive delimiters are counted as one character.

To include an apostrophe in a character constant that is enclosed by apostrophes, place two consecutive apostrophes (') in the format specification; for example:

```
50   FORMAT ('TODAY''S∆DATE∆IS:∆',I2,'/',I2,'/',I2)
```

Similarly, to include a quotation mark in a character constant that is enclosed by quotation marks, place two consecutive quotation marks ("") in the format specification.

See Also

- "Format Specifications"
- Character Constants in "Character Data Type"

H Editing

The H edit descriptor transfers data between the external record and the H edit descriptor itself. The H edit descriptor is a deleted feature in Fortran 95; it was obsolescent in Fortran 90. Intel Fortran fully supports features deleted in Fortran 95.

An H edit descriptor has the form of a Hollerith constant, as follows:

*n*H*string*

n

Is an unsigned, positive default integer literal constant (with no kind parameter) indicating the number of characters in *string* (including blanks and tabs).

The range of n is 1 through 2147483647 (2**31–1) on Intel Itanium processors; 1 through 32767 (2**15–1) on IA-32 processors. Actual useful ranges may be constrained by record sizes (RECL) and the file system.

string

Is a string of printable ASCII characters.

On input, the H edit descriptor transfers *n* characters from the external field to the edit descriptor. The first character appears immediately after the letter H. Any characters in the edit descriptor before input are replaced by the input characters.

On output, the H edit descriptor causes *n* characters following the letter H to be output to an external record.

See Also

- "Format Specifications"
- Appendix A, "Deleted and Obsolescent Language Features" for details on obsolescent features in Fortran 95 and Fortran 90

Nested and Group Repeat Specifications

Format specifications can include nested format specifications enclosed in parentheses; for example:

```
15   FORMAT (E7.2,I8,I2,(A5,I6))
35   FORMAT (A6,(L8(3I2)),A)
```

A group repeat specification can precede a nested group of edit descriptors. For example, the following statements are equivalent, and the second statement shows a group repeat specification:

```
50   FORMAT (I8,I8,F8.3,E15.7,F8.3,E15.7,F8.3,E15.7,I5,I5)
50   FORMAT (2I8,3(F8.3,E15.7),2I5)
```

If a nested group does not show a repeat count, a default count of 1 is assumed.

Normally, the string edit descriptors and control edit descriptors cannot be repeated (except for slash), but any of these descriptors can be enclosed in parentheses and preceded by a group repeat specification. For example, the following statements are valid:

```
76   FORMAT ('MONTHLY',3('TOTAL'))
100  FORMAT (I8,4(T7),A4)
```

See Also

- "Forms for Data Edit Descriptors" for details on repeat specifications for data edit descriptors
- "Interaction Between Format Specifications and I/O Lists" for details on group repeat specifications and format reversion

Variable Format Expressions

A variable format expression is a numeric expression enclosed in angle brackets (<>) that can be used in a FORMAT statement or in character format specifications.

The numeric expression can be any valid Fortran expression, including function calls and references to dummy arguments.

If the expression is not of type integer, it is converted to integer type before being used.

If the value of a variable format expression does not obey the restrictions on magnitude applying to its use in the format, an error occurs.

Variable format expressions cannot be used with the H edit descriptor, and they are not allowed in character format specifications.

Variable format expressions are evaluated each time they are encountered in the scan of the format. If the value of the variable used in the expression changes during the execution of the I/O statement, the new value is used the next time the format item containing the expression is processed.

Examples

Consider the following statement:

```
FORMAT (I<J+1>)
```

When the format is scanned, the preceding statement performs an I (integer) data transfer with a field width of J+1. The expression is reevaluated each time it is encountered in the normal format scan.

Consider the following statements:

```
     DIMENSION A(5)
     DATA A/1.,2.,3.,4.,5./
     DO 10 I=1,10
     WRITE (6,100) I
100  FORMAT (I<MAX(I,5)>)
10   CONTINUE

     DO 20 I=1,5
     WRITE (6,101) (A(I), J=1,I)
101  FORMAT (<I>F10.<I-1>)
20   CONTINUE
END
```

On execution, these statements produce the following output:

```
 1
 2
 3
 4
 5
  6
   7
    8
     9
     10
     1.
    2.0        2.0
   3.00       3.00     3.00
  4.000      4.000    4.000     4.000
 5.0000    5.0000    5.0000    5.0000    5.0000
```

See Also

"Interaction Between Format Specifications and I/O Lists"

Printing of Formatted Records

On output, if a file was opened with CARRIAGECONTROL='FORTRAN' in effect or the file is being processed by the `fortpr` format utility, the first character of a record transmitted to a line printer or terminal is typically a character that is not printed, but used to control vertical spacing.

Table 11-5 lists the valid control characters for printing.

Table 11-5 Control Characters for Printing

Character	Meaning	Effect
+	Overprinting	Outputs the record (at the current position in the current line) and a carriage return.
Δ	One line feed	Outputs the record (at the beginning of the following line) and a carriage return.
0	Two line feeds	Outputs the record (after skipping a line) and a carriage return.
1	Next page	Outputs the record (at the beginning of a new page) and a carriage return.
$	Prompting	Outputs the record (at the beginning of the following line), but no carriage return.

Table 11-5 Control Characters for Printing

Character	Meaning	Effect
ASCII NUL[1]	Overprinting with no advance	Outputs the record (at the current position in the current line), but no carriage return.

1. Specify as CHAR(0).

Any other character is interpreted as a blank and is deleted from the print line. If you do not specify a control character for printing, the first character of the record is not printed.

Interaction Between Format Specifications and I/O Lists

Format control begins with the execution of a formatted I/O statement. Each action of format control depends on information provided jointly by the next item in the I/O list (if one exists) and the next edit descriptor in the format specification.

Both the I/O list and the format specification are interpreted from left to right, unless repeat specifications or implied-DO lists appear.

If an I/O list specifies at least one list item, at least one data edit descriptor (I, B, O, Z, F, E, EN, ES, D, G, L, or A) or the Q edit descriptor must appear in the format specification; otherwise, an error occurs.

Each data edit descriptor (or Q edit descriptor) corresponds to one item in the I/O list, except that an I/O list item of type complex requires the interpretation of two F, E, EN, ES, D, or G edit descriptors. No I/O list item corresponds to a control edit descriptor (X, P, T, TL, TR, SP, SS, S, BN, BZ, $, or :), or a character string edit descriptor (H and character constants). For character string edit descriptors, data transfer occurs directly between the external record and the format specification.

When format control encounters a data edit descriptor in a format specification, it determines whether there is a corresponding I/O list item specified. If there is such an item, it is transferred under control of the edit descriptor, and then format control proceeds. If there is no corresponding I/O list item, format control terminates.

If there are no other I/O list items to be processed, format control also terminates when the following occurs:

- A colon edit descriptor is encountered.
- The end of the format specification is reached.

If additional I/O list items remain, part or all of the format specification is reused in format reversion.

In format reversion, the current record is terminated and a new one is initiated. Format control then reverts to one of the following (in order) and continues from that point:

1. The group repeat specification whose opening parenthesis matches the next-to-last closing parenthesis of the format specification
2. The initial opening parenthesis of the format specification

Format reversion has no effect on the scale factor, the sign control edit descriptors (S, SP, or SS), or the blank interpretation edit descriptors (BN or BZ).

Example

The data in file FOR002.DAT is to be processed 2 records at a time. Each record starts with a number to be put into an element of a vector B, followed by 5 numbers to be put in a row in matrix A.

FOR002.DAT contains the following data:

```
001 0101 0102 0103 0104 0105
002 0201 0202 0203 0204 0205
003 0301 0302 0303 0304 0305
004 0401 0402 0403 0404 0405
005 0501 0502 0503 0504 0505
006 0601 0602 0603 0604 0605
007 0701 0702 0703 0704 0705
008 0801 0802 0803 0804 0805
009 0901 0902 0903 0904 0905
010 1001 1002 1003 1004 1005
```

Example 11-1 shows how several different format specifications interact with I/O lists to process data in file FOR002.DAT.

Example 11-1 Interaction Between Format Specifications and I/O Lists

```
          INTEGER I, J, A(2,5), B(2)
          OPEN (unit=2, access='sequential', file='FOR002.DAT')

1         READ (2,100) (B(I), (A(I,J), J=1,5),I=1,2)

2   100   FORMAT (2 (I3, X, 5(I4,X), /) )

3         WRITE (6,999) B, ((A(I,J),J=1,5),I=1,2)
    999   FORMAT (' B is ', 2(I3, X), ';  A is', /
         1       (' ', 5 (I4, X)) )
4         READ (2,200) (B(I), (A(I,J), J=1,5),I=1,2)
    200   FORMAT (2 (I3, X, 5(I4,X), :/) )
```

Example 11-1 Interaction Between Format Specifications and I/O Lists

```
5        WRITE (6,999) B, ((A(I,J),J=1,5),I=1,2)

6        READ (2,300) (B(I), (A(I,J), J=1,5),I=1,2)
   300   FORMAT ( (I3, X, 5(I4,X)) )
7        WRITE (6,999) B, ((A(I,J),J=1,5),I=1,2)

8        READ (2,400) (B(I), (A(I,J), J=1,5),I=1,2)
   400   FORMAT ( I3, X, 5(I4,X) )

9        WRITE (6,999) B, ((A(I,J),J=1,5),I=1,2)
         END
```

1 This statement reads B(1); then A(1,1) through A(1,5); then B(2) and A(2,1) through A(2,5).

The first record read (starting with 001) starts the processing of the I/O list.

2 There are two records, each in the format I3, X, 5(I4, X). The slash (/) forces the reading of the second record after A(1,5) is processed. It also forces the reading of the third record after A(2,5) is processed; no data is taken from that record.

3 This statement produces the following output:

```
B is   1   2 ;  A is
 101  102  103  104  105
 201  202  203  204  205
```

4 This statement reads the record starting with 004. The slash (/) forces the reading of the next record after A(1,5) is processed. The colon (:) stops the reading after A(2,5) is processed, but before the slash (/) forces another read.

5 This statement produces the following output:

```
B is   4   5 ;  A is
 401  402  403  404  405
 501  502  503  504  505
```

6 This statement reads the record starting with 006. After A(1,5) is processed, format reversion causes the next record to be read and starts format processing at the left parenthesis before the I3.

7 This statement produces the following output:

```
B is   6   7 ;  A is
 601  602  603  604  605
 701  702  703  704  705
```

8 This statement reads the record starting with 008. After A(1,5) is processed, format reversion causes the next record to be read and starts format processing at the left parenthesis before the I4.

9 This statement produces the following output:

```
B is   8  90 ;  A is
 801  802  803  804  805
 9010 9020 9030 9040 100
```

The record 009 0901 0902 0903 0904 0905 is processed with I4 as "009 " for B(2), which is 90. X skips the next "0". Then "901 " is processed for A(2,1), which is 9010, "902 " for A(2,2), "903 " for A(2,3), and "904 " for A(2,4). The repeat specification of 5 is now exhausted and the format ends. Format reversion causes another record to be read and starts format processing at the left parenthesis before the I4, so "010 " is read for A(2,5), which is 100.

See Also

- "Data Edit Descriptors"
- "Control Edit Descriptors"
- "Character String Edit Descriptors"
- "Scale Factor Editing (P)"
- "Character Count Editing (Q)" for details on the Q edit descriptor

File Operation I/O Statements

12

This chapter contains information on the following file connection, inquiry, and positioning statements:

- The "BACKSPACE Statement"
  Positions a sequential file at the beginning of the preceding record.
- The "CLOSE Statement"
  Terminates the connection between a logical unit and a file or device.
- The "DELETE Statement"
  Deletes a record from a relative file.
- The "ENDFILE Statement"
  For sequential files, writes an end-of-file record to the file and positions the file after this record. For direct access files, truncates the file after the current record.
- The "INQUIRE Statement"
  Requests information on the status of specified properties of a file or logical unit.
- The "OPEN Statement"
  Connects a Fortran logical unit to a file or device and declares attributes for read and write operations.
- The "REWIND Statement"
  Positions a sequential file at the beginning of the file.
- The "UNLOCK Statement"
  Frees a record in a relative or sequential file that was locked by a previous READ statement.

See Also

- Chapter 10, "Data Transfer I/O Statements"
- "I/O Control List" for details on control specifiers
- Your user's guide for details on record position, advancement, and transfer

BACKSPACE Statement

The BACKSPACE statement positions a sequential file at the beginning of the preceding record, making it available for subsequent I/O processing. It takes one of the following forms:

BACKSPACE ([UNIT=]*io-unit* [, ERR=*label*] [, IOSTAT=*i-var*])

BACKSPACE *io-unit*

io-unit

Is an external unit specifier.

label

Is the label of the branch target statement that receives control if an error occurs.

i-var

Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs.

Rules and Behavior

The I/O unit number must specify an open file on disk or magnetic tape.

Backspacing from the current record *n* is performed by rewinding to the start of the file and then performing *n*–1 successive READs to reach the previous record.

A BACKSPACE statement must not be specified for a file that is open for direct or append access, because *n* is not available to the Fortran I/O system.

If a file is already positioned at the beginning of a file, a BACKSPACE statement has no effect.

Examples

The following statement repositions the file connected to I/O unit 4 back to the preceding record:

```
BACKSPACE 4
```

Consider the following statement:

```
BACKSPACE (UNIT=9, IOSTAT=IOS, ERR=10)
```

This statement positions the file connected to unit 9 back to the preceding record. If an error occurs, control is transferred to the statement labeled 10, and a positive integer is stored in variable IOS.

See Also

- "Unit Specifier" for details on the UNIT control specifier
- "Branch Specifiers" for details on the ERR control specifier
- "I/O Status Specifier" for details on the IOSTAT control specifier

- "ACCESS Specifier" for details on append access
- Your user's guide for details on record position, advancement, and transfer

CLOSE Statement

The CLOSE statement disconnects a file from a unit. It takes the following form:

CLOSE ([UNIT=]*io-unit* [, { STATUS | DISPOSE | DISP } =*p*] [, ERR=*label*] [, IOSTAT=*i-var*])

io-unit

Is an external unit specifier.

p

Is a scalar default character expression indicating the status of the file after it is closed. It has one of the following values:

'KEEP' or 'SAVE'	Retains the file after the unit closes.
'DELETE'	Deletes the file after the unit closes.[1]
'PRINT'[2]	Submits the file to the line printer spooler, then retains it.
'PRINT/DELETE'[2]	Submits the file to the line printer spooler, then deletes it.
'SUBMIT'	Forks a process to execute the file.
'SUBMIT/DELETE'	Forks a process to execute the file, then deletes the file after the fork is completed.

1. Unless OPEN(READONLY) is in effect.
2. Use only on sequential files.

The default is 'DELETE' for scratch files and QuickWin applications (W*32, W*64). For all other files, the default is 'KEEP'.

label

Is the label of the branch target statement that receives control if an error occurs.

i-var

Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs.

Rules and Behavior

The CLOSE statement specifiers can appear in any order. An I/O unit must be specified, but the UNIT keyword is optional if the unit specifier is the first item in the I/O control list.

The status specified in the CLOSE statement supersedes the status specified in the OPEN statement, except that a file opened as a scratch file cannot be saved, printed, or submitted, and a file opened for read-only access cannot be deleted.

If a CLOSE statement is specified for a unit that is not open, it has no effect.

Example

Consider the following statement:

```
CLOSE (UNIT=J, STATUS='DELETE', ERR=99)
```

This statement closes the file connected to unit J and deletes it. If an error occurs, control is transferred to the statement labeled 99.

See Also

- "READONLY Specifier"
- "Unit Specifier" for details on the UNIT control specifier
- "Branch Specifiers" for details on the ERR control specifier
- "I/O Status Specifier" for details on the IOSTAT control specifier

DELETE Statement

The DELETE statement deletes a record from a relative file. It takes the following form:

DELETE ([UNIT=]*io-unit*, REC=*r* [, ERR=*label*] [, IOSTAT=*i-var*])

io-unit

Is an external unit specifier.

r

Is a scalar numeric expression indicating the record number to be deleted.

label

Is the label of the branch target statement that receives control if an error occurs.

i-var

Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs.

Rules and Behavior

In a relative file, the DELETE statement deletes the direct access record specified by *r*. If REC=*r* is omitted, the current record is deleted. When the direct access record is deleted, any associated variable is set to the next record number.

The DELETE statement logically removes the appropriate record from the specified file by locating the record and marking it as a deleted record. It then frees the position formerly occupied by the deleted record so that a new record can be written at that position.

NOTE. *You must use the compiler option specifying OpenVMS defaults for* READ*s to detect that a record has been deleted.*

Examples

The following statement deletes the fifth record in the file connected to I/O unit 10:

```
DELETE (10, REC=5)
```

Suppose the following statement is specified:

```
DELETE (UNIT=9, REC=10, IOSTAT=IOS, ERR=20)
```

The tenth record in the file connected to unit 9 is deleted. If an error occurs, control is transferred to the statement labeled 20, and a positive integer is stored in the variable IOS.

See Also

- "Alternative Syntax for the DELETE Statement"
- "Unit Specifier" for details on the UNIT control specifier
- "Branch Specifiers" for details on the ERR control specifier
- "I/O Status Specifier" for details on the IOSTAT control specifier
- "Record Specifier" for details on the REC control specifier
- Your user's guide for details on compiler options

ENDFILE Statement

For sequential files, the ENDFILE statement writes an end-of-file record to the file and positions the file after this record (the terminal point). For direct access files, the ENDFILE statement truncates the file after the current record.

An ENDFILE statement takes one of the following forms:

ENDFILE ([UNIT=]*io-unit* [, ERR=*label*] [, IOSTAT=*i-var*])

ENDFILE *io-unit*

io-unit

Is an external unit specifier.

label

Is the label of the branch target statement that receives control if an error occurs.

i-var

Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs.

Rules and Behavior

If the unit specified in the ENDFILE statement is not open, the default file is opened for unformatted output.

An end-of-file record can be written only to files with sequential organization that are accessed as formatted-sequential or unformatted-segmented sequential files.

An ENDFILE statement performed on a direct access file always truncates the file.

End-of-file records should not be written in files that are read by programs written in a language other than Fortran.

NOTE. *If you use the compiler option specifying OpenVMS defaults and an* ENDFILE *is performed on a sequential unit, an actual one byte record containing a* CTRL+Z *is written to the file. If this option is not specified, an internal* ENDFILE *flag is set and the file is truncated. The option does not affect* ENDFILE *on relative files; such files are truncated.*

Examples

The following statement writes an end-of-file record to I/O unit 2:

```
ENDFILE 2
```

Suppose the following statement is specified:

```
ENDFILE (UNIT=9, IOSTAT=IOS, ERR=10)
```

An end-of-file record is written to the file connected to unit 9. If an error occurs, control is transferred to the statement labeled 10, and a positive integer is stored in variable IOS.

See Also

- "Unit Specifier" for details on the UNIT control specifier
- "Branch Specifiers" for details on the ERR control specifier
- "I/O Status Specifier" for details on the IOSTAT control specifier
- Your user's guide for details on record position, advancement, and transfer

INQUIRE Statement

The INQUIRE statement returns information on the status of specified properties of a file or logical unit. It takes one of the following forms:

Inquiring by File:

INQUIRE (FILE=*name* [, ERR=*label*] [, IOSTAT=*i-var*] [, DEFAULTFILE=*def*], *slist*)

Inquiring by Unit:

INQUIRE ([UNIT=]*io-unit* [, ERR=*label*] [, IOSTAT=*i-var*], *slist*)

Inquiring by Output List:

INQUIRE (IOLENGTH=*len*) *out-item-list*

name

Is a scalar default character expression specifying the name of the file for inquiry.

label

Is the label of the branch target statement that receives control if an error occurs.

i-var

Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs.

slist

Is one or more inquiry specifiers. Each specifier can appear only once. (The inquiry specifiers are described individually in the following sections.)

def

Is a scalar default character expression specifying a default file pathname string. (For more information on the DEFAULTFILE specifier, see "DEFAULTFILE Specifier".)

io-unit

Is an external unit specifier.

The unit does not have to exist, nor does it need to be connected to a file. If the unit is connected to a file, the inquiry encompasses both the connection and the file.

len

Is a scalar integer variable that is assigned a value corresponding to the length of an unformatted, direct-access record resulting from the use of the *out-item-list* in a WRITE statement.

The value is suitable to use as a RECL specifier value in an OPEN statement that connects a file for unformatted, direct access.

The unit of the value is 4-byte longwords, by default. However, if you specify the compiler option specifying record length in bytes, the unit is bytes.

out-item-list

Is a list of one or more output items (see "I/O Lists").

Rules and Behavior

The control specifiers ([UNIT=]*io-unit*, ERR=*label*, and IOSTAT=*i-var*) and inquiry specifiers can appear anywhere within the parentheses following INQUIRE. However, if the UNIT keyword is omitted, the *io-unit* must appear first in the list.

An INQUIRE statement can be executed before, during, or after a file is connected to a unit. The specifier values returned are those that are current when the INQUIRE statement executes.

To get file characteristics, specify the INQUIRE statement after opening the file.

Examples

The following are examples of INQUIRE statements:

```
INQUIRE (FILE='FILE_B', EXIST=EXT)
INQUIRE (4, FORM=FM, IOSTAT=IOS, ERR=20)
INQUIRE (IOLENGTH=LEN) A, B
```

In the last statement, you can use the length returned in LEN as the value for the RECL specifier in an OPEN statement that connects a file for unformatted direct access. If you have already specified a value for RECL, you can check LEN to verify that A and B are less than or equal to the record length you specified.

See Also

- "Unit Specifier" for details on the UNIT control specifier
- "Branch Specifiers" for details on the ERR control specifier
- "I/O Status Specifier" for details on the IOSTAT control specifier
- "RECL Specifier" for details on using the specifier in OPEN statements
- "FILE Specifier" for details on using the specifier in OPEN statements
- "DEFAULTFILE Specifier" for details on using the specifier in OPEN statements

ACCESS Specifier

The ACCESS specifier asks how a file is connected. It takes the following form:

ACCESS = *acc*

acc

Is a scalar default character variable that is assigned one of the following values:

'SEQUENTIAL'	If the file is connected for sequential access
'DIRECT'	If the file is connected for direct access
'UNDEFINED'	If the file is not connected

ACTION Specifier

The ACTION specifier asks which I/O operations are allowed for a file. It takes the following form:

ACTION = *act*

act

Is a scalar default character variable that is assigned one of the following values:

'READ'	If the file is connected for input only
'WRITE'	If the file is connected for output only
'READWRITE'	If the file is connected for both input and output
'UNDEFINED'	If the file is not connected

BINARY Specifier (W*32, W*64)

The BINARY specifier asks whether a file is connected to a binary file. It takes the following form:

BINARY = *bin*

bin

Is a scalar default character variable that is assigned one of the following values:

'YES'	If the file is connected to a binary file
'NO'	If the file is connected to a nonbinary file
'UNKNOWN'	If the file is not connected

BLANK Specifier

The BLANK specifier asks what type of blank control is in effect for a file. It takes the following form:

BLANK = *blnk*

blnk

Is a scalar default character variable that is assigned one of the following values:

'NULL'	If null blank control is in effect for the file
'ZERO'	If zero blank control is in effect for the file
'UNDEFINED'	If the file is not connected, or it is not connected for formatted data transfer

BLOCKSIZE Specifier

The BLOCKSIZE specifier asks about the I/O buffer size. It takes the following form:

BLOCKSIZE = *bks*

bks

Is a scalar integer variable.

The *bks* is assigned the current size of the I/O buffer. If the unit or file is not connected, the value assigned is zero.

BUFFERED Specifier

The BUFFERED specifier asks whether run-time buffering is in effect. It takes the following form:

BUFFERED = *bf*

bf

Is a scalar default character variable that is assigned one of the following values:

'YES'	If the file or unit is connected and buffering is in effect
'NO'	If the file or unit is connected and buffering is not in effect
'UNKNOWN'	If the file or unit is not connected

CARRIAGECONTROL Specifier

The CARRIAGECONTROL specifier asks what type of carriage control is in effect for a file. It takes the following form:

CARRIAGECONTROL = *cc*

cc

Is a scalar default character variable that is assigned one of the following values:

'FORTRAN'	If the file is connected with Fortran carriage control in effect
'LIST'	If the file is connected with implied carriage control in effect
'NONE'	If the file is connected with no carriage control in effect
'UNKNOWN'	If the file is not connected

CONVERT Specifier

The CONVERT specifier asks what type of data conversion is in effect for a file. It takes the following form:

CONVERT = *fm*

fm

Is a scalar default character variable that is assigned one of the following values:

'LITTLE_ENDIAN'	If the file is connected with little endian integer and IEEE* floating-point data conversion in effect
'BIG_ENDIAN'	If the file is connected with big endian integer and IEEE floating-point data conversion in effect
'CRAY'	If the file is connected with big endian integer and CRAY* floating-point data conversion in effect
'FDX'	If the file is connected with little endian integer and VAX* processor F_floating, D_floating, and IEEE X_floating data conversion in effect
'FGX'	If the file is connected with little endian integer and VAX processor F_floating, G_floating, and IEEE X_floating data conversion in effect
'IBM'	If the file is connected with big endian integer and IBM* System\370 floating-point data conversion in effect
'VAXD'	If the file is connected with little endian integer and VAX processor F_floating, D_floating, and H_floating in effect
'VAXG'	If the file is connected with little endian integer and VAX processor F_floating, G_floating, and H_floating in effect
'NATIVE'	If the file is connected with no data conversion in effect
'UNKNOWN'	If the file or unit is not connected for unformatted data transfer

DELIM Specifier

The DELIM specifier asks how character constants are delimited in list-directed and namelist output. It takes the following form:

DELIM = *del*

del

Is a scalar default character variable that is assigned one of the following values:

'APOSTROPHE'	If apostrophes are used to delimit character constants in list-directed and namelist output
'QUOTE'	If quotation marks are used to delimit character constants in list-directed and namelist output
'NONE'	If no delimiters are used
'UNDEFINED'	If the file is not connected, or is not connected for formatted data transfer

DIRECT Specifier

The DIRECT specifier asks whether a file is connected for direct access. It takes the following form:

DIRECT = *dir*

dir

Is a scalar default character variable that is assigned one of the following values:

'YES'	If the file is connected for direct access
'NO'	If the file is not connected for direct access
'UNKNOWN'	If the file is not connected

EXIST Specifier

The EXIST specifier asks whether a file exists and can be opened. It takes the following form:

EXIST = *ex*

ex

Is a scalar default logical variable that is assigned one of the following values:

.TRUE.	If the specified file exists and can be opened, or if the specified unit exists

.FALSE.	If the specified file or unit does not exist or if the file exists but cannot be opened

The unit exists if it is a number in the range allowed by the processor.

FORM Specifier

The FORM specifier asks whether a file is connected for formatted, unformatted, or binary (W*32, W*64) data transfer. It takes the following form:

FORM = *fm*

fm

Is a scalar default character variable that is assigned one of the following values:

'FORMATTED'	If the file is connected for formatted data transfer
'UNFORMATTED'	If the file is connected for unformatted data transfer
'BINARY'	If the file is connected for binary data transfer
'UNDEFINED'	If the file is not connected

FORMATTED Specifier

The FORMATTED specifier asks whether a file is connected for formatted data transfer. It takes the following form:

FORMATTED = *fmt*

fmt

Is a scalar default character variable that is assigned one of the following values:

'YES'	If the file is connected for formatted data transfer
'NO'	If the file is not connected for formatted data transfer
'UNKNOWN'	If the processor cannot determine whether the file is connected for formatted data transfer

IOFOCUS Specifier (W*32, W*64)

The IOFOCUS specifier asks if the indicated unit is the active window in a QuickWin application. It takes the following form:

IOFOCUS = *iof*

iof

Is a scalar default logical variable that is assigned one of the following values:

.TRUE.	If the specified unit is the active window in a QuickWin application
.FALSE.	If the specified unit is not the active window in a QuickWin application

If you use this specifier with a non-Windows* application, an error occurs.

MODE Specifier

MODE is a nonstandard synonym for ACTION (see "ACTION Specifier").

NAME Specifier

The NAME specifier returns the name of a file. It takes the following form:

NAME = *nme*

nme

Is a scalar default character variable that is assigned the name of the file to which the unit is connected. If the file does not have a name, *nme* is undefined.

The value assigned to *nme* is not necessarily the same as the value given in the FILE specifier. However, the value that is assigned is always valid for use with the FILE specifier in an OPEN statement, unless the value has been truncated in a way that makes it unacceptable. (Values are truncated if the declaration of *nme* is too small to contain the entire value.)

NOTE. *The* FILE *and* NAME *specifiers are synonyms when used with the* OPEN *statement, but not when used with the* INQUIRE *statement.*

See Also

The appropriate manual in your operating system documentation set for details on the maximum size of file pathnames

NAMED Specifier

The NAMED specifier asks whether a file is named. It takes the following form:

NAMED = *nmd*

nmd

Is a scalar default logical variable that is assigned one of the following values:

.TRUE.	If the file has a name
.FALSE.	If the file does not have a name

NEXTREC Specifier

The NEXTREC specifier asks where the next record can be read or written in a file connected for direct access. It takes the following form:

NEXTREC = *nr*

nr

Is a scalar integer variable that is assigned a value as follows:

- If the file is connected for direct access and a record (r) was previously read or written, the value assigned is r + 1.
- If no record has been read or written, the value assigned is 1.
- If the file is not connected for direct access, or if the file position cannot be determined because of an error condition, the value assigned is zero.
- If the file is connected for direct access and a REWIND has been performed on the file, the value assigned is 1.

NUMBER Specifier

The NUMBER specifier asks the number of the unit connected to a file. It takes the following form:

NUMBER = *num*

num

Is a scalar integer variable.

The *num* is assigned the number of the unit currently connected to the specified file. If there is no unit connected to the file, the value assigned is –1.

OPENED Specifier

The OPENED specifier asks whether a file is connected. It takes the following form:

OPENED = *od*

od

Is a scalar default logical variable that is assigned one of the following values:

.TRUE.	If the specified file or unit is connected
.FALSE.	If the specified file or unit is not connected

ORGANIZATION Specifier

The ORGANIZATION specifier asks how the file is organized. It takes the following form:

ORGANIZATION = *org*

org

Is a scalar default character variable that is assigned one of the following values:

'SEQUENTIAL'	If the file is a sequential file
'RELATIVE'	If the file is a relative file
'UNKNOWN'	If the processor cannot determine the file's organization

PAD Specifier

The PAD specifier asks whether blank padding was specified for the file. It takes the following form:

PAD = *pd*

pd

Is a scalar default character variable that is assigned one of the following values:

'YES'	If the file or unit is not connected, or it was connected with PAD='YES'
'NO'	If the file or unit was connected with PAD='NO'

POSITION Specifier

The POSITION specifier asks the position of the file. It takes the following form:

POSITION = *pos*

pos

Is a scalar default character variable that is assigned one of the following values:

'REWIND'	If the file is connected with its position at its initial point

'APPEND'	If the file is connected with its position at its terminal point (or before its end-of-file record, if any)
'ASIS'	If the file is connected without changing its position
'UNDEFINED'	If the file is not connected, or is connected for direct access data transfer and a REWIND statement has not been performed on the unit

See Also

Your user's guide for details on record position, advancement, and transfer

READ Specifier

The READ specifier asks whether a file can be read. It takes the following form:

READ = *rd*

rd

Is a scalar default character variable that is assigned one of the following values:

'YES'	If the file can be read
'NO'	If the file cannot be read
'UNKNOWN'	If the processor cannot determine whether the file can be read

READWRITE Specifier

The READWRITE specifier asks whether a file can be both read and written to. It takes the following form:

READWRITE = *rdwr*

rdwr

Is a scalar default character variable that is assigned one of the following values:

'YES'	If the file can be both read and written to
'NO'	If the file cannot be both read and written to
'UNKNOWN'	If the processor cannot determine whether the file can be both read and written to

RECL Specifier

The RECL specifier asks the maximum record length for a file. It takes the following form:

RECL = *rcl*

rcl

Is a scalar integer variable that is assigned a value as follows:

- If the file or unit is connected, the value assigned is the maximum record length allowed.
- If the file does not exist, or is not connected, the value assigned is zero.

The assigned value is expressed in 4-byte units if the file is currently (or was previously) connected for unformatted data transfer; otherwise, the value is expressed in bytes.

RECORDTYPE Specifier

The RECORDTYPE specifier asks which type of records are in a file. It takes the following form:

RECORDTYPE = *rtype*

rtype

Is a scalar default character variable that is assigned one of the following values:

'FIXED'	If the file is connected for fixed-length records
'VARIABLE'	If the file is connected for variable-length records
'SEGMENTED'	If the file is connected for unformatted sequential data transfer using segmented records
'STREAM'	If the file's records are not terminated
'STREAM_CR'	If the file's records are terminated with a carriage return
'STREAM_LF'	If the file's records are terminated with a line feed
'UNKNOWN'	If the file is not connected

SEQUENTIAL Specifier

The SEQUENTIAL specifier asks whether a file is connected for sequential access. It takes the following form:

SEQUENTIAL = *seq*

seq

Is a scalar default character variable that is assigned one of the following values:

'YES'	If the file is connected for sequential access
'NO'	If the file is not connected for sequential access

'UNKNOWN'	If the processor cannot determine whether the file is connected for sequential access

SHARE Specifier (W*32, W*64)

The SHARE specifier asks the current share status of a file or unit. It takes the following form:

SHARE = *shr*

shr

Is a scalar default character variable that is assigned one of the following values:

'DENYRW'	If the file is connected for deny-read/write mode
'DENYWR'	If the file is connected for deny-write mode
'DENYRD'	If the file is connected for deny-read mode
'DENYNONE'	If the file is connected for deny-none mode
'UNKNOWN'	If the file or unit is not connected

UNFORMATTED Specifier

The UNFORMATTED specifier asks whether a file is connected for unformatted data transfer. It takes the following form:

UNFORMATTED = *unf*

unf

Is a scalar default character variable that is assigned one of the following values:

'YES'	If the file is connected for unformatted data transfer
'NO'	If the file is not connected for unformatted data transfer
'UNKNOWN'	If the processor cannot determine whether the file is connected for unformatted data transfer

WRITE Specifier

The WRITE specifier asks whether a file can be written to. It takes the following form:

WRITE = *wr*

wr

Is a scalar default character variable that is assigned one of the following values:

'YES'	If the file can be written to
'NO'	If the file cannot be written to
'UNKNOWN'	If the processor cannot determine whether the file can be written to

OPEN Statement

The OPEN statement connects an external file to a unit, creates a new file and connects it to a unit, creates a preconnected file, or changes certain properties of a connection.

The OPEN statement takes the following form:

OPEN ([UNIT=]*io-unit* [, FILE=*name*] [, ERR=*label*] [, IOSTAT=*i-var*], *slist*)

io-unit

Is an external unit specifier.

name

Is a character or numeric expression specifying the name of the file to be connected. For more information, see "FILE Specifier".

label

Is the label of the branch target statement that receives control if an error occurs.

i-var

Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs.

slist

Is one or more OPEN specifiers in the form *specifier=value* or *specifier*. Each specifier can appear only once.

The OPEN specifiers and their acceptable values are summarized in Table 12-1.

The OPEN specifiers are described individually in the following sections. The control specifiers that can be specified in an OPEN statement (UNIT, ERR, and IOSTAT) are discussed in "I/O Control List".

Table 12-1 OPEN Statement Specifiers and Values

Specifier	Values[1]	Function	Default
ACCESS	'SEQUENTIAL' 'DIRECT' 'APPEND'	Access mode	'SEQUENTIAL'

Table 12-1 OPEN Statement Specifiers and Values

Specifier	Values[1]	Function	Default
ACTION (or MODE)	'READ' 'WRITE' 'READWRITE'	File access	'READWRITE'
ASSOCIATEVARIABLE	var	Next direct access record	No default
BLANK	'NULL' 'ZERO'	Interpretation of blanks	'NULL'
BLOCKSIZE	n_expr	Physical block size	Filesystem default
BUFFERCOUNT	n_expr	Number of I/O buffers	One
BUFFERED	'YES' 'NO'	Buffering for WRITE operations	'NO'
CARRIAGECONTROL	'FORTRAN' 'LIST' 'NONE'	Print control	Formatted: 'LIST'[2] Unformatted: 'NONE'
CONVERT	'LITTLE_ENDIAN' 'BIG_ENDIAN' 'CRAY' 'FDX' 'FGX' 'IBM' 'VAXD' 'VAXG' 'NATIVE'	Numeric format specification	'NATIVE'
DEFAULTFILE	c_expr	Default file pathname	Current working directory
DELIM	'APOSTROPHE' 'QUOTE' 'NONE'	Delimiter for character constants	'NONE'
DISPOSE (or DISP)	'KEEP' or 'SAVE' 'DELETE' 'PRINT' 'PRINT/DELETE' 'SUBMIT' 'SUBMIT/DELETE'	File disposition at close	'KEEP'
ERR	label	Error transfer control	No default
FILE (or NAME)	c_expr	File pathname (file name)	fort.n[3]
FORM	'FORMATTED' 'UNFORMATTED' 'BINARY' [2]	Format type	Depends on ACCESS setting

Table 12-1 OPEN Statement Specifiers and Values

Specifier	Values[1]	Function	Default
IOFOCUS[2]	.TRUE. .FALSE.	Active window in QuickWin application	.TRUE.[4]
IOSTAT	var	I/O status	No default
MAXREC	n_expr	Direct access record limit	No limit
ORGANIZATION	'SEQUENTIAL' 'RELATIVE'	File organization	'SEQUENTIAL'
PAD	'YES' 'NO'	Record padding	'YES'
POSITION	'ASIS' 'REWIND' 'APPEND'	File positioning	'ASIS'
READONLY	No value	Write protection	No default
RECL (or RECORDSIZE)	n_expr	Record length	Depends on RECORDTYPE, ORGANIZATION, and FORM settings[5]
RECORDTYPE	'FIXED' 'VARIABLE' 'SEGMENTED' 'STREAM' 'STREAM_CR' 'STREAM_LF'	Record type	Depends on ORGANIZATION, ACCESS, CARRIAGECONTROL, and FORM settings
SHARE[2,6]	'DENYRW' 'DENYWR' 'DENYRD' 'DENYNONE'	File locking	'DENYWR'
SHARED[6]	No value	File sharing allowed	L*X: SHARED W*32, W*64: Not shared
STATUS (or TYPE)	'OLD' 'NEW' 'SCRATCH' 'REPLACE' 'UNKNOWN'	File status at open	'UNKNOWN'[7]
TITLE[2]	c_expr	Title for child window in QuickWin application	No default
UNIT	n_expr	Logical unit number	No default; an io-unit must be specified

Table 12-1 OPEN Statement Specifiers and Values

Specifier	Values[1]	Function	Default
USEROPEN	func	User program option	No default

1. Key to Values:
   c_expr: A scalar default character expression
   func: An external function
   label: A statement label
   n_expr: A scalar numeric expression
   var: A scalar default integer variable
2. W*32, W*64
3. n is the unit number
4. If unit '*' is specified, the default is .FALSE.; otherwise, the default is .TRUE..
5. On Linux systems, the default depends only on the FORM setting
6. For information on file sharing, see your user's guide.
7. The default differs under certain conditions (see "STATUS Specifier").

Rules and Behavior

The control specifiers ([UNIT=]*io-unit*, ERR=*label*, and IOSTAT=*i-var*) and OPEN specifiers can appear anywhere within the parentheses following OPEN. However, if the UNIT keyword is omitted, the *io-unit* must appear first in the list.

Specifier values that are scalar numeric expressions can be any integer or real expression. The value of the expression is converted to integer data type before it is used in the OPEN statement.

Only one unit at a time can be connected to a file, but multiple OPENs can be performed on the same unit. If an OPEN statement is executed for a unit that already exists, the following occurs:

- If FILE is not specified, or FILE specifies the same file name that appeared in a previous OPEN statement, the current file remains connected.

  If the file names are the same, the values for the BLANK, CARRIAGECONTROL, CONVERT, DELIM, DISPOSE, ERR, IOSTAT, and PAD specifiers can be changed. Other OPEN specifier values cannot be changed, and the file position is unaffected.
- If FILE specifies a different file name, the previous file is closed and the new file is connected to the unit.

The ERR and IOSTAT specifiers from any previously executed OPEN statement have no effect on any currently executing OPEN statement. If an error occurs, no file is opened or created.

Secondary operating system messages do not display when IOSTAT is specified. To display these messages, remove IOSTAT or use a platform-specific method. (For more information, see your user's guide.)

Examples

You can specify character values at run time by substituting a character expression for a specifier value in the OPEN statement. The character value can contain trailing blanks but not leading or embedded blanks; for example:

```
CHARACTER*6 FINAL /' '/
...
IF (expr) FINAL = 'DELETE'
OPEN (UNIT=1, STATUS='NEW', DISP=FINAL)
```

The following statement creates a new sequential formatted file on unit 1 with the default file name fort.1:

```
OPEN (UNIT=1, STATUS='NEW', ERR=100)
```

The following statement creates a file on magnetic tape:

```
OPEN (UNIT=I, FILE='/dev/rmt8',                                  &
     STATUS='NEW', ERR=14, RECL=1024)
```

The following statement opens the file (created in the previous example) for input:

```
OPEN (UNIT=I, FILE='/dev/rmt8', READONLY, STATUS='OLD',          &
     RECL=1024)
```

The following example opens the existing file /usr/users/someone/test.dat:

```
 OPEN (unit=10, DEFAULTFILE='/usr/users/someone/', FILE='test.dat',
1     FORM='FORMATTED', STATUS='OLD')
```

See Also

- "Unit Specifier" for details on the UNIT control specifier
- "Branch Specifiers" for details on the ERR control specifier
- "I/O Status Specifier" for details on the IOSTAT control specifier
- "INQUIRE Statement" for details on using INQUIRE to get file attributes of existing files
- Your user's guide for details on Fortran I/O status, and details on OPEN statements and file connection

ACCESS Specifier

The ACCESS specifier indicates the access method for the connection of the file. It takes the following form:

ACCESS = *acc*

acc

Is a scalar default character expression that evaluates to one of the following values:

'DIRECT'	Indicates direct access.
'SEQUENTIAL'	Indicates sequential access.
'APPEND'	Indicates sequential access, but the file is positioned at the end-of-file record.

The default is 'SEQUENTIAL'.

There are limitations on record access by file organization and record type. For more information, see your user's guide.

ACTION Specifier

The ACTION specifier indicates the allowed I/O operations for the file connection. It takes the following form:

ACTION = *act*

act

Is a scalar default character expression that evaluates to one of the following values:

'READ'	Indicates that only READ statements can refer to this connection.
'WRITE'	Indicates that only WRITE, DELETE, and ENDFILE statements can refer to this connection.
'READWRITE'	Indicates that READ, WRITE, DELETE, and ENDFILE statements can refer to this connection.

The default is 'READWRITE'. On Windows* systems, the default can be affected by a compiler option. For more information, see your user's guide.

ASSOCIATEVARIABLE Specifier

The ASSOCIATEVARIABLE specifier indicates a variable that is updated after each direct access I/O operation, to reflect the record number of the next sequential record in the file. It takes the following form:

ASSOCIATEVARIABLE = *asv*

asv

Is a scalar integer variable. It cannot be a dummy argument to the routine in which the OPEN statement appears.

Direct access READs, direct access WRITEs, and the FIND, DELETE, and REWRITE statements can affect the value of *asv*.

This specifier is valid only for direct access; it is ignored for other access modes.

BLANK Specifier

The BLANK specifier indicates how blanks are interpreted in a file. It takes the following form:

BLANK = *blnk*

blnk

Is a scalar default character expression that evaluates to one of the following values:

'NULL'	Indicates all blanks are ignored, except for an all-blank field (which has a value of zero).
'ZERO'	Indicates all blanks (other than leading blanks) are treated as zeros.

The default is 'NULL' (for explicitly OPENed files, preconnected files, and internal files). If you specify compiler option F66 (or OPTIONS/NOF77), the default is 'ZERO'. For the correct form of this option, see your user's guide.

If the BN or BZ edit descriptors are specified for a formatted input statement, they supersede the default interpretation of blanks.

See Also

"Blank Editing" for details on the BN and BZ edit descriptors

BLOCKSIZE Specifier

The BLOCKSIZE specifier indicates the physical I/O transfer size for the file. It takes the following form:

BLOCKSIZE = *bks*

bks

Is a scalar numeric expression. If necessary, the value is converted to integer data type before use.

If you specify a nonzero number for *bks*, it is rounded up to a multiple of 512 byte blocks.

If you do not specify BLOCKSIZE or you specify zero for *bks*, the filesystem default is assumed.

BUFFERCOUNT Specifier

The BUFFERCOUNT specifier indicates the number of buffers to be associated with the unit for multibuffered I/O. It takes the following form:

BUFFERCOUNT = *bc*

bc

Is a scalar numeric expression in the range 1 through 127. If necessary, the value is converted to integer data type before use.

The BLOCKSIZE specifier determines the size of each buffer. For example, if BUFFERCOUNT=3 and BLOCKSIZE=2048, the total number of bytes allocated for buffers is 3*2048, or 6144 bytes.

If you do not specify BUFFERCOUNT or you specify zero for *bc*, the default is 1.

See Also

- "The BLOCKSIZE specifier indicates the physical I/O transfer size for the file. It takes the following form:"
- Your user's guide for details on obtaining optimal run-time performance

BUFFERED Specifier

The BUFFERED specifier indicates run-time library behavior following WRITE operations. It takes the following form:

BUFFERED = *bf*

bf

Is a scalar default character expression that evaluates to one of the following values:

'YES'	Requests that the run-time library accumulate output data in its internal buffer, possibly across several WRITE operations, before the data is sent to the file system. Buffering may improve run-time performance for output-intensive applications.
'NO'	Requests that the run-time library send output data to the file system after each WRITE operation.

The default is 'NO'.

If BUFFERED='YES' is specified, the request may or may not be honored, depending on the output device and other file or connection characteristics.

If BLOCKSIZE and BUFFERCOUNT have been specified for OPEN, their product determines the size in bytes of the internal buffer. Otherwise, the default size of the internal buffer is 8192 bytes.

NOTE. *On Windows* systems, the default size of the internal buffer is 1024 bytes if compiler option /fpscomp:general is used.*

The internal buffer will grow to hold the largest single record but will never shrink.

CARRIAGECONTROL Specifier

The CARRIAGECONTROL specifier indicates the type of carriage control used when a file is displayed at a terminal. It takes the following form:

CARRIAGECONTROL = *cc*

cc

Is a scalar default character expression that evaluates to one of the following values:

'FORTRAN'	Indicates normal Fortran interpretation of the first character.
'LIST'	Indicates one line feed between records.
'NONE'	Indicates no carriage control processing.

The default for binary (W*32, W*64) and unformatted files is 'NONE'. The default for formatted files is 'LIST'. However, if you use the compiler option specifying OpenVMS defaults, and the unit is connected to a terminal, the default is 'FORTRAN'.

CONVERT Specifier

The CONVERT specifier indicates a nonnative numeric format for unformatted data. It takes the following form:

CONVERT = *fm*

fm

Is a scalar default character expression that evaluates to one of the following values:

'LITTLE_ENDIAN'[1]	Little endian integer data[2] and IEEE* floating-point data.[3]
'BIG_ENDIAN'[1]	Big endian integer data[2] and IEEE floating-point data.[3]
'CRAY'	Big endian integer data[2] and CRAY* floating-point data of size REAL(8) or COMPLEX(8).

'FDX'	Little endian integer data[2] and VAX* floating-point data of format F_floating for REAL(4) or COMPLEX(4), D_floating for size REAL(8) or COMPLEX(8), and IEEE X_floating for REAL(16) or COMPLEX(16).
'FGX'	Little endian integer data[2] and VAX processor floating-point data of format F_floating for REAL(4) or COMPLEX(4), G_floating for size REAL(8) or COMPLEX(8), and IEEE X_floating for REAL(16) or COMPLEX(16).
'IBM'	Big endian integer data[2] and IBM* System\370 floating-point data of size REAL(4) or COMPLEX(4) (IBM short 4), and size REAL(8) or COMPLEX(8) (IBM long 8).
'VAXD'	Little endian integer data[2] and VAX processor floating-point data of format F_floating for size REAL(4) or COMPLEX(4), D_floating for size REAL(8) or COMPLEX(8), and H_floating for REAL(16) or COMPLEX(16).
'VAXG'	Little endian integer data[2] and VAX processor floating-point data of format F_floating for size REAL(4) or COMPLEX(4), G_floating for size REAL(8) or COMPLEX(8), and H_floating for REAL(16) or COMPLEX(16).
'NATIVE'	No data conversion. This is the default.

1. INTEGER(1) data is the same for little endian and big endian.
2. Of the appropriate size: INTEGER(1), INTEGER(2), INTEGER(4), or INTEGER(8).
3. Of the appropriate size and type: REAL(4), REAL(8), REAL(16), COMPLEX(4), COMPLEX*(8), or COMPLEX(16).

You can use CONVERT to specify multiple formats in a single program, usually one format for each specified unit number.

When reading a nonnative format, the nonnative format on disk is converted to native format in memory. If a converted nonnative value is outside the range of the native data type, a run-time message appears.

There are other ways to specify numeric format for unformatted files: you can specify an environment variable, the compiler option specifying CONVERT, or OPTIONS/CONVERT. The following shows the order of precedence:

Method Used	Precedence
An environment variable	Highest
OPEN (CONVERT=)	.
OPTIONS/CONVERT	.

Method Used	Precedence
The CONVERT compiler option	Lowest

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

- Chapter 3, "Data Types, Constants, and Variables" and your user's guide for details on supported ranges for data types
- Your user's guide for details on compiler options, and details on using environment variables to specify CONVERT options

DEFAULTFILE Specifier

The DEFAULTFILE specifier indicates a default file pathname string. It takes the following form:

DEFAULTFILE = *def*

def

Is a character expression indicating a default file pathname string.

The default file pathname string is used primarily when accepting file pathnames interactively. File pathnames known to a user program normally appear in the FILE specifier.

DEFAULTFILE supplies a value to the Fortran I/O system that is prefixed to the name that appears in FILE.

If *def* does not end in a slash (/), a slash is added.

If DEFAULTFILE is omitted, the Fortran I/O system uses the current working directory.

DELIM Specifier

The DELIM specifier indicates what characters (if any) are used to delimit character constants in list-directed and namelist output. It takes the following form:

DELIM = *del*

del

Is a scalar default character expression that evaluates to one of the following values:

APOSTROPHE'	Indicates apostrophes delimit character constants. All internal apostrophes are doubled.
'QUOTE'	Indicates quotation marks delimit character constants. All internal quotation marks are doubled.
'NONE'	Indicates character constants have no delimiters. No internal apostrophes or quotation marks are doubled.

The default is 'NONE'.

The DELIM specifier is only allowed for files connected for formatted data transfer; it is ignored during input.

DISPOSE Specifier

The DISPOSE (or DISP) specifier indicates the status of the file after the unit is closed. It takes one of the following forms:

DISPOSE = *dis*

DISP = *dis*

dis

Is a scalar default character expression that evaluates to one of the following values:

'KEEP' or 'SAVE'	Retains the file after the unit closes.
'DELETE'	Deletes the file after the unit closes.
'PRINT'[1]	Submits the file to the line printer spooler and retains it.
'PRINT/DELETE'[1]	Submits the file to the line printer spooler and then deletes it.
'SUBMIT'	Forks a process to execute the file.
'SUBMIT/DELETE'	Forks a process to execute the file, and then deletes the file after the fork is completed.

1. Use only on sequential files.

The default is 'DELETE' for scratch files. For all other files, the default is 'KEEP'.

FILE Specifier

The FILE specifier indicates the name of the file to be connected to the unit. It takes the following form:

FILE = *name*

name

Is a character or numeric expression.

The *name* can be any pathname allowed by the operating system.

Any trailing blanks in the name are ignored.

If the following conditions occur:

- FILE is omitted
- The unit is not connected to a file
- STATUS='SCRATCH' is not specified
- The corresponding FORT*n* environment variable is not set for the unit number

Intel® Fortran generates a file name in the form fort.*n*, where *n* is the logical unit number. On Windows systems, if compiler option /fpscomp:general is specified, omitting FILE implies STATUS='SCRATCH'.

If the file name is stored in a numeric scalar or array, the name must consist of ASCII characters terminated by an ASCII null character (zero byte). However, if it is stored in a character scalar or array, it must not contain a zero byte.

See Also

- Your user's guide for details on default file name conventions
- The appropriate manual in your system documentation set for details on allowable file pathnames

FORM Specifier

The FORM specifier indicates whether the file is being connected for formatted, unformatted, or binary (W*32, W*64) data transfer. It takes the following form:

FORM = *fm*

fm

Is a scalar default character expression that evaluates to one of the following values:

'FORMATTED'	Indicates formatted data transfer.
'UNFORMATTED'	Indicates unformatted data transfer.

'BINARY'	Indicates binary data transfer.

The default is 'FORMATTED' for sequential access files, and 'UNFORMATTED' for direct access files.

IOFOCUS Specifier (W*32, W*64)

The IOFOCUS specifier indicates whether a particular unit is the active window in a QuickWin application. It takes the following form:

IOFOCUS = *iof*

iof

Is a scalar default logical expression that evaluates to one of the following values:

.TRUE.	Indicates the QuickWin child window is the active window.
.FALSE.	Indicates the QuickWin child window is not the active window.

If unit '*' is specified, the default is .FALSE.; otherwise, the default is .TRUE..

A value of .TRUE. causes a call to FOCUSQQ immediately before any READ, WRITE, or PRINT statement to that window.

See Also

The Windows system online help for details on QuickWin applications.

MAXREC Specifier

The MAXREC specifier indicates the maximum number of records that can be transferred from or to a direct access file while the file is connected. It takes the following form:

MAXREC = *mr*

mr

Is a scalar numeric expression. If necessary, the value is converted to integer data type before use.

The default is an unlimited number of records.

MODE Specifier

MODE is a nonstandard synonym for ACTION (see "ACTION Specifier").

NAME Specifier

NAME is a nonstandard synonym for FILE (see "FILE Specifier").

ORGANIZATION Specifier

The ORGANIZATION specifier indicates the internal organization of the file. It takes the following form:

ORGANIZATION = *org*

org

Is a scalar default character expression that evaluates to one of the following values:

'SEQUENTIAL'	Indicates a sequential file.
'RELATIVE'	Indicates a relative file.

The default is 'SEQUENTIAL'.

PAD Specifier

The PAD specifier indicates whether a formatted input record is padded with blanks when an input list and format specification requires more data than the record contains.

The PAD specifier takes the following form:

PAD = *pd*

pd

Is a scalar default character expression that evaluates to one of the following values:

'YES'	Indicates the record will be padded with blanks when necessary.
'NO'	Indicates the record will not be padded with blanks. The input record must contain the data required by the input list and format specification.

The default is 'YES'.

This behavior is different from FORTRAN 77, which never pads short records with blanks. For example, consider the following:

```
READ (5,'(I5)') J
```

If you enter 123 followed by a carriage return, FORTRAN 77 turns the I5 into an I3 and J is assigned 123.

However, Intel Fortran pads the 123 with 2 blanks unless you explicitly open the unit with PAD='NO'.

You can override blank padding by explicitly specifying the BN edit descriptor.

The PAD specifier is ignored during output.

POSITION Specifier

The POSITION specifier indicates the position of a file connected for sequential access. It takes the following form:

POSITION = *pos*

pos

Is a scalar default character expression that evaluates to one of the following values:

'ASIS'	Indicates the file position is unchanged if the file exists and is already connected. The position is unspecified if the file exists but is not connected.
'REWIND'	Indicates the file is positioned at its initial point.
'APPEND'	Indicates the file is positioned at its terminal point (or before its end-of-file record, if any).

The default is 'ASIS'. (On Fortran I/O systems, this is the same as 'REWIND'.)

A new file (whether specified as new explicitly or by default) is always positioned at its initial point.

See Also

Your user's guide for details on record position, advancement, and transfer

READONLY Specifier

The READONLY specifier indicates only READ statements can refer to this connection. It takes the following form:

READONLY

READONLY is similar to specifying ACTION='READ', but READONLY prevents deletion of the file if it is closed with STATUS='DELETE' in effect.

The Fortran I/O system's default privileges for file access are READWRITE. If access is denied, the I/O system automatically retries accessing the file for READ access.

However, if you use the compiler option specifying OpenVMS defaults, the I/O system does not retry accessing for READ access. So, run-time I/O errors can occur if the file protection does not permit WRITE access. To prevent such errors, if you wish to read a file for which you do not have write access, specify READONLY.

RECL Specifier

The RECL specifier indicates the length of each record in a file connected for direct access, or the maximum length of a record in a file connected for sequential access.

The RECL specifier takes the following form:

RECL = *rl*

rl

Is a positive numeric expression indicating the length of records in the file. If necessary, the value is converted to integer data type before use.

If the file is connected for formatted data transfer, the value must be expressed in bytes (characters). Otherwise, the value is expressed in 4-byte units (longwords).

If the file is connected for unformatted data transfer, the value can be expressed in bytes if compiler option ASSUME BYTERECL is specified. (For the correct form of this option, see your user's guide.)

Except for segmented records, the *rl* is the length for record data only, it does not include space for control information.

The length specified is interpreted depending on the type of records in the connected file, as follows:

- For segmented records, RECL indicates the maximum length for any segment (including the four bytes of control information).
- For fixed-length records, RECL indicates the size of each record; it *must* be specified. If the records are unformatted, the size must be expressed as an even multiple of four.

  You can use the RECL specifier in an INQUIRE statement to get the record length before opening the file (see "RECL Specifier").
- For variable-length records, RECL indicates the maximum length for any record.

If you read a fixed-length file with a record length different from the one used to create the file, indeterminate results can occur.

The maximum length for *rl* depends on the record type and the setting of the CARRIAGECONTROL specifier, as shown in Table 12-2.

Table 12-2 Maximum Record Lengths (RECL)

Record Type	CARRIAGECONTROL	Formatted (size in bytes)
Fixed-length	'NONE'	2147483647 (2**31–1)[1]
Variable-length	'NONE'	2147483640 (2**31–8)
Segmented	'NONE'	32764 (2**15–4)
Stream	'NONE'	2147483647 (2**31–1)

Table 12-2 Maximum Record Lengths (RECL)

Record Type	CARRIAGECONTROL	Formatted (size in bytes)
Stream_CR	'LIST'	2147483647 (2**31–1)
	'FORTRAN'	2147483646 (2**31–2)
Stream_LF	'LIST'	2147483647 (2**31–1)[2]
	'FORTRAN'	2147483646 (2**31–2)

1. Subtract 1 if the compiler option specifying OpenVMS defaults is used.
2. L*X only

The default value depends on the setting of the RECORDTYPE specifier, as shown in Table 12-3.

Table 12-3 Default Record Lengths (RECL)

RECORDTYPE	RECL Value
'FIXED'	None; value must be explicitly specified.
All other settings	132 bytes for formatted records; 510 longwords for unformatted records.

RECORDSIZE Specifier

RECORDSIZE is a nonstandard synonym for RECL (see "RECL Specifier").

RECORDTYPE Specifier

The RECORDTYPE specifier indicates the type of records in a file. It takes the following form:

RECORDTYPE = *typ*

typ

Is a scalar default character expression that evaluates to one of the following values:

'FIXED'	Indicates fixed-length records.
'VARIABLE'	Indicates variable-length records.
'SEGMENTED'	Indicates segmented records.
'STREAM'	Indicates stream-type variable length records.
'STREAM_LF'	Indicates stream-type variable length records, terminated with a line feed.
'STREAM_CR'	Indicates stream-type variable length records, terminated with a carriage-return.

When you open a file, default record types are as follows:

'FIXED'	For relative files
'FIXED'	For direct access sequential files
'STREAM_LF'	For formatted sequential access files
'VARIABLE'	For unformatted sequential access files

A *segmented record* is a logical record consisting of segments that are physical records. Since the length of a segmented record can be greater than 65,535 bytes, only use segmented records for unformatted sequential access to disk or raw magnetic tape files.

Files containing segmented records can be accessed only by unformatted sequential data transfer statements.

If an output statement does not specify a full record for a file containing fixed-length records, the following occurs:

- In formatted files, the record is filled with blanks
- In unformatted files, the record is filled with zeros

See Also

Your user's guide for details on record types and file organization

SHARE Specifier (W*32, W*64)

The SHARE specifier indicates whether file locking is implemented while the unit is open. It takes the following form:

SHARE = *shr*

shr

Is a scalar default character expression that evaluates to one of the following values:

'DENYRW'	Indicates deny-read/write mode. No other process can open the file.
'DENYWR'	Indicates deny-write mode. No process can open the file with write access.
'DENYRD'	Indicates deny-read mode. No process can open the file with read access.
'DENYNONE'	Indicates deny-none mode. Any process can open the file in any mode.

The default is 'DENYWR'. However, if compiler option /fpscomp=general or the SHARED specifier is used, the default is 'DENYNONE'.

See Also

Your user's guide for details on limitations on record access

SHARED Specifier

The SHARED specifier indicates that the file is connected for shared access by more than one program executing simultaneously. It takes the following form:

SHARED

On Linux* systems, shared access is the default for the Fortran I/O system. On Windows* systems, it is the default if SHARED or compiler option /fpscomp:general is specified.

See Also

Your user's guide for details on file sharing

STATUS Specifier

The STATUS specifier indicates the status of a file when it is opened. It takes the following form:

STATUS = *sta*

sta

Is a scalar default character expression that evaluates to one of the following values:

'OLD'	Indicates an existing file.
'NEW'	Indicates a new file; if the file already exists, an error occurs. Once the file is created, its status changes to 'OLD'.
'SCRATCH'	Indicates a new file that is unnamed (called a scratch file). When the file is closed or the program terminates, the scratch file is deleted.
'REPLACE'	Indicates the file replaces another. If the file to be replaced exists, it is deleted and a new file is created with the same name. If the file to be replaced does not exist, a new file is created and its status changes to 'OLD'.
'UNKNOWN'	Indicates the file may or may not exist. If the file does not exist, a new file is created and its status changes to 'OLD'.

Scratch files go into a temporary directory and are visible while they are open. (For more information, see your user's guide.)

The default is 'UNKNOWN'. This is also the default if you implicitly open a file by using WRITE. However, if you implicitly open a file using READ, the default is 'OLD'. If you specify compiler option F66 (or OPTIONS /NOF77), the default is 'NEW'. For the correct form of this option, see your user's guide.

NOTE. *The* STATUS *specifier can also appear in* CLOSE *statements to indicate the file's status after it is closed. However, in* CLOSE *statements the status values are the same as those listed for the* DISPOSE *specifier (see* "DISPOSE Specifier"*).*

TITLE Specifier (W*32, W*64)

The TITLE specifier indicates the name of a child window in a QuickWin application. It takes the following form:

TITLE = *name*

name

Is a character expression.

If TITLE is specified in a non-Quickwin application, a run-time error occurs.

See Also

The Windows system online help for details on QuickWin applications.

TYPE Specifier

TYPE is a nonstandard synonym for STATUS (see "STATUS Specifier").

USEROPEN Specifier

The USEROPEN specifier indicates a user-written external function that controls the opening of the file. It takes the following form:

USEROPEN = *function-name*

function-name

Is the name of the user-written function to receive control.

The function must be declared in a previous EXTERNAL statement; if it is typed, it must be of type INTEGER(4) (INTEGER*4).

The USEROPEN specifier lets experienced users use additional features of the operating system that are not normally available in Fortran.

See Also

Your user's guide for details on user-supplied functions to use with USEROPEN, including examples

REWIND Statement

The REWIND statement positions a sequential or direct access file at the beginning of the file (the initial point). It takes one of the following forms:

REWIND ([UNIT=]*io-unit* [, ERR=*label*] [, IOSTAT=*i-var*])

REWIND *io-unit*

io-unit

Is an external unit specifier.

label

Is the label of the branch target statement that receives control if an error occurs.

i-var

Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs.

Rules and Behavior

The unit number must refer to a file on disk or magnetic tape, and the file must be open for sequential, direct, or append access.

If a REWIND is done on a direct access file, the NEXTREC specifier is assigned a value of 1.

If a file is already positioned at the initial point, a REWIND statement has no effect.

If a REWIND statement is specified for a unit that is not open, it has no effect.

Examples

The following statement repositions the file connected to I/O unit 3 to the beginning of the file:

```
REWIND 3
```

Consider the following statement:

```
REWIND (UNIT=9, IOSTAT=IOS, ERR=10)
```

This statement positions the file connected to unit 9 at the beginning of the file. If an error occurs, control is transferred to the statement labeled 10, and a positive integer is stored in variable IOS.

See Also

- "Unit Specifier" for details on the UNIT control specifier
- "Branch Specifiers" for details on the ERR control specifier
- "I/O Status Specifier" for details on the IOSTAT control specifier
- Your user's guide for details on record position, advancement, and transfer

UNLOCK Statement

The UNLOCK statement frees a record in a relative or sequential file that was locked by a previous READ statement.

The UNLOCK statement takes one of the following forms:

UNLOCK ([UNIT=]*io-unit* [, ERR=*label*] [, IOSTAT=*i-var*])
UNLOCK *io-unit*

io-unit

Is an external unit specifier.

label

Is the label of the branch target statement that receives control if an error occurs.

i-var

Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs.

If no record is locked, the UNLOCK statement has no effect.

Examples

The following statement frees any record previously read and locked in the file connected to I/O unit 4:

```
UNLOCK 4
```

Consider the following statement:

```
UNLOCK (UNIT=9, IOSTAT=IOS, ERR=10)
```

This statement frees any record previously read and locked in the file connected to unit 9. If an error occurs, control is transferred to the statement labeled 10, and a positive integer is stored in variable IOS.

See Also

- "Unit Specifier" for details on the UNIT control specifier
- "Branch Specifiers" for details on the ERR control specifier

- "I/O Status Specifier" for details on the IOSTAT control specifier
- Your user's guide for details on shared files and locked records

Compilation Control Statements

13

In addition to specifying options on the compiler command line, you can specify the following statements in a program unit to influence compilation:

- The "INCLUDE Statement"
  Incorporates external source code into programs.
- The "OPTIONS Statement"
  Sets options usually specified in the compiler command line. OPTIONS statement settings override command line options.

INCLUDE Statement

The INCLUDE statement directs the compiler to stop reading statements from the current file and read statements in an included file or text module.

The INCLUDE statement takes the following form:

INCLUDE '*file-name* [/[NO]LIST]'

file-name

Is a character string specifying the name of the file to be included; it must not be a named constant.

The form of the file name must be acceptable to the operating system, as described in your system documentation.

/[NO]LIST

Specifies whether the incorporated code is to appear in the compilation source listing. In the listing, a number precedes each incorporated statement. The number indicates the "include" nesting depth of the code. The default is /NOLIST. /LIST and /NOLIST must be spelled completely.

You can only use /[NO]LIST if you specify the compiler option that sets OpenVMS defaults.

Rules and Behavior

An INCLUDE statement can appear anywhere within a scoping unit. The statement can span more than one source line, but no other statement can appear on the same line. The source line cannot be labeled.

An included file or text module cannot begin with a continuation line, and each Fortran statement must be completely contained within a single file.

An included file or text module can contain any source text, but it cannot begin or end with an incomplete Fortran statement.

The included statements, when combined with the other statements in the compilation, must satisfy the statement-ordering restrictions shown in Figure 2-1.

Included files or text modules can contain additional INCLUDE statements, but they must not be recursive. INCLUDE statements can be nested until system resources are exhausted.

When the included file or text module completes execution, compilation resumes with the statement following the INCLUDE statement.

Example

In Example 13-1, a file named COMMON.FOR (in the current working directory) is included and read as input.

Example 13-1 Including Text from a File

Main Program File

```
PROGRAM
  INCLUDE 'COMMON.FOR'
  REAL, DIMENSION(M) :: Z
  CALL CUBE
  DO I = 1, M
    Z(I) = X(I) + SQRT(Y(I))
    ...
  END DO
END
```

COMMON.FOR File

```
INTEGER, PARAMETER :: M=100
REAL, DIMENSION(M) :: X, Y
COMMON X, Y
```

Example 13-1 Including Text from a File

```
SUBROUTINE CUBE
  INCLUDE 'COMMON.FOR'
  DO I=1,M
    X(I) = Y(I)**3
  END DO
  RETURN
END
```

The file COMMON.FOR defines a named constant M, and defines arrays X and Y as part of blank common.

See Also

Your user's guide for details on compiler options

OPTIONS Statement

The OPTIONS statement overrides or confirms the compiler options in effect for a program unit. It takes the following form:

OPTIONS *option* [*option*...]

option

Is one of the following:

/ASSUME =	[NO]UNDERSCORE
/CHECK =	ALL
	[NO]BOUNDS
	[NO]OVERFLOW
	NONE
/NOCHECK	
/CONVERT =	BIG_ENDIAN
	CRAY
	FDX
	FGX
	IBM
	LITTLE_ENDIAN
	NATIVE
	VAXD
	VAXG
/NOEXTENDSOURCE	

/NOF77
/NOI4
/NORECURSIVE

Note that an option must always be preceded by a slash (/).

Some OPTIONS statement options are equivalent to compiler options.

Rules and Behavior

The OPTIONS statement must be the first statement in a program unit, preceding the PROGRAM, SUBROUTINE, FUNCTION, MODULE, and BLOCK DATA statements.

OPTIONS statement options override compiler options, but only until the end of the program unit for which they are defined. If you want to override compiler options in another program unit, you must specify the OPTIONS statement before that program unit.

Examples

The following are valid OPTIONS statements:

```
OPTIONS /CHECK=ALL/F77
OPTIONS /I4
```

See Also

Your user’s guide for details on compiler options

Directive Enhanced Compilation

14

Directive enhanced compilation is performed by using compiler directives. Compiler directives are special commands that let you perform various tasks during compilation. They are similar to compiler options, but can provide more control within your program.

Compiler directives are preceded by a special prefix that identifies them to the compiler.

This chapter contains information on the following topics:

- "Syntax Rules for Compiler Directives"
- "General Compiler Directives"

  Perform general-purpose tasks during compilation.
- "OpenMP* Fortran Compiler Directives"

  Specify parallel regions and characteristics of data and threads for shared access of data in memory.

Syntax Rules for Compiler Directives

The following syntax rules apply to all general and OpenMP* Fortran compiler directives. You must follow these rules precisely to compile your program properly and obtain meaningful results.

A directive prefix (tag) takes one of the following forms:

General compiler directives:

*c*DEC$

OpenMP Fortran compiler directives:

c$OMP

c

Is one of the following: C (or c), !, or *.

The following prefix forms can be used in place of *c*DEC$: *c*DIR$ or *c*MS$.

The following are source form rules for directive prefixes:

- Prefixes beginning with C (or c) and * are only allowed in fixed and tab source forms.

  In these source forms, the prefix must appear in columns 1 through 5; column 6 must be a blank or tab[1]. From column 7 on, blanks are insignificant, so the directive can be positioned anywhere on the line after column 6.
- Prefixes beginning with ! are allowed in all source forms.

  The prefix can appear in any valid column, but it cannot be preceded by any nonblank characters on the same line. It can only be preceded by whitespace.

A compiler directive ends in column 72 (or column 132, if a compiler option is specified).

General compiler directives cannot be continued. OpenMP Fortran directives can be continued.

A comment can follow a compiler directive on the same line.

Additional Fortran statements (or directives) cannot appear on the same line as the compiler directive.

Compiler directives cannot appear within a continued Fortran statement.

If a blank common is used in a compiler directive, it must be specified as two slashes (/ /).

If the source line starts with a valid directive prefix but the directive is not recognized, the compiler prints an informational message and ignores the line.

General Compiler Directives

Intel® Fortran provides several general-purpose compiler directives to perform tasks during compilation. You do not need to specify a compiler option to enable general directives.

This section describes the following directives:

- "ALIAS Directive"

  Specifies an alternate external name to be used when referring to external subprograms.
- "ATTRIBUTES Directive"

  Specifies properties for data objects and procedures.
- "DECLARE and NODECLARE Directives"

  Generate or disable warnings for variables that have been used but not declared.
- "DEFINE and UNDEFINE Directives"

  Define (or undefine) a symbolic variable whose existence (or value) can be tested during conditional compilation.

1. Except for prefix *c*MS$

- "DISTRIBUTE POINT Directive"
  Specifies distribution for a DO loop.
- "FIXEDFORMLINESIZE Directive"
  Sets the line length for fixed-form source code.
- "FREEFORM and NOFREEFORM Directives"
  Specify free-format or fixed-format source code.
- "IDENT Directive"
  Specifies an identifier for an object module.
- "IF and IF DEFINED Directives"
  Specify a conditional compilation construct.
- "INTEGER Directive"
  Specifies the default integer kind.
- "IVDEP Directive"
  Assists the compiler's dependence analysis of iterative DO loops.
- "LOOP COUNT Directive"
  Specifies the loop count for a DO loop; this assists the optimizer.
- "MESSAGE Directive"
  Specifies a character string to be sent to the standard output device during the first compiler pass.
- "OBJCOMMENT Directive"
  Specifies a library search path in an object file.
- "OPTIONS Directive"
  Affects data alignment and warnings about data alignment.
- "PACK Directive"
  Specifies the memory starting addresses of derived-type items.
- "PARALLEL and NOPARALLEL Directives"
  Enables or disables auto-parallelization for an immediately following DO loop.
- "PREFETCH and NOPREFETCH Directives"
  Enables or disables a data prefetch from memory.
- "PSECT Directive"
  Modifies certain characteristics of a common block.
- "REAL Directive"
  Specifies the default real kind.

- "STRICT and NOSTRICT Directives"
  Disables or enables language features not found in the language standard specified on the command line (Fortran 95 or Fortran 90).
- "SWP and NOSWP Directives (i64 only)"
  Enables or disables software pipelining for a DO loop.
- "TITLE and SUBTITLE Directives"
  Specifies a title or subtitle for a listing header.
- "UNROLL and NOUNROLL Directives"
  Tells the compiler's optimizer how many times to unroll a DO loop or disables the unrolling of a DO loop.
- "VECTOR ALIGNED and VECTOR UNALIGNED Directives (i32 only)"
  Specifies that all data in a DO loop is aligned or not aligned.
- "VECTOR ALWAYS and NOVECTOR Directives (i32 only)"
  Enables or disables vectorization of a DO loop.
- "VECTOR NONTEMPORAL Directive (i32 only)"
  Enables streaming storage.

Rules for General Directives that Affect DO Loops

This table lists the general directives that affect DO loops:

DISTRIBUTE POINT	NOUNROLL	VECTOR ALIGNED[1]
IVDEP	NOVECTOR[1]	VECTOR ALWAYS[1]
LOOP COUNT	PARALLEL	VECTOR NONTEMPORAL[1]
NOPARALLEL	PREFETCH	VECTOR UNALIGNED[1]
NOPREFETCH	SWP[2]	
NOSWP[2]	UNROLL	

1. i32 only
2. i64 only

The following rules apply to all of these directives:

- The directive must precede the DO statement for each DO loop it affects.
- No source code lines, other than the following, can be placed between the directive statement and the DO statement:
  — One of the other general directives that affect DO loops
  — An OpenMP* Fortran PARALLEL DO directive

— Comment lines
— Blank lines

Other rules may apply to these directives. For more information, see the description of each directive.

ALIAS Directive

The ALIAS directive lets you specify an alternate external name to be used when referring to external subprograms. This can be useful when compiling applications written for other platforms that have different naming conventions.

The ALIAS directive takes the following form:

*c*DEC$ ALIAS *internal-name*, *external-name*

c

Is one of the following: C (or c), !, or * (see "Syntax Rules for Compiler Directives").

internal-name

Is the name of the subprogram as used in the current program unit.

external-name

Is a name, or a character constant delimited by apostrophes or quotation marks.

Rules and Behavior

If a name is specified, the name (in uppercase) is used as the external name for the specified *internal-name*. If a character constant is specified, it is used as is; the string is not changed to uppercase, nor are blanks removed.

The ALIAS directive affects only the external name used for references to the specified *internal-name*.

Names that are not acceptable to the linker will cause link-time errors.

See Also

- `ld(1)` for details on the linker for Linux* systems
- The online help on the linker for details on the linker for Windows* systems

ATTRIBUTES Directive

The ATTRIBUTES directive lets you specify properties for data objects and procedures. It takes the following form:

*c*DEC$ ATTRIBUTES *att* [, *att*]... :: *object* [, *object*]...

c

Is one of the following: C (or c), !, or * (see "Syntax Rules for Compiler Directives").

att

Is one of the following properties (or options):

ALIAS	DEFAULT	NO_ARG_CHECK
ALIGN	DLLEXPORT[1]	NOINLINE
ALLOCATABLE	DLLIMPORT[1]	NOMIXED_STR_LEN_ARG
ALLOW_NULL	EXTERN	REFERENCE
ARRAY_VISUALIZER[2]	FORCEINLINE	STDCALL
C	IGNORE_LOC	VALUE
DECORATE	INLINE	VARYING

1. W*32, W*64
2. W*32 only

object

Is the name of a data object or procedure.

The following table shows which ATTRIBUTES properties can be used with various objects:

Property	Variable and Array Declarations	Common Block Names[1]	Subprogram Specification and EXTERNAL Statements
ALIAS	No	Yes	Yes
ALIGN	Yes	No	No
ALLOCATABLE	Yes[2]	No	No
ALLOW_NULL	Yes	No	No
ARRAY_VISUALIZER	Yes[2]	No	No
C	No	Yes	Yes
DECORATE	No	No	Yes
DEFAULT	No	Yes	Yes
DLLEXPORT	Yes [3]	Yes	Yes
DLLIMPORT	Yes	Yes	Yes
EXTERN	Yes	No	No
FORCEINLINE	No	No	Yes
IGNORE_LOC	Yes[4]	No	No
INLINE	No	No	Yes

Property	Variable and Array Declarations	Common Block Names[1]	Subprogram Specification and EXTERNAL Statements
NO_ARG_CHECK	Yes	No	Yes[5]
NOINLINE	No	No	Yes
NOMIXED_STR_LEN_ARG	No	No	Yes
REFERENCE	Yes	No	Yes
STDCALL	No	Yes	Yes
VALUE	Yes	No	No
VARYING	No	No	Yes

1. A common block name is specified as [/]common-block-name[/].
2. This property can only be applied to arrays.
3. Module-level variables and arrays only
4. This property can only be applied to interface blocks.
5. This property cannot be applied to EXTERNAL statements.

These properties can be used in function and subroutine definitions, in type declarations, and with the INTERFACE and ENTRY statements.

Properties applied to entities available through use or host association are in effect during the association. For example, consider the following:

```
MODULE MOD1
   INTERFACE
     SUBROUTINE SUB1
     !DEC$ ATTRIBUTES C, ALIAS:'othername' :: NEW_SUB
     END SUBROUTINE
   END INTERFACE
   CONTAINS
     SUBROUTINE SUB2
     CALL NEW_SUB
     END SUBROUTINE
END MODULE
```

In this case, the call to NEW_SUB within SUB2 uses the C and ALIAS properties specified in the interface block.

The following are ATTRIBUTES properties (or options):

- "ATTRIBUTES ALIAS"
- "ATTRIBUTES ALIGN"
- "ATTRIBUTES ALLOCATABLE"

- "ATTRIBUTES ALLOW_NULL"
- "ATTRIBUTES ARRAY_VISUALIZER (W*32 only)"
- "ATTRIBUTES C and STDCALL"
- "ATTRIBUTES DECORATE"
- "ATTRIBUTES DEFAULT"
- "ATTRIBUTES DLLEXPORT and DLLIMPORT (W*32, W*64)"
- "ATTRIBUTES EXTERN"
- "ATTRIBUTES IGNORE_LOC"
- "ATTRIBUTES INLINE, NOINLINE, and FORCEDINLINE"
- "ATTRIBUTES NO_ARG_CHECK"
- "ATTRIBUTES NOMIXED_STR_LEN_ARG"
- "ATTRIBUTES REFERENCE and VALUE"
- "ATTRIBUTES VARYING"

Properties C, STDCALL, REFERENCE, VALUE, and VARYING affect the calling conventions of routines:

- You can specify C, STDCALL, REFERENCE, and VARYING for an entire routine.
- You can specify VALUE and REFERENCE for individual arguments.

See Also

Your user's guide for details on using the cDEC$ ATTRIBUTES directive

ATTRIBUTES ALIAS

The ATTRIBUTES directive option ALIAS specifies an alternate external name to be used when referring to external subprograms. It takes the following form:

*c*DEC$ ATTRIBUTES ALIAS: *external-name* :: *subprogram*

external-name

Is a character constant delimited by apostrophes or quotation marks. The character constant is used as is; the string is not changed to uppercase, nor are blanks removed.

subprogram

Is an external subprogram.

The ALIAS property overrides the C (and STDCALL) property. If both C and ALIAS are specified for a subprogram, the subprogram is given the C calling convention, but not the C naming convention. It instead receives the name given for ALIAS, with no modifications.

ALIAS cannot be used with internal procedures, and it cannot be applied to dummy arguments.

The following example gives the subroutine happy the name "_OtherName@4" outside this scoping unit:

```
INTERFACE
  SUBROUTINE happy(i)
    !DEC$ ATTRIBUTES STDCALL, DECORATE, ALIAS:'OtherName' :: happy
    INTEGER i
  END SUBROUTINE
END INTERFACE
```

*c*DEC$ ATTRIBUTES ALIAS has the same effect as the "ALIAS Directive".

ATTRIBUTES ALIGN

The ATTRIBUTES directive option ALIGN specifies the byte alignment for a variable. It takes the following form:

*c*DEC$ ATTRIBUTES ALIGN: *n*:: *var*

n

Is the number of bytes for the alignment boundary.

var

Is the variable to be aligned.

ATTRIBUTES ALLOCATABLE

The ATTRIBUTES directive option ALLOCATABLE is provided for compatibility with older programs. It lets you delay allocation of storage for a particular declared entity until some point at run time when you explicitly call a routine that dynamically allocates storage for the entity. The ALLOCATABLE option takes the following form:

*c*DEC$ ATTRIBUTES ALLOCATABLE :: *entity*

entity

Is the name of the entity that should have allocation delayed.

The recommended method for dynamically allocating storage is to use the "ALLOCATABLE Attribute and Statement".

ATTRIBUTES ALLOW_NULL

The ATTRIBUTES directive option ALLOW_NULL enables a corresponding dummy argument to pass a NULL pointer (defined by a zero or the NULL intrinsic) by value for the argument. It takes the following form:

*c*DEC$ ATTRIBUTES ALLOW_NULL :: *arg*

arg

Is the name of the argument.

ALLOW_NULL is only valid if the REFERENCE property is also specified; otherwise, it has no effect.

ATTRIBUTES ARRAY_VISUALIZER (W*32 only)

The ATTRIBUTES directive option ARRAY_VISUALIZER enhances the performance of the Intel® Array Visualizer. It takes the following form:

*c*DEC$ ATTRIBUTES ARRAY_VISUALIZER :: *array*

array

Is the array to be viewed using the Intel® Array Viewer.

When declaring allocatable arrays to be viewed using the Array Viewer, this option can improve the performance of the Array Viewer. For example:

```
real(4), allocatable :: MyArray(:, :)
!DEC$ ATTRIBUTES array_visualizer :: MyArray
```

When this property is used, array memory is shared between the Array Viewer and your application. Otherwise, the array data is copied during each faglUpdate call.

The ARRAY_VISUALIZER option is not useful unless the array is viewed in the Array Visualizer by using fagl* calls.

For more information on fagl* routines, see your online documentation for the Array Visualizer.

ATTRIBUTES C and STDCALL

The ATTRIBUTES directive options C and STDCALL specify how data is to be passed when you use routines written in C or assembler with FORTRAN or Fortran 95/90 routines. They take the following forms:

*c*DEC$ ATTRIBUTES C :: *object* [, *object*] ...

*c*DEC$ ATTRIBUTES STDCALL :: *object* [, *object*] ...

object

Is the name of a data object or procedure.

On IA-32 processors, C and STDCALL have slightly different meanings; on all other platforms, they are interpreted as synonyms.

When applied to a subprogram, these propertiess define the subprogram as having a specific set of calling conventions.

The following table summarizes the differences between the calling conventions:

Convention	C[1]	STDCALL[1]	Default[2]
Arguments passed by value	Yes	Yes	No
Case of external subprogram names	L*X: Lowercase W*32, W*64: Lowercase	L*X: Lowercase W*32, W*64: Lowercase	L*X: Lowercase W*32, W*64: Uppercase
L*X only:			
Trailing underscore added	No	No	Yes[3]
W*32, W*64 only:			
Leading underscore added	Yes	Yes	Yes[4]
Number of arguments added	No	Yes	No
Caller stack cleanup	Yes	No	Yes
Variable number of arguments	Yes	No	Yes

1. C and STDCALL are synonyms on L*X systems.
2. The Fortran 95/90 calling convention
3. If there are one or more underscores in the external name, two trailing underscores are added; if there are no underscores, one is added.
4. W*32 only

If C or STDCALL is specified for a subprogram, arguments (except for arrays and characters) are passed by value. Subprograms using standard Fortran 95/90 conventions pass arguments by reference.

On IA-32 processors, an underscore (_) is placed at the beginning of the external name of a subprogram. If STDCALL is specified, an at sign (@) followed by the number of argument bytes being passed is placed at the end of the name. For example, a subprogram named SUB1 that has three INTEGER(4) arguments and is defined with STDCALL is assigned the external name _sub1@12.

Character arguments are passed as follows:

- By default:
  - On Linux systems, hidden lengths are put at the end of the argument list.
  - On Windows systems, hidden lengths immediately follow the variable. You can get the Linux behavior by specifying the appropriate compiler option.
- If C or STDCALL (only) is specified:

  On all systems, the first character of the string is passed (and padded with zeros out to INTEGER(4) length).
- If C or STDCALL is specified, and REFERENCE is specified for the argument:

  On all systems, the string is passed with no length.

- If C or STDCALL is specified, and REFERENCE is specified for the routine (but REFERENCE is *not* specified for the argument, if any):

  On all systems, the string is passed with the length.

For more details, see information on mixed-language programming in your user's guide. See also the description of REFERENCE in "ATTRIBUTES REFERENCE and VALUE".

ATTRIBUTES DECORATE

The ATTRIBUTES directive option DECORATE specifies that the external name used in *c*DEC$ ALIAS or *c*DEC$ ATTRIBUTES ALIAS should have the prefix and postfix decorations performed on it that are associated with the calling mechanism that is in effect. These are the same decorations performed on the procedure name when ALIAS is not specified.

The DECORATE option takes the following form:

*c*DEC$ ATTRIBUTES DECORATE :: *exname*

exname

Is an external name.

The case of the ALIAS external name is not modified.

If ALIAS is not specified, this option has no effect.

See also the example in the description of "ATTRIBUTES ALIAS", and the summary of prefix and postfix decorations in the description of "ATTRIBUTES C and STDCALL".

ATTRIBUTES DEFAULT

The ATTRIBUTES directive option DEFAULT overrides certain compiler options that can affect external routine and COMMON block declarations. It takes the following form:

*c*DEC$ ATTRIBUTES DEFAULT :: *entity*

entity

Is an external procedure or COMMON block.

It specifies that the compiler should ignore compiler options that change the default conventions for external symbol naming and argument passing for routines and COMMON blocks.

This property can be combined with other ATTRIBUTES options, such as STDCALL, C, REFERENCE, ALIAS, etc. to specify properties different from the compiler defaults.

This property is useful when declaring INTERFACE blocks for external routines, since it prevents compiler options from changing calling or naming conventions.

ATTRIBUTES DLLEXPORT and DLLIMPORT (W*32, W*64)

The ATTRIBUTES directive options DLLEXPORT and DLLIMPORT define a dynamic-link library's (DLL) interface for processes that use them. The properties can be assigned to data objects or procedures. They take the following forms:

*c*DEC$ ATTRIBUTES DLLEXPORT :: *object* [, *object*] ...

*c*DEC$ ATTRIBUTES DLLIMPORT :: *object* [, *object*] ...

object

Is the name of a data object or procedure.

DLLEXPORT specifies that procedures or data are being exported to other applications or DLLs. This causes the compiler to produce efficient code, eliminating the need for a module definition (.def) file to export symbols.

DLLEXPORT should be specified in the routine to which it applies.

Symbols defined in a DLL are imported by programs that use them. The program must link with the import DLL and use the DLLIMPORT property inside the program unit that imports the symbol. DLLIMPORT is specified in a declaration, not a definition, since you cannot define a symbol you are importing.

For details on working with DLL applications, see your user's guide.

ATTRIBUTES EXTERN

The ATTRIBUTES directive option EXTERN specifies that a variable is allocated in another source file. EXTERN can be used in global variable declarations, but it must not be applied to dummy arguments. It takes the following form:

*c*DEC$ ATTRIBUTES EXTERN :: *var*

var

Is the variable to be allocated.

This option must be used when accessing variables declared in other languages.

ATTRIBUTES IGNORE_LOC

The ATTRIBUTES directive option IGNORE_LOC enables %LOC to be stripped from an argument. It takes the following form:

*c*DEC$ ATTRIBUTES IGNORE_LOC :: *arg*

arg

Is the name of an argument.

IGNORE_LOC is only valid if the REFERENCE property is also specified; otherwise, it has no effect.

ATTRIBUTES INLINE, NOINLINE, and FORCEDINLINE

The ATTRIBUTES directive options INLINE, NOINLINE, and FORCEINLINE can be used to control inline decisions made by the compiler. You should place the directive option in the procedure that calls the routine whose inlining you want to influence.

The INLINE option specifies that a function or subroutine can be inlined. The inlining can be ignored by the compiler if inline heuristics determine it may have a negative impact on performance or will cause too much of an increase in code size. It takes the following form:

*c*DEC$ ATTRIBUTES INLINE :: *procedure*

procedure

Is the function or subroutine that can be inlined.

The NOINLINE option disables inlining of a function. It takes the following form:

*c*DEC$ ATTRIBUTES NOINLINE :: *procedure*

procedure

Is the function or subroutine that must not be inlined.

The FORCEINLINE option specifies that a function or subroutine *must* be inlined unless it will cause errors. It takes the following form:

*c*DEC$ ATTRIBUTES FORCEINLINE :: *procedure*

procedure

Is the function or subroutine that must be inlined.

ATTRIBUTES NO_ARG_CHECK

The ATTRIBUTES directive option NO_ARG_CHECK specifies that type and shape matching rules related to explicit interfaces are to be ignored. This permits the construction of an INTERFACE block for an external procedure or a module procedure that accepts an argument of any type or shape; for example, a memory copying routine. The NO_ARG_CHECK option takes the following form:

*c*DEC$ ATTRIBUTES NO_ARG_CHECK :: *object*

object

Is the name of an argument or procedure.

NO_ARG_CHECK can appear only in an INTERFACE block for a non-generic procedure or in a module procedure. It can be applied to an individual dummy argument name or to the routine name, in which case the property is applied to all dummy arguments in that interface.

NO_ARG_CHECK *cannot* be used for procedures with the PURE or ELEMENTAL prefix. If an argument has an INTENT or OPTIONAL attribute, any NO_ARG_CHECK specification is ignored.

ATTRIBUTES NOMIXED_STR_LEN_ARG

The ATTRIBUTES directive option NOMIXED_STR_LEN_ARG specifies that hidden lengths be placed in sequential order at the end of the argument list. It takes the following form:

*c*DEC$ ATTRIBUTES NOMIXED_STR_LEN_ARG :: *args*

args

Is a list of arguments.

ATTRIBUTES REFERENCE and VALUE

The ATTRIBUTES directive options REFERENCE and VALUE specify how a dummy argument is to be passed. They take the following form:

*c*DEC$ ATTRIBUTES REFERENCE :: *arg*

*c*DEC$ ATTRIBUTES VALUE :: *arg*

arg

Is the name of a dummy argument.

REFERENCE specifies a dummy argument's memory location is to be passed instead of the argument's value.

VALUE specifies a dummy argument's value is to be passed instead of the argument's memory location.

When a dummy argument has the VALUE property, the actual argument passed to it can be of a different type. If necessary, type conversion is performed before the subprogram is called.

When a complex (KIND=4 or KIND=8) argument is passed by value, *two* floating-point arguments (one containing the real part, the other containing the imaginary part) are passed by immediate value.

Character values, substrings, assumed-size arrays, and adjustable arrays cannot be passed by value.

If REFERENCE (only) is specified for a character argument, the following occurs:

- On Linux systems, the string is passed with no length.
- On Windows systems, hidden lengths immediately follow the variable. You can get the Linux behavior by specifying the /iface:nomixed_str_len_arg compiler option.

If REFERENCE is specified for a character argument, and C (or STDCALL) has been specified for the routine, the string is passed with no length. This is true even if REFERENCE is also specified for the routine. If REFERENCE and C (or STDCALL) are specified for a routine, but REFERENCE has *not* been specified for the argument, the string is passed with the length.

VALUE is the default if the C or STDCALL property is specified in the subprogram definition.

In the following example integer x is passed by value:

```
      SUBROUTINE Subr (x)
      INTEGER x
!DEC$ ATTRIBUTES VALUE :: x
```

See Also

- "ATTRIBUTES C and STDCALL"
- "Adjusting Calling Conventions in Mixed-Language Programming" in your user's guide.

ATTRIBUTES VARYING

The ATTRIBUTES directive option VARYING allows a variable number of calling arguments. It takes the following form:

*c*DEC$ ATTRIBUTES VARYING :: *var* [, *var*] ...

Is the name of a variable.

Either the first argument must be a number indicating how many arguments to process, or the last argument must be a special marker (such as –1) indicating it is the final argument. The sequence of the arguments, and types and kinds must be compatible with the called procedure.

If VARYING is specified, the C property must also be specified.

DECLARE and NODECLARE Directives

The DECLARE directive generates warnings for variables that have been used but have not been declared (like the IMPLICIT NONE statement). The NODECLARE directive (the default) disables these warnings.

These directives take the following form:

*c*DEC$ DECLARE

*c*DEC$ NODECLARE

c

Is one of the following: C (or c), !, or * (see "Syntax Rules for Compiler Directives").

The DECLARE directive is primarily a debugging tool that locates variables that have not been properly initialized, or that have been defined but never used.

See Also

"IMPLICIT Statement" for details on the IMPLICIT NONE statement

DEFINE and UNDEFINE Directives

The DEFINE directive creates a symbolic variable whose existence or value can be tested during conditional compilation. The UNDEFINE directive removes a defined symbol.

These directives take the following form:

*c*DEC$ DEFINE *name* [=*val*]

*c*DEC$ UNDEFINE *name*

c

Is one of the following: C (or c), !, or * (see "Syntax Rules for Compiler Directives").

name

Is the name of the variable.

val

Is an INTEGER(4) value assigned to *name*.

Rules and Behavior

DEFINE creates and UNDEFINE removes variables for use with the IF (or IF DEFINED) directive. Symbols defined with the DEFINE directive are local to the directive. They cannot be declared in the Fortran program.

Because Fortran programs cannot access the named variables, the names can duplicate Fortran keywords, intrinsic functions, or user-defined names without conflict.

To test whether a symbol has been defined, use the IF DEFINED (name) directive. You can assign an integer value to a defined symbol. To test the assigned value of *name*, use the IF directive. IF test expressions can contain most logical and arithmetic operators.

Attempting to undefine a symbol which has not been defined produces a compiler warning.

The DEFINE and UNDEFINE directives can appear anywhere in a program, enabling and disabling symbol definitions.

Example

Consider the following:

```
!DEC$ DEFINE testflag
!DEC$ IF DEFINED (testflag)
   WRITE (*,*) 'Compiling first line'
```

```
!DEC$ ELSE
   WRITE (*,*) 'Compiling second line'
!DEC$ ENDIF
!DEC$ UNDEFINE testflag
```

See Also

"IF and IF DEFINED Directives"

DISTRIBUTE POINT Directive

The DISTRIBUTE POINT directive specifies distribution for a DO loop. This directive takes the following form:

*c*DEC$ DISTRIBUTE POINT

c

Is one of the following: C (or c), !, or * (see "Syntax Rules for Compiler Directives").

Rules and Behavior

Loop distribution may cause large loops be distributed into smaller ones, which may cause software pipelining to be applied to more loops.

If the directive is placed before a loop, the compiler will determine where to distribute; data dependencies are observed.

If the directive is placed inside a loop, the distribution is performed after the directive and any loop-carried dependencies are ignored. Currently only one distribute directive is supported if the directive is placed inside the loop.

Example

Consider the following:

```
!DEC$ DISTRIBUTE POINT
   do i =1, m
     b(i) = a(i) +1
     ....
     c(i) = a(i) + b(i) ! Compiler will decide
     ! where to distribute.
     ! Data dependencies are
     ! observed
     ....
     d(i) = c(i) + 1
```

```
   enddo
   do i =1, m
     b(i) = a(i) +1
     ....
!DEC$ DISTRIBUTE POINT
     call sub(a, n)! Distribution will start here,
     ! ignoring all loop-carried
     ! depedencies
     c(i) = a(i) + b(i)
     ....
     d(i) = c(i) + 1
   enddo
```

See Also

"Rules for General Directives that Affect DO Loops"

FIXEDFORMLINESIZE Directive

The FIXEDFORMLINESIZE directive sets the line length for fixed-form source code. The directive takes the following form:

*c*DEC$ FIXEDFORMLINESIZE:{72 | 80 | 132}

c

Is one of the following: C (or c), !, or * (see "Syntax Rules for Compiler Directives").

Rules and Behavior

You can set FIXEDFORMLINESIZE to 72 (the default), 80, or 132 characters. The FIXEDFORMLINESIZE setting remains in effect until the end of the file, or until it is reset.

The FIXEDFORMLINESIZE directive sets the source-code line length in include files, but not in USE modules, which are compiled separately. If an include file resets the line length, the change does not affect the host file.

This directive has no effect on free-form source code.

Example

Consider the following:

```
cDEC$ NOFREEFORM
cDEC$ FIXEDFORMLINESIZE:132
      WRITE (*,*) 'Sentence that goes beyond the 72nd column'
```

See Also

"Fixed and Tab Source Forms" for details on fixed-format source code

FREEFORM and NOFREEFORM Directives

The FREEFORM directive specifies that source code is in free-form format. The NOFREEFORM directive specifies that source code is in fixed-form format.

These directives take the following form:

*c*DEC$ FREEFORM

*c*DEC$ NOFREEFORM

c

Is one of the following: C (or c), !, or * (see "Syntax Rules for Compiler Directives").

When the FREEFORM or NOFREEFORM directives are used, they remain in effect for the remainder of the file, or until the opposite directive is used. When in effect, they apply to include files, but do not affect USE modules, which are compiled separately.

See Also

"Source Forms" for details on free-form and fixed-form source code

IDENT Directive

The IDENT directive specifies a string that identifies an object module. The compiler places the string in the identification field of an object module when it generates the module for each source program unit. The IDENT directive takes the following form:

*c*DEC$ IDENT *string*

c

Is one of the following: C (or c), !, or * (see "Syntax Rules for Compiler Directives").

string

Is a character constant containing up to 31 printable characters.

Only the first IDENT directive is effective; the compiler ignores any additional IDENT directives in a program unit or module.

IF and IF DEFINED Directives

The IF and IF DEFINED directives specify a conditional compilation construct. IF tests whether a logical expression is .TRUE. or .FALSE.. IF DEFINED tests whether a symbol has been defined.

The directive-initiated construct takes the following form:

*c*DEC$ IF (*expr*) [or *c*DEC$ IF DEFINED (*name*)]
  block
[*c*DEC$ ELSE IF (*expr*)
  block]...
[*c*DEC$ ELSE
  block]
*c*DEC$ ENDIF

c

Is one of the following: C (or c), !, or * (see "Syntax Rules for Compiler Directives").

expr

Is a logical expression that evaluates to .TRUE. or .FALSE..

name

Is the name of a symbol to be tested for definition.

block

Are executable statements that are compiled (or not) depending on the value of logical expressions in the IF directive construct.

Rules and Behavior

The IF and IF DEFINED directive constructs end with an ENDIF directive and can contain one or more ELSEIF directives and at most one ELSE directive. If the logical condition within a directive evaluates to .TRUE. at compilation, and all preceding conditions in the IF construct evaluate to .FALSE., then the statements contained in the directive block are compiled.

A *name* can be defined with a DEFINE directive, and can optionally be assigned an integer value. If the symbol has been defined, with or without being assigned a value, IF DEFINED (name) evaluates to .TRUE.; otherwise, it evaluates to .FALSE..

If the logical condition in the IF or IF DEFINED directive is .TRUE., statements within the IF or IF DEFINED block are compiled. If the condition is .FALSE., control transfers to the next ELSEIF or ELSE directive, if any.

If the logical expression in an ELSEIF directive is .TRUE., statements within the ELSEIF block are compiled. If the expression is .FALSE., control transfers to the next ELSEIF or ELSE directive, if any.

If control reaches an ELSE directive because all previous logical conditions in the IF construct evaluated to .FALSE., the statements in an ELSE block are compiled unconditionally.

You can use any Fortran logical or relational operator or symbol in the logical expression of the directive, including: .LT., <, .GT., >, .EQ., ==, .LE., <=, .GE., >=, .NE., /=, .EQV., .NEQV., .NOT., .AND., .OR., and .XOR.. The logical expression can be as complex as you like, but the whole directive must fit on one line.

Example

Consider the following:

```
!  When the following code is compiled and run,
!  the output depends on whether one of the expressions
!  tests .TRUE., or all test .FALSE.
!DEC$ DEFINE   flag=3

!DEC$ IF (flag .LT. 2)
   WRITE (*,*) "This is compiled if flag less than 2."
!DEC$ ELSEIF (flag >= 8)
   WRITE (*,*) "Or this is compiled if flag greater than &
                or equal to 8."
!DEC$ ELSE
   WRITE (*,*) "Or this is compiled if all preceding &
                conditions .FALSE."
!DEC$ ENDIF
```

See Also

"DEFINE and UNDEFINE Directives"

INTEGER Directive

The INTEGER directive specifies the default integer kind. This directive takes the following form:

cDEC$ INTEGER:{1 | 2 | 4 | 8}

c

Is one of the following: C (or c), !, or * (see "Syntax Rules for Compiler Directives").

Rules and Behavior

The INTEGER directive specifies a size of 1 (KIND=1), 2 (KIND=2), 4 (KIND=4), or 8 (KIND=8) bytes for default integer numbers.

When the INTEGER directive is effect, all default integer variables are of the kind specified in the directive. Only numbers specified or implied as INTEGER without KIND are affected.

The INTEGER directive can only appear at the top of a program unit. A program unit is a main program, an external subroutine or function, a module or a block data program unit. The directive cannot appear between program units, or at the beginning of internal subprograms. It does not affect modules invoked with the USE statement in the program unit that contains it.

The default logical kind is the same as the default integer kind. So, when you change the default integer kind you also change the default logical kind.

Example

Consider the following:

```
INTEGER i                  ! a 4-byte integer
WRITE(*,*) KIND(i)
CALL INTEGER2( )

WRITE(*,*) KIND(i)         ! still a 4-byte integer
                           !   not affected by setting in subroutine
END
SUBROUTINE INTEGER2( )
  !DEC$ INTEGER:2
  INTEGER j                ! a 2-byte integer
  WRITE (*,*) KIND(j)
END SUBROUTINE
```

See Also

- "Integer Data Types"
- "REAL Directive"

IVDEP Directive

The IVDEP directive assists the compiler's dependence analysis of iterative DO loops.

The IVDEP directive takes the following form:

*c*DEC$ IVDEP [: *option*]

c

Is one of the following: C (or c), !, or * (see "Syntax Rules for Compiler Directives").

option

Is LOOP or BACK. This argument is only available on IA-32 processors.

Rules and Behavior

The IVDEP directive is an assertion to the compiler's optimizer about the order of memory references inside a DO loop.

IVDEP:LOOP implies no loop-carried dependencies. IVDEP:BACK implies no backward dependencies.

When no *option* is specified, the following occurs:

- On Intel® Itanium® processors, the behavior is the same as IVDEP:BACK. You can modify the behavior to be the same as IVDEP:LOOP by using a compiler option.
- On IA-32 processors, the compiler begins dependence analysis by assuming all dependences occur in the same forward direction as their appearance in the normal scalar execution order. This contrasts with normal compiler behavior, which is for the dependence analysis to make no initial assumptions about the direction of a dependence.

*c*DEC$ IVDEP with no option can also be spelled *c*DEC$ INIT_DEP_FWD (INITialize DEPendences ForWarD).

The IVDEP directive is applied to a DO loop in which the user knows that dependences are in lexical order. For example, if two memory references in the loop touch the same memory location and one of them modifies the memory location, then the first reference to touch the location has to be the one that appears earlier lexically in the program source code. This assumes that the right-hand side of an assignment statement is "earlier" than the left-hand side.

The IVDEP directive informs the compiler that the program would behave correctly if the statements were executed in certain orders other than the sequential execution order, such as executing the first statement or block to completion for all iterations, then the next statement or block for all iterations, and so forth. The optimizer can use this information, along with whatever else it can prove about the dependences, to choose other execution orders.

Example

In the following example, the IVDEP directive provides more information about the dependences within the loop, which may enable loop transformations to occur:

```
!DEC$ IVDEP
   DO I=1, N
     A(INDARR(I)) = A(INDARR(I)) + B(I)
   END DO
```

In this case, the scalar execution order follows:

1. Retrieve INDARR(I).
2. Use the result from step 1 to retrieve A(INDARR(I)).
3. Retrieve B(I).

4. Add the results from steps 2 and 3.
5. Store the results from step 4 into the location indicated by A(INDARR(I)) from step 1.

IVDEP directs the compiler to initially assume that when steps 1 and 5 access a common memory location, step 1 always accesses the location first because step 1 occurs earlier in the execution sequence. This approach lets the compiler reorder instructions, as long as it chooses an instruction schedule that maintains the relative order of the array references.

See Also

"Rules for General Directives that Affect DO Loops"

LOOP COUNT Directive

The LOOP COUNT directive specifies the loop count for a DO loop; this assists the optimizer This directive takes the following form:

*c*DEC$ LOOP COUNT (*n*)

c

Is one of the following: C (or c), !, or * (see "Syntax Rules for Compiler Directives").

n

Is an integer constant.

Rules and Behavior

The value of the loop count affects heuristics used in software pipelining, vectorization, and loop-transformations.

Example

Consider the following:

```
cDEC$ LOOP COUNT (10000)
   do i =1,m
     b(i) = a(i) +1 ! This is likely to enable
     ! the loop to get software-pipelined
   enddo
```

See Also

"Rules for General Directives that Affect DO Loops"

MESSAGE Directive

The MESSAGE directive specifies a character string to be sent to the standard output device during the first compiler pass; this aids debugging.

This directive takes the following form:

*c*DEC$ MESSAGE:*string*

c

Is one of the following: C (or c), !, or * (see "Syntax Rules for Compiler Directives").

string

Is a character constant specifying a message.

Example

Consider the following:

```
!DEC$ MESSAGE:'Compiling Sound Speed Equations'
```

OBJCOMMENT Directive

The OBJCOMMENT directive specifies a library search path in an object file. This directive takes the following form:

*c*DEC$ OBJCOMMENT LIB:*library*

c

Is one of the following: C (or c), !, or * (see "Syntax Rules for Compiler Directives").

library

Is a character constant specifying the name and, if necessary, the path of the library that the linker is to search.

Rules and Behavior

The linker searches for the library named by the OBJCOMMENT directive as if you named it on the command line, that is, before default library searches. You can place multiple library search directives in the same source file. Each search directive appears in the object file in the order it is encountered in the source file.

If the OBJCOMMENT directive appears in the scope of a module, any program unit that uses the module also contains the directive, just as if the OBJCOMMENT directive appeared in the source file using the module.

If you want to have the OBJCOMMENT directive in a module, but do not want it in the program units that use the module, place the directive outside the module that is used.

Example

Consider the following:

```
! MOD1.F90
MODULE a
  !DEC$ OBJCOMMENT LIB: "opengl32.lib"
END MODULE a

! MOD2.F90
!DEC$ OBJCOMMENT LIB: "graftools.lib"
MODULE b
!
END MODULE b

! USER.F90
PROGRAM go
  USE a     ! library search contained in MODULE a
            !   included here
  USE b     ! library search not included
END
```

OPTIONS Directive

The OPTIONS directive affects data alignment and warnings about data alignment. It takes the following form:

*c*DEC$ OPTIONS *option* [*option*]

...

*c*DEC$ END OPTIONS

c

Is one of the following: C (or c), !, or * (see "Syntax Rules for Compiler Directives").

option

Is one (or both) of the following:

- /WARN=[NO]ALIGNMENT

  Controls whether warnings are issued by the compiler for data that is not naturally aligned. By default, you receive compiler messages when misaligned data is encountered (/WARN=ALIGNMENT).

- /[NO]ALIGN[= *p*]

  Controls alignment of fields in record structures and data items in common blocks. The fields and data items can be naturally aligned (for performance reasons) or they can be packed together on arbitrary byte boundaries.

  p

  Is a specifier with one of the following forms:

  [*class* =] *rule*

  (*class* = *rule*,...)

  ALL

  NONE

  class

  Is one of the following keywords:

  — COMMONS: For common blocks

  — RECORDS: For records

  — STRUCTURES: A synonym for RECORDS

  rule

  Is one of the following keywords:

  — PACKED

  Packs fields in records or data items in common blocks on arbitrary byte boundaries.

  — NATURAL

  Naturally aligns fields in records and data items in common blocks on up to 64-bit boundaries (inconsistent with the Fortran 95/90 standard).

  This keyword causes the compiler to naturally align all data in a common block, including INTEGER(8), REAL(8), and all COMPLEX data.

  — STANDARD

  Naturally aligns data items in common blocks on up to 32-bit boundaries (consistent with the Fortran 95/90 standard).

  This keyword only applies to common blocks; so, you can specify /ALIGN=COMMONS=STANDARD, but you cannot specify /ALIGN=STANDARD.

  ALL

  Is the same as specifying OPTIONS /ALIGN, OPTIONS /ALIGN=NATURAL, and OPTIONS /ALIGN=(RECORDS=NATURAL,COMMONS=NATURAL).

  NONE

  Is the same as specifying OPTIONS /NOALIGN, OPTION /ALIGN=PACKED, and OPTIONS/ALIGN=(RECORDS=PACKED,COMMONS=PACKED).

Rules and Behavior

The OPTIONS (and accompanying END OPTIONS) directives must come after OPTIONS, SUBROUTINE, FUNCTION, and BLOCK DATA statements (if any) in the program unit, and before the executable part of the program unit.

The OPTIONS directive supersedes the compiler option that sets alignment.

For performance reasons, Intel Fortran aligns local data items on natural boundaries. However, EQUIVALENCE, COMMON, RECORD, and STRUCTURE data declaration statements can force misaligned data. If /WARN=NOALIGNMENT is specified, warnings will not be issued if misaligned data is encountered.

NOTE. *Misaligned data significantly increases the time it takes to execute a program. As the number of misaligned fields encountered increases, so does the time needed to complete program execution. Specifying* /ALIGN *(or the compiler option that sets alignment) minimizes misaligned data.*

If you want aligned data in common blocks, do one of the following:

- Specify /ALIGN=COMMONS=STANDARD for data items up to 32 bits in length.
- Specify /ALIGN=COMMONS=NATURAL for data items up to 64 bits in length.
- Place source data declarations within the common block in descending size order, so that each data item is naturally aligned.

If you want packed, unaligned data in a record structure, do one of the following:

- Specify /ALIGN=RECORDS=PACKED.
- Place source data declarations in the record structure so that the data is naturally aligned.

An OPTIONS directive must be accompanied by an END OPTIONS directive; the directives can be nested up to 100 levels. For example:

```
CDEC$ OPTIONS   /ALIGN=PACKED          ! Start of Group A
   declarations
CDEC$ OPTIONS   /ALIGN=RECO=NATU          ! Start of nested Group B
   more declarations
CDEC$ END   OPTIONS                       ! End of Group B
   still more declarations
CDEC$ END   OPTIONS                    ! End of Group A
```

The OPTIONS specification for Group B only applies to RECORDS; common blocks within Group B will be PACKED. This is because COMMONS retains the previous setting (in this case, from the Group A specification).

See Also

Your user's guide for details on alignment and data sizes, and details on compiler options

PACK Directive

The PACK directive specifies the memory starting addresses of derived-type items. This directive takes the following form:

cDEC$ PACK[: [{1 | 2 | 4 | 8}]]

c

Is one of the following: C (or c), !, or * (see "Syntax Rules for Compiler Directives").

Rules and Behavior

Items of derived types and record structures are aligned in memory on the smaller of two sizes: the size of the type of the item, or the current alignment setting. The current alignment setting can be 1, 2, 4, or 8 bytes. The default initial setting is 8 bytes (unless a compiler option specifies otherwise). By reducing the alignment setting, you can pack variables closer together in memory.

The PACK directive lets you control the packing of derived-type or record structure items inside your program by overriding the current memory alignment setting.

For example, if PACK:1 is specified, all variables begin at the next available byte, whether odd or even. Although this slightly increases access time, no memory space is wasted. If PACK:4 is specified, INTEGER(1), LOGICAL(1), and all character variables begin at the next available byte, whether odd or even. INTEGER(2) and LOGICAL(2) begin on the next even byte; all other variables begin on 4-byte boundaries.

If the PACK directive is specified without a number, packing reverts to the compiler option setting (if any), or the default setting of 8.

The directive can appear anywhere in a program before the derived-type definition or record structure definition. It cannot appear *inside* a derived-type or record structure definition.

Example

Consider the following:

```
! Use 4-byte packing for this derived type
! Note PACK is used outside of the derived-type definition
!DEC$ PACK:4
TYPE pair
  INTEGER a, b
END TYPE
```

```
! revert to default or compiler option
!DEC$ PACK
```

See Also

- "Record Structures"
- Your user's guide for details on compiler options that affect packing

PARALLEL and NOPARALLEL Directives

The PARALLEL directive enables auto-parallelization for an immediately following DO loop. The NOPARALLEL directive (the default) disables this auto-parallelization.

These directives take the following form:

*c*DEC$ PARALLEL

*c*DEC$ NOPARALLEL

c

Is one of the following: C (or c), !, or * (see "Syntax Rules for Compiler Directives").

Rules and Behavior

PARALLEL instructs the compiler to ignore dependencies that it assumes may exist and which would prevent correct parallelization in the immediately following loop. However, if dependencies are proven, they are not ignored.

Example

Consider the following:

```
program main
parameter (n=100)
integer x(n),a(n)
!DEC$ NOPARALLEL
   do i=1,n
     x(i) = i
   enddo
!DEC$ PARALLEL
   do i=1,n
     a( x(i) ) = i
   enddo end
```

See Also

"Rules for General Directives that Affect DO Loops"

PREFETCH and NOPREFETCH Directives

The PREFETCH directive enables a data prefetch from memory. Prefetching data can minimize the effects of memory latency. The NOPREFETCH directive (the default) disables data prefetching. These directives affect the heuristics used in the compiler.

The directives take the following form:

*c*DEC$ PREFETCH [*var1* [: *hint1* [: *distance1*]] [, *var2* [: *hint2* [: *distance2*]]]...]

*c*DEC$ NOPREFETCH [*var1* [, *var2*]...]

c

Is one of the following: C (or c), !, or *. (See "Syntax Rules for Compiler Directives".)

var

Is an optional memory reference.

hint

Is an optional integer initialization expression with the value 0, 1, 2, or 3. These are the same as the values for *hint* in the intrinsic subroutine MM_PREFETCH. To use this argument, you must also specify *var*.

distance

Is an optional integer initialization expression with a value greater than 0. It indicates the number of loop iterations to perform before the prefetch. To use this argument, you must also specify *var* and *hint*.

Rules and Behavior

To use these directives, compiler option /O3 must be set.

This directive affects the DO loop it precedes.

If you specify PREFETCH with no arguments, all arrays accessed in the DO loop will be prefetched.

If a loop includes expression A(j), placing *c*DEC$ PREFETCH A in front of the loop, instructs the compiler to insert prefetches for A(j + d) within the loop. The d is determined by the compiler.

Example

Consider the following:

```
cDEC$ NOPREFETCH c
cDEC$ PREFETCH a
do i = 1, m
  b(i) = a(c(i)) + 1
enddo
```

See Also

- "Rules for General Directives that Affect DO Loops"
- intrinsic subroutine "MM_PREFETCH"

PSECT Directive

The PSECT directive modifies several characteristics of a common block. It takes the following form:

*c*DEC$ PSECT /*common-name*/ *a* [, *a*] . . .

c

Is one of the following: C (or c), !, or * (see "Syntax Rules for Compiler Directives").

common-name

Is the name of the common block. The slashes (/) are required.

a

Is one of the following keywords:

- ALIGN=*val* or ALIGN=*keyword*

  Specifies alignment for the common block.

  The *val* is a constant ranging from 0 through 6 on Windows systems and 0 through 4 on Linux systems. The specified number is interpreted as a power of 2. The value of the expression is the alignment in bytes.

  The *keyword* is one of the following:

Keyword	Equivalent to *val*
BYTE	0
WORD	1
LONG	2
QUAD	3
OCTA	4
PAGE [1]	i32: 12
	i64: 13

1. Range is 0 to 13 except on L*X32, where the range is 0 to 12.

- [NO]WRT

  Determines whether the contents of a common block can be modified during program execution.

Rules and Behavior

Global or local scope is significant for an image that has more than one cluster. Program sections with the same name that are from different modules in different clusters are placed in separate clusters if local scope is in effect. They are placed in the same cluster if global scope is in effect.

If one program unit changes one or more characteristics of a common block, all other units that reference that common block must also change those characteristics in the same way.

Default characteristics apply if you do not modify them with a PSECT directive. Table 14-1 lists the default characteristics of common blocks and how they can be modified by PSECT.

Table 14-1 Common Block Defaults and PSECT Modification

Default Characteristics	PSECT Modification
Relocatable	None
Overlaid	None
Global scope	Global or local scope
Not executable	None
Not multilanguage	Multilanguage or not multilanguage
Writable	Writable or not writable
Readable	None
No protection	None
Octaword alignment[1]	W*32, W*64: 0 through 6[2] L*X: 0 through 4[2]
On i64:	
Not shareable	Shareable or not shareable
Position dependent	None
On i32:	
Shareable	Shareable or not shareable
Position independent	None

1. An address that is an integral multiple of 16.
2. Or keywords BYTE through OCTA

See Also

- `ld(1)` for details on the linker for Linux* systems
- The online help on the linker for details on the linker for Windows* systems
- Your user's guide for details on compiler options

REAL Directive

The REAL directive specifies the default real kind. This directive takes the following form:

*c*DEC$ REAL:{ 4 | 8 | 16 }

c

Is one of the following: C (or c), !, or * (see "Syntax Rules for Compiler Directives").

Rules and Behavior

The REAL directive specifies a size of 4 (KIND=4), 8 (KIND=8), or 16 (KIND=16) bytes for default real numbers.

When the REAL directive is effect, all default real variables are of the kind specified in the directive. Only numbers specified or implied as REAL without KIND are affected.

The REAL directive can only appear at the top of a program unit. A program unit is a main program, an external subroutine or function, a module or a block data program unit. The directive cannot appear between program units, or at the beginning of internal subprograms. It does not affect modules invoked with the USE statement in the program unit that contains it.

Example

Consider the following:

```
REAL r                    ! a 4-byte REAL
WRITE(*,*) KIND(r)
CALL REAL8( )
WRITE(*,*) KIND(r)        ! still a 4-byte REAL
                          !   not affected by setting in subroutine
END
SUBROUTINE REAL8( )
  !DEC$ REAL:8
  REAL s                  ! an 8-byte REAL
  WRITE(*,*) KIND(s)
END SUBROUTINE
```

See Also

- "Real Data Types"
- "INTEGER Directive"
- Your user's guide for details on compiler options that can affect REAL types

STRICT and NOSTRICT Directives

The STRICT directive disables language features not found in the language standard specified on the command line (Fortran 95 or Fortran 90). The NOSTRICT directive (the default) enables these language features.

These directives take the following form:

*c*DEC$ STRICT
*c*DEC$ NOSTRICT

c

Is one of the following: C (or c), !, or * (see "Syntax Rules for Compiler Directives").

Rules and Behavior

If STRICT is specified and no language standard is specified on the command line, the default is to disable features not found in Fortran 90.

The STRICT and NOSTRICT directives can appear only appear at the top of a program unit. A program unit is a main program, an external subroutine or function, a module, or a block data program unit. The directives cannot appear between program units, or at the beginning of internal subprograms. They do not affect any modules invoked with the USE statement in the program unit that contains them.

Example

Consider the following:

```
! NOSTRICT by default
TYPE stuff
  INTEGER(4)    k
  INTEGER(4)    m
  CHARACTER(4)  name
END TYPE stuff
TYPE (stuff)    examp
DOUBLE COMPLEX  cd     ! non-standard data type, no error
cd = (3.0D0, 4.0D0)
examp.k = 4            ! non-standard component designation,
                       !   no error
END
SUBROUTINE STRICTDEMO( )
  !DEC$ STRICT
   TYPE stuff
```

```
      INTEGER(4)    k
      INTEGER(4)    m
      CHARACTER(4)  name
    END TYPE stuff
    TYPE (stuff)    samp
    DOUBLE COMPLEX  cd      ! ERROR
    cd = (3.0D0, 4.0D0)
    samp.k = 4              ! ERROR
END SUBROUTINE
```

SWP and NOSWP Directives (i64 only)

The SWP directive enables software pipelining for a DO loop. The NOSWP directive (the default) disables this software pipelining. These directives are only available on Intel Itanium processors.

The directives take the following form:

*c*DEC$ SWP

*c*DEC$ NOSWP

c

Is one of the following: C (or c), !, or *. (See "Syntax Rules for Compiler Directives".)

Rules and Behavior

The SWP directive does not help data dependence, but overrides heuristics based on profile counts or lop-sided control flow.

The software pipelining optimization specified by the SWP directive applies instruction scheduling to certain innermost loops, allowing instructions within a loop to be split into different stages.

This allows increased instruction level parallelism, which can reduce the impact of long-latency operations, resulting in faster loop execution.

Loops chosen for software pipelining are always innermost loops containing procedure calls that are inlined. Because the optimizer no longer considers fully unrolled loops as innermost loops, fully unrolling loops can allow an additional loop to become the innermost loop (see the compiler option for loop unrolling in your user's guide).

You can request and view the optimization report to see whether software pipelining was applied (see "Optimizer Report Generation" in your user's guide, Volume II).

Example

Consider the following:

```
!DEC$ SWP
  do i = 1, m
    if (a(i) .eq. 0) then
      b(i) = a(i) + 1
    else
      b(i) = a(i)/c(i)
    endif
  enddo
```

See Also

"Rules for General Directives that Affect DO Loops"

TITLE and SUBTITLE Directives

The TITLE directive specifies a string for the title field of a listing header. Similarly, SUBTITLE specifies a string for the subtitle field of a listing header.

These directives take the following form:

*c*DEC$ TITLE *string*
*c*DEC$ SUBTITLE *string*

c

Is one of the following: C (or c), !, or * (see "Syntax Rules for Compiler Directives").

string

Is a character constant containing up to 31 printable characters.

Rules and Behavior

To enable TITLE and SUBTITLE directives, you must specify the compiler option that produces a source listing file.

When TITLE or SUBTITLE appear on a page of a listing file, the specified string appears in the listing header of the following page.

If two or more of either directive appear on a page, the last directive is the one in effect for the following page.

If either directive does not specify a string, no change occurs in the listing file header.

See Also

Your user's guide for details on compiler options

UNROLL and NOUNROLL Directives

The UNROLL directive tells the compiler's optimizer how many times to unroll a DO loop. The NOUNROLL directive (the default) disables the unrolling of a DO loop. These directives can only be applied to iterative DO loops.

The directives take the following form:

*c*DEC$ UNROLL [(*n*)]
*c*DEC$ NOUNROLL

c

Is one of the following: C (or c), !, or * (see "Syntax Rules for Compiler Directives").

n

Is an integer constant. The range of *n* is 0 through 255.

Rules and Behavior

If *n* is specified, the optimizer unrolls the loop *n* times. If *n* is omitted, or if it is outside the allowed range, the optimizer picks the number of times to unroll the loop.

The UNROLL directive overrides any setting of loop unrolling from the command line.

See Also

"Rules for General Directives that Affect DO Loops"

VECTOR ALIGNED and VECTOR UNALIGNED Directives (i32 only)

The VECTOR ALIGNED directive specifies that all data in a DO loop is aligned. The VECTOR UNALIGNED directive specifies that the data in the loop is not aligned. These directives are only available on IA-32 processors.

The directives take the following form:

*c*DEC$ VECTOR ALIGNED
*c*DEC$ VECTOR UNALIGNED

c

Is one of the following: C (or c), !, or *. (See "Syntax Rules for Compiler Directives".)

Rules and Behavior

These directives override efficiency heuristics in the optimizer. The qualifiers UNALIGNED and ALIGNED instruct the compiler to use, respectively, unaligned and aligned data movement instructions for all array references. This disables all the advanced alignment optimizations of the compiler, such as determining alignment properties from the program context or using dynamic loop peeling to make references aligned.

CAUTION. *The directives* VECTOR ALIGNED and VECTOR UNALIGNED *should be used with care. Overriding the efficiency heuristics of the compiler should only be done if you are absolutely sure the vectorization will improve performance.*
Furthermore, instructing the compiler to implement all array references with aligned data movement instructions will cause a runtime exception if some of the access patterns are actually unaligned.

See Also

"Rules for General Directives that Affect DO Loops"

VECTOR ALWAYS and NOVECTOR Directives (i32 only)

The VECTOR ALWAYS directive enables vectorization of a DO loop. The NOVECTOR directive disables this vectorization. These directives are only available IA-32 processors.

The directives take the following form:

*c*DEC$ VECTOR ALWAYS
*c*DEC$ NOVECTOR

c

Is one of the following: C (or c), !, or *. (See "Syntax Rules for Compiler Directives".)

Rules and Behavior

The VECTOR ALWAYS and NOVECTOR directives override the default behavior of the compiler. The VECTOR ALWAYS directive also overrides efficiency heuristics of the vectorizer, but it only works if the loop can actually be vectorized. You should use the "IVDEP Directive" to ignore assumed dependences.

CAUTION. *The directive* VECTOR ALWAYS *should be used with care. Overriding the efficiency heuristics of the compiler should only be done if you are absolutely sure the vectorization will improve performance.*

Examples

The compiler normally does not vectorize DO loops that have a large number of non-unit stride references (compared to the number of unit stride references).

In the following example, vectorization would be disabled by default, but the directive overrides this behavior:

```
!DEC$ VECTOR ALWAYS
  do i = 1, 100, 2
    ! two references with stride 2 follow
    a(i) = b(i)
  enddo
```

There may be cases where you want to explicitly avoid vectorization of a loop; for example, if vectorization would result in a performance regression rather than an improvement. In these cases, you can use the NOVECTOR directive to disable vectorization of the loop.

In the following example, vectorization would be performed by default, but the directive overrides this behavior:

```
!DEC$ NOVECTOR
  do i = 1, 100
   a(i) = b(i) + c(i)
  enddo
```

See Also

"Rules for General Directives that Affect DO Loops"

VECTOR NONTEMPORAL Directive (i32 only)

The VECTOR NONTEMPORAL directive enables streaming storage. This directive is only available on IA-32 processors. It takes the following form:

*c*DEC$ VECTOR NONTEMPORAL

c

Is one of the following: C (or c), !, or *. (See "Syntax Rules for Compiler Directives".)

Streaming storage may cause significant performance improvements over non-streaming storage for large numbers on certain IA-32 processors.

For more information on this directive, including an example, see "Vectorization Support" in your user's guide, Volume II: Optimizing Applications.

See Also

"Rules for General Directives that Affect DO Loops"

OpenMP* Fortran Compiler Directives

Intel Fortran provides OpenMP* Fortran compiler directives that comply with OpenMP Fortran Application Program Interface (API) specification Version 1.1 and most of Version 2.0.

To use these directives, you must specify the compiler option that enables the directives. For information on the form for this compiler option and how to use these directives, see your user's guide.

This section discusses the following topics:

- "Data Scope Attribute Clauses"
- "Conditional Compilation Rules"
- "Nesting and Binding Rules"
- "ATOMIC Directive"
  Specifies that a specific memory location is to be updated dynamically.
- "BARRIER Directive"
  Synchronizes all the threads in a team.
- "CRITICAL Directive"
  Restricts access for a block of code to only one thread at a time.
- "DO Directive"
  Specifies that the iterations of the immediately following DO loop must be executed in parallel.
- "FLUSH Directive"
  Specifies synchronization points where the implementation must have a consistent view of memory.
- "MASTER Directive"
  Specifies a block of code to be executed by the master thread of the team.
- "ORDERED Directive"
  Specifies a block of code to be executed sequentially.
- "PARALLEL Directive"
  Defines a parallel region.
- "PARALLEL DO Directive"
  Defines a parallel region that contains a single DO directive.
- "PARALLEL SECTIONS Directive"
  Defines a parallel region that contains SECTIONS directives.
- "SECTIONS Directive"
  Specifies a block of code to be divided among threads in a team (a worksharing area).

- "SINGLE Directive"
  Specifies a block of code to be executed by only one thread in a team.
- "THREADPRIVATE Directive"
  Makes named common blocks private to a thread but global within the thread.

The OpenMP parallel directives can be grouped into the categories shown in Table 14-2.

Table 14-2 Categories of OpenMP Fortran Parallel Directives

Category	Description
Parallel region	Defines a parallel region: PARALLEL
Work sharing	Divide the execution of the enclosed block of code among the members of the team that encounter it: DO and SECTIONS
Combined parallel work sharing	Shortcut for denoting a parallel region that contains only one work-sharing construct: PARALLEL DO and PARALLEL SECTIONS
Synchronization	Provide various aspects of synchronization; for example, access to a block of code, or execution order of statements within a block of code: ATOMIC, BARRIER, CRITICAL, FLUSH, MASTER, and ORDERED
Data environment	Control the data environment during the execution of parallel constructs: THREADPRIVATE

Note that certain general directives and rules can affect DO loops. For more information, see "Rules for General Directives that Affect DO Loops".

Examples

The following examples are equivalent:

```
!$OMP PARALLEL DO  &
!$OMP SHARED(A,B,C)
```

```
!$OMP PARALLEL  &
!$OMP DO SHARED(A,B,C)
```

```
!$OMP PARALLEL DO SHARED(A,B,C)
```

See Also

- Your user's guide (Volume II) for details on how to use these directives
- http://www.openmp.org/ for details on OpenMP

Data Scope Attribute Clauses

Some of the OpenMP Fortran directives have clauses (or options) you can specify to control the scope attributes of variables for the duration of the directive. This section discusses the following data scope attribute clauses:

- "COPYIN Clause"
- "COPYPRIVATE Clause"
- "DEFAULT Clause"
- "FIRSTPRIVATE Clause"
- "LASTPRIVATE Clause"
- "PRIVATE Clause"
- "REDUCTION Clause"
- "SHARED Clause"

Other clauses (or options) are available for some OpenMP Fortran directives. For more information, see each directive description.

COPYIN Clause

The COPYIN clause specifies that the data in the master thread of the team is to be copied to the thread private copies of the common block at the beginning of the parallel region. It takes the following form:

COPYIN (*list*)

list

Is the name of one or more variables or common blocks that are accessible to the scoping unit. Subobjects cannot be specified. Each name must be separated by a comma, and a named common block must appear between slashes (/ /).

The COPYIN clause applies only to common blocks declared as THREADPRIVATE.

You do not need to specify the whole THREADPRIVATE common block, you can specify named variables within the common block.

COPYPRIVATE Clause

The COPYPRIVATE clause uses a private variable to broadcast a value, or a pointer to a shared object, from one member of a team to the other members. The COPYPRIVATE clause can only appear in the END SINGLE directive. It takes the following form:

COPYPRIVATE (*list*)

list

Is the name of one or more variables or common blocks that are accessible to the scoping unit. Subobjects cannot be specified. Each name must be separated by a comma, and a named common block must appear between slashes (/ /).

Variables in the list must not appear in a PRIVATE or FIRSTPRIVATE clause for the SINGLE directive construct.

If the directive is encountered in the dynamic extent of a parallel region, variables in the list must be private in the enclosing context.

If a common block is specified, it must be declared as THREADPRIVATE; the effect is the same as if the variable names in its common block object list were specified.

The effect of the COPYPRIVATE clause on the variables in its list occurs after the execution of the code enclosed within the SINGLE construct, and before any threads in the team have left the barrier at the end of the construct.

DEFAULT Clause

The DEFAULT clause lets you specify a scope for all variables in the lexical extent of a parallel region. It takes the following form:

$$\text{DEFAULT}\left(\left\{\begin{array}{c}\text{PRIVATE}\\ \text{SHARED}\\ \text{NONE}\end{array}\right\}\right)$$

The specifications have the following effects:

- PRIVATE — Makes all named objects in the lexical extent of the parallel region, including common block variables but excluding THREADPRIVATE variables, private to a thread as if you explicitly listed each variable in a PRIVATE clause.
- SHARED — Makes all named objects in the lexical extent of the parallel region shared among the threads in a team, as if you explicitly listed each variable in a SHARED clause. If you do not specify a DEFAULT clause, this is the default.
- NONE — Specifies that there is no implicit default as to whether variables are PRIVATE or SHARED. In this case, you must specify the PRIVATE, SHARED, FIRSTPRIVATE, LASTPRIVATE, or REDUCTION property of each variable you use in the lexical extent of the parallel region.

You can specify only one DEFAULT clause in a PARALLEL directive. You can exclude variables from a defined default by using the PRIVATE, SHARED, FIRSTPRIVATE, LASTPRIVATE, or REDUCTION clauses.

Variables in THREADPRIVATE common blocks are not affected by this clause.

FIRSTPRIVATE Clause

The FIRSTPRIVATE clause provides a superset of the functionality provided by the PRIVATE clause (see "PRIVATE Clause"); objects are declared PRIVATE and they are initialized with certain values. It takes the following form:

FIRSTPRIVATE (*list*)

list

Is the name of one or more variables or common blocks that are accessible to the scoping unit. Subobjects cannot be specified. Each name must be separated by a comma, and a named common block must appear between slashes (/ /).

Variables that appear in a FIRSTPRIVATE list are subject to PRIVATE clause semantics. In addition, private (local) copies of each variable in the different threads are initialized to the value the variable had before the parallel region started.

LASTPRIVATE Clause

The LASTPRIVATE clause provides a superset of the functionality provided by the PRIVATE clause (see "PRIVATE Clause"); objects are declared PRIVATE and they are given certain values when the parallel region is exited. It takes the following form:

LASTPRIVATE (*list*)

list

Is the name of one or more variables or common blocks that are accessible to the scoping unit. Subobjects cannot be specified. Each name must be separated by a comma, and a named common block must appear between slashes (/ /).

Variables that appear in a LASTPRIVATE list are subject to PRIVATE clause semantics. In addition, once the parallel region is exited, each variable has the value provided by the sequentially last section or loop iteration.

When the LASTPRIVATE clause appears in a DO directive, the thread that executes the sequentially last iteration updates the version of the object it had before the construct. When the LASTPRIVATE clause appears in a SECTIONS directive, the thread that executes the lexically last SECTION updates the version of the object it had before the construct.

Subobjects that are not assigned a value by the last iteration of the DO or the lexically last SECTION of the SECTIONS directive are undefined after the construct.

PRIVATE Clause

The PRIVATE clause declares specified variables to be private to each thread in a team. It takes the following form:

PRIVATE (*list*)

list

Is the name of one or more variables or common blocks that are accessible to the scoping unit. Subobjects cannot be specified. Each name must be separated by a comma, and a named common block must appear between slashes (/ /).

Rules and Behavior

The following occurs when variables are declared in a PRIVATE clause:

- A new object of the same type is declared once for each thread in the team. The new object is no longer storage associated with the original object.
- All references to the original object in the lexical extent of the directive construct are replaced with references to the private object.
- Variables defined as PRIVATE are undefined for each thread on entering the construct and the corresponding shared variable is undefined on exit from a parallel construct.
- Contents, allocation state, and association status of variables defined as PRIVATE are undefined when they are referenced outside the lexical extent (but inside the dynamic extent) of the construct, unless they are passed as actual arguments to called routines.

REDUCTION Clause

The REDUCTION clause performs a commutative reduction operation on the specified variables. It takes the following form:

$$\text{REDUCTION} \left(\left\{ \begin{array}{c} operator \\ intrinsic \end{array} \right\} : list \right)$$

operator

Is one of the following: +, *, –, .AND., .OR., .EQV., or .NEQV.

intrinsic

Is one of the following: MAX, MIN, IAND, IOR, or IEOR.

list

Is the name of one or more scalar variables or intrinsic type or common blocks that are accessible to the scoping unit. Subobjects cannot be specified. Each name must be separated by a comma, and a named common block must appear between slashes (/ /).

Variables that appear in a REDUCTION clause must be SHARED in the enclosing context. A private copy of each variable in *list* is created for each thread as if the PRIVATE clause had been used. The private copy is initialized according to the operator (see Table 14-3).

At the end of the REDUCTION, the shared variable is updated to reflect the result of combining the original value of the shared reduction variable with the final value of each of the private copies using the operator specified. The reduction operators are all associative (except for subtraction), and the compiler can freely reassociate the computation of the final value; the partial results of a subtraction reduction are added to form the final value.

The value of the shared variable becomes undefined when the first thread reaches the clause containing the reduction, and it remains undefined until the reduction computation is complete. Normally, the computation is complete at the end of the REDUCTION construct.

However, if the REDUCTION clause is used in a construct to which NOWAIT is also applied, the shared variable remains undefined until a barrier synchronization has been performed. This ensures that all the threads complete the REDUCTION clause.

The REDUCTION clause must be used in a region or worksharing construct where the reduction variable is used only in a reduction statement having one of the following forms:

```
x = x operator expr
x = expr operator x (except for subtraction)
x = intrinsic (x, expr)
x = intrinsic (expr, x)
```

Some reductions can be expressed in other forms. For instance, a MAX reduction can be expressed as follows:

```
IF (x .LT. expr) x = expr
```

Alternatively, the reduction might be hidden inside a subroutine call. Be careful that the operator you specify in the REDUCTION clause matches the reduction operation.

Table 14-3 lists the operators and intrinsics and their initialization values. The actual initialization value will be consistent with the data type of the reduction variable.

Table 14-3 Initialization Values for REDUCTION Operators and Intrinsics

Operator	Initialization Value
+	0
*	1
–	0
.AND.	.TRUE.
.OR.	.FALSE.
.EQV.	.TRUE.
.NEQV.	.FALSE.

Table 14-3 Initialization Values for REDUCTION Operators and Intrinsics

Intrinsic	Initialization Value
MAX	Smallest representable number
MIN	Largest representable number
IAND	All bits on
IOR	0
IEOR	0

If a directive allows reduction clauses, the number you can specify is not limited. However, each variable name can appear in only one of the clauses.

SHARED Clause

The SHARED clause specifies variables that will be shared by all the threads in a team. It takes the following form:

SHARED (*list*)

list

Is the name of one or more variables or common blocks that are accessible to the scoping unit. Subobjects cannot be specified. Each name must be separated by a comma, and a named common block must appear between slashes (/ /).

All threads within a team access the same storage area for SHARED data.

Conditional Compilation Rules

The OpenMP Fortran API lets you conditionally compile Intel Fortran statements if you use the appropriate directive prefix.

The prefix depends on which source form you are using, although !$ is valid in all forms.

The prefix must be followed by a valid Intel Fortran statement on the same line.

Free Source Form

The free source form conditional compilation prefix is !$. This prefix can appear in any column as long as it is preceded only by white space. It must appear as a single word with no intervening white space. Free-form source rules apply to the directive line.

Initial lines must have a space after the prefix. Continued lines must have an ampersand as the last nonblank character on the line. Continuation lines can have an ampersand after the prefix with optional white space before and after the ampersand.

Fixed Source Form

For fixed source form programs, the conditional compilation prefix is one of the following: !$, C$ (or c$), or *$.

The prefix must start in column one and appear as a single string with no intervening white space. Fixed-form source rules apply to the directive line.

Initial lines must have a space or zero in column six, and continuation lines must have a character other than a space or zero in column six. For example, the following forms for specifying conditional compilation are equivalent:

```
c23456789
!$    IAM = OMP_GET_THREAD_NUM( ) +
!$   *      INDEX

#IFDEF _OPENMP
      IAM = OMP_GET_THREAD_NUM( ) +
     *      INDEX
#ENDIF
```

See Also

- Your user's guide for details on how the conditional prefix is interpreted, and for details on a macro that can be used to denote conditional compilation

Nesting and Binding Rules

This section describes the dynamic nesting and binding rules for OpenMP Fortran API directives.

Binding Rules

The following rules apply to dynamic binding:

- The DO, SECTIONS, SINGLE, MASTER, and BARRIER directives bind to the dynamically enclosing PARALLEL directive, if one exists.
- The ORDERED directive binds to the dynamically enclosing DO directive.
- The ATOMIC directive enforces exclusive access with respect to ATOMIC directives in all threads, not just the current team.
- The CRITICAL directive enforces exclusive access with respect to CRITICAL directives in all threads, not just the current team.
- A directive can never bind to any directive outside the closest enclosing PARALLEL directive.

Nesting Rules

The following rules apply to dynamic nesting:

- A PARALLEL directive dynamically inside another PARALLEL directive logically establishes a new team, which is composed of only the current thread unless nested parallelism is enabled.
- DO, SECTIONS, and SINGLE directives that bind to the same PARALLEL directive are not allowed to be nested one inside the other.
- DO, SECTIONS, and SINGLE directives are not permitted in the dynamic extent of CRITICAL and MASTER directives.
- BARRIER directives are not permitted in the dynamic extent of DO, SECTIONS, SINGLE, MASTER, and CRITICAL directives.
- MASTER directives are not permitted in the dynamic extent of DO, SECTIONS, and SINGLE directives.
- ORDERED sections are not allowed in the dynamic extent of CRITICAL sections.
- Any directive set that is legal when executed dynamically inside a PARALLEL region is also legal when executed outside a parallel region. When executed dynamically outside a user-specified parallel region, the directive is executed with respect to a team composed of only the master thread.

Examples

The following example shows nested PARALLEL regions:

```
c$OMP PARALLEL DEFAULT(SHARED)
c$OMP DO
      DO I =1, N
c$OMP PARALLEL SHARED(I,N)
c$OMP DO
      DO J =1, N
        CALL WORK(I,J)
      END DO
c$OMP END   PARALLEL
      END DO
c$OMP END PARALLEL
```

Note that the inner and outer DO directives bind to different PARALLEL regions.

The following example shows a variation of the preceding example:

```
c$OMP PARALLEL DEFAULT(SHARED)
c$OMP DO
      DO I =1, N
```

```
        CALL SOME_WORK(I,N)
      END DO
c$OMP END PARALLEL
      ...
      SUBROUTINE SOME_WORK(I,N)
c$OMP PARALLEL DEFAULT(SHARED)
c$OMP DO
      DO J =1, N
        CALL WORK(I,J)
      END DO
c$OMP END PARALLEL
      RETURN
      END
```

ATOMIC Directive

The ATOMIC directive ensures that a specific memory location is updated dynamically; this prevents the possibility of multiple, simultaneous writing threads. It takes the following form:

c$OMP ATOMIC

c

Is one of the following: C (or c), !, or * (see "Syntax Rules for Compiler Directives").

Rules and Behavior

The ATOMIC directive permits optimization beyond that of the critical section around the assignment. An implementation can replace ATOMIC directives by enclosing each statement in a critical section. The critical section (or sections) must use the same unique name.

The ATOMIC directive applies only to the immediately following statement, which must have one of the following forms:

```
x = x operator expr
x = expr operator x
x = intrinsic (x, expr)
x = intrinsic (expr, x)
```

In the preceding statements:

- *x* is a scalar variable of intrinsic type
- *operator* is +, *, -, /, .AND., .OR., .EQV., or .NEQV.
- *expr* is a scalar expression that does not reference *x*
- *intrinsic* is MAX, MIN, IAND, IOR, or IEOR

All references to storage location *x* must have the same type and type parameters.

Only the loading and storing of *x* are dynamic; the evaluation of *expr* is not dynamic. To avoid race conditions (or concurrency races), all updates of the location in parallel must be protected using the ATOMIC directive, except those that are known to be free of race conditions. The function *intrinsic*, the operator *operator*, and the assignment must be the intrinsic function, operator, and assignment.

Example

The following example shows a way to avoid race conditions by using ATOMIC to protect all simultaneous updates of the location by multiple threads:

```
c$OMP PARALLEL DO DEFAULT(PRIVATE) SHARED(X,Y,INDEX,N)
      DO I=1,N
        CALL WORK(XLOCAL, YLOCAL)

c$OMP ATOMIC
        X(INDEX(I)) = X(INDEX(I)) + XLOCAL
        Y(I) = Y(I) + YLOCAL
      END DO
```

Since the ATOMIC directive applies only to the statement immediately following it, note that Y is *not* updated atomically.

BARRIER Directive

The BARRIER directive synchronizes all the threads in a team. It causes each thread to wait until all of the other threads in the team have reached the barrier.

The BARRIER directive takes the following form:

c$OMP BARRIER

c

Is one of the following: C (or c), !, or * (see "Syntax Rules for Compiler Directives").

The BARRIER directive must be encountered by all threads in a team or by none at all. It must also be encountered in the same order by all threads in a team.

Example

The directive binding rules call for a BARRIER directive to bind to the closest enclosing PARALLEL directive. In the following example, the BARRIER directive ensures that all threads have executed the first loop and that it is safe to execute the second loop:

```
c$OMP PARALLEL
```

```
c$OMP DO PRIVATE(i)
      DO i = 1, 100
        b(i) = i
      END DO
c$OMP BARRIER
c$OMP DO PRIVATE(i)
      DO i = 1, 100
        a(i) = b(101-i)
      END DO
c$OMP END PARALLEL
```

See Also

"Nesting and Binding Rules" for details about directive binding

CRITICAL Directive

The CRITICAL directive restricts access to a block of code to only one thread at a time. It takes the following form:

c$OMP CRITICAL [(*name*)]
  block
c$OMP END CRITICAL [(*name*)]

c

Is one of the following: C (or c), !, or * (see "Syntax Rules for Compiler Directives").

name

Is the name of the critical section.

block

Is a structured block (section) of statements or constructs. You cannot branch into or out of the block.

Rules and Behavior

A thread waits at the beginning of a critical section until no other thread in the team is executing a critical section having the same name. All unnamed CRITICAL directives map to the same name.

If a name is specified in the CRITICAL directive, the same name must appear in the corresponding END CRITICAL directive. If no name appears in the CRITICAL directive, no name can appear in the corresponding END CRITICAL directive.

Critical section names are global entities of the program. If the name specified conflicts with any other entity, the behavior of the program is undefined.

Example

The following example shows a queuing model in which a task is dequeued and worked on. To guard against multiple threads dequeuing the same task, the dequeuing operation is placed in a critical section.

Because there are two independent queues in this example, each queue is protected by CRITICAL directives having different names, XAXIS and YAXIS, respectively:

```
c$OMP PARALLEL DEFAULT(PRIVATE) SHARED(X,Y)
c$OMP CRITICAL(XAXIS)
      CALL DEQUEUE(IX_NEXT, X)
c$OMP END CRITICAL(XAXIS)
      CALL WORK(IX_NEXT, X)
c$OMP CRITICAL(YAXIS)
      CALL DEQUEUE(IY_NEXT,Y)
c$OMP END CRITICAL(YAXIS)
      CALL WORK(IY_NEXT, Y)
c$OMP END PARALLEL
```

DO Directive

The DO directive specifies that the iterations of the immediately following DO loop must be executed in parallel. It takes the following form:

c$OMP DO [*clause*[[,] *clause*] . . .]
  do_loop
[*c*$OMP END DO [NOWAIT]]

c

Is one of the following: C (or c), !, or * (see "Syntax Rules for Compiler Directives").

clause

Is one of the following:

- FIRSTPRIVATE (*list*)
  See "FIRSTPRIVATE Clause".
- LASTPRIVATE (*list*)
  See "LASTPRIVATE Clause".

- ORDERED
  Must be used if ordered sections are contained in the dynamic extent of the DO directive. For more information about ordered sections, see the description in "ORDERED Directive".
- PRIVATE (*list*)
  See "PRIVATE Clause".
- REDUCTION (*operator* | *intrinsic* : *list*)
  See "REDUCTION Clause".
- SCHEDULE (*type* [, *chunk*])
  Specifies how iterations of the DO loop are divided among the threads of the team. *chunk* must be a positive scalar integer expression. The following four *type*s are permitted, three of which allow the optional parameter *chunk*:

Type	Effect
STATIC	Divides iterations into contiguous pieces by dividing the number of iterations by the number of threads in the team. Each piece is then dispatched to a thread before loop execution begins. If *chunk* is specified, iterations are divided into pieces of a size specified by *chunk*. The pieces are statically dispatched to threads in the team in a round-robin fashion in the order of the thread number.
DYNAMIC	Can be used to get a set of iterations dynamically. It defaults to 1 unless *chunk* is specified. If *chunk* is specified, the iterations are broken into pieces of a size specified by *chunk*. As each thread finishes a piece of the iteration space, it dynamically gets the next set of iterations.
GUIDED	Can be used to specify a minimum number of iterations. It defaults to 1 unless *chunk* is specified. If *chunk* is specified, the chunksize is reduced exponentially with each succeeding dispatch. The *chunk* specifies the minimum number of iterations to dispatch each time. If there are less than *chunk* iterations remaining, the rest are dispatched.
RUNTIME[1]	Defers the scheduling decision until run time. You can choose a schedule type and chunksize at run time by using the environment variable OMP_SCHEDULE.

1. No *chunk* is permitted for this type.

If the SCHEDULE clause is not used, the default schedule type is STATIC.

do_loop

Is a DO iteration (an iterative DO loop). It cannot be a DO WHILE or a DO loop without loop control. The DO loop iteration variable must be of type integer.

The iterations of the DO loop are distributed across the existing team of threads. The values of the loop control parameters of the DO loop associated with a DO directive must be the same for all the threads in the team.

You cannot branch out of a DO loop associated with a DO directive.

Rules and Behavior

If used, the END DO directive must appear immediately after the end of the loop. If you do not specify an END DO directive, an END DO directive is assumed at the end of the DO loop.

If you specify the NOWAIT clause in the END DO directive, threads do not synchronize at the end of the parallel loop. Threads that finish early proceed straight to the instruction following the loop without waiting for the other members of the team to finish the DO directive.

Parallel DO loop control variables are block-level entities within the DO loop. If the loop control variable also appears in the LASTPRIVATE list of the parallel DO, it is copied out to a variable of the same name in the enclosing PARALLEL region. The variable in the enclosing PARALLEL region must be SHARED if it is specified in the LASTPRIVATE list of a DO directive.

Only a single SCHEDULE clause and ORDERED clause can appear in a DO directive.

DO directives must be encountered by all threads in a team or by none at all. It must also be encountered in the same order by all threads in a team.

Examples

In the following example, the loop iteration variable is private by default, and it is not necessary to explicitly declare it. The END DO directive is optional:

```
c$OMP PARALLEL
c$OMP DO
      DO I=1,N
        B(I) = (A(I) + A(I-1)) / 2.0
      END DO
c$OMP END DO
c$OMP END PARALLEL
```

If there are multiple independent loops within a parallel region, you can use the NOWAIT clause to avoid the implied BARRIER at the end of the DO directive, as follows:

```
c$OMP PARALLEL
c$OMP DO
```

```
      DO I=2,N
        B(I) = (A(I) + A(I-1)) / 2.0
      END DO
c$OMP END DO NOWAIT
c$OMP DO
      DO I=1,M
        Y(I) = SQRT(Z(I))
      END DO
c$OMP END DO NOWAIT
c$OMP END PARALLEL
```

Correct execution sometimes depends on the value that the last iteration of a loop assigns to a variable. Such programs must list all such variables as arguments to a LASTPRIVATE clause so that the values of the variables are the same as when the loop is executed sequentially, as follows:

```
c$OMP PARALLEL
c$OMP DO LASTPRIVATE(I)
      DO I=1,N
        A(I) = B(I) + C(I)
      END DO
c$OMP END PARALLEL
      CALL REVERSE(I)
```

In this case, the value of I at the end of the parallel region equals N+1, as in the sequential case.

Ordered sections are useful for sequentially ordering the output from work that is done in parallel. Assuming that a reentrant I/O library exists, the following program prints out the indexes in sequential order:

```
c$OMP DO ORDERED SCHEDULE(DYNAMIC)
      DO I=LB,UB,ST
        CALL WORK(I)
      END DO
      ...
      SUBROUTINE WORK(K)
c$OMP ORDERED
      WRITE(*,*) K
c$OMP END ORDERED
```

See Also

"Rules for General Directives that Affect DO Loops"

FLUSH Directive

The FLUSH directive identifies synchronization points at which the implementation must provide a consistent view of memory. It takes the following form:

c$OMP FLUSH [(*list*)]

c

Is one of the following: C (or c), !, or * (see "Syntax Rules for Compiler Directives").

list

Is the name of one or more variables to be flushed. Names must be separated by commas.

Rules and Behavior

The FLUSH directive must appear at the precise point in the code at which the synchronization is required. To avoid flushing all variables, specify a *list*.

Thread-visible variables are written back to memory at the point at which this directive appears. Modifications to thread-visible variables are visible to all threads after this point. Subsequent reads of thread-visible variables fetch the latest copy of the data.

Thread-visible variables include the following data items:

- Globally visible variables (common blocks and modules)
- Local variables that do not have the SAVE attribute but have had their address taken and saved or have had their address passed to another subprogram
- Local variables that do not have the SAVE attribute but are declared shared in a parallel region within the subprogram
- Dummy arguments
- All pointer dereferences

The FLUSH directive is implied for the following directives (unless the NOWAIT keyword is used):

- BARRIER
- CRITICAL and END CRITICAL
- END DO
- END SECTIONS
- END SINGLE
- ORDERED and END ORDERED
- PARALLEL and END PARALLEL
- PARALLEL DO and END PARALLEL DO
- PARALLEL SECTIONS and END PARALLEL SECTIONS

Example

The following example uses the FLUSH directive for point-to-point synchronization between pairs of threads:

```
c$OMP PARALLEL DEFAULT(PRIVATE) SHARED(ISYNC)
      IAM = GET_THREAD_NUM()
      ISYNC(IAM) = 0
c$OMP BARRIER
      CALL WORK( )
C I AM DONE WITH MY WORK, SYNCHRONIZE WITH MY NEIGHBOR
      ISYNC(IAM) = 1
c$OMP FLUSH(ISYNC)
C WAIT TILL NEIGHBOR IS DONE
      DO WHILE (ISYNC(NEIGH) .EQ. 0)
c$OMP FLUSH(ISYNC)
      END DO
c$OMP END PARALLEL
```

MASTER Directive

The MASTER directive specifies a block of code to be executed by the master thread of the team. It takes the following form:

c$OMP MASTER
  block
c$OMP END MASTER

c

Is one of the following: C (or c), !, or * (see "Syntax Rules for Compiler Directives").

block

Is a structured block (section) of statements or constructs. You cannot branch into or out of the block.

Rules and Behavior

When the MASTER directive is specified, the other threads in the team skip the enclosed block (section) of code and continue execution. There is no implied barrier, either on entry to or exit from the master section.

Example

The following example forces the master thread to execute the routines OUTPUT and INPUT:

```
c$OMP PARALLEL DEFAULT(SHARED)
      CALL WORK(X)
c$OMP MASTER
      CALL OUTPUT(X)
      CALL INPUT(Y)
c$OMP END MASTER
      CALL WORK(Y)
c$OMP END PARALLEL
```

ORDERED Directive

The ORDERED directive specifies a block of code to be executed in the order in which iterations would be executed in sequential execution. It takes the following form:

c$OMP ORDERED
  block
c$OMP END ORDERED

c

Is one of the following: C (or c), !, or * (see "Syntax Rules for Compiler Directives").

block

Is a structured block (section) of statements or constructs. You cannot branch into or out of the block.

Rules and Behavior

An ORDERED directive can appear only in the dynamic extent of a DO or PARALLEL DO directive. The DO directive to which the ordered section binds *must* have the ORDERED clause specified.

An iteration of a loop using a DO directive must not execute the same ORDERED directive more than once, and it must not execute more than one ORDERED directive.

One thread is allowed in an ordered section at a time. Threads are allowed to enter in the order of the loop iterations. No thread can enter an ordered section until it can be guaranteed that all previous iterations have completed or will never execute an ordered section. This sequentializes and orders code within ordered sections while allowing code outside the section to run in parallel.

Ordered sections that bind to different DO directives are independent of each other.

Example

Ordered sections are useful for sequentially ordering the output from work that is done in parallel. Assuming that a reentrant I/O library exists, the following program prints out the indexes in sequential order:

```
c$OMP DO ORDERED SCHEDULE(DYNAMIC)
      DO I=LB,UB,ST
        CALL WORK(I)
      END DO
      ...
      SUBROUTINE WORK(K)
c$OMP ORDERED
      WRITE(*,*) K
c$OMP END ORDERED
```

PARALLEL Directive

The PARALLEL directive defines a parallel region. It takes the following form:

c$OMP PARALLEL [*clause*[[,] *clause*] . . .]
  block
c$OMP END PARALLEL

c

Is one of the following: C (or c), !, or * (see "Syntax Rules for Compiler Directives").

clause

Is one of the following:

- COPYIN (*list*)
  See "COPYIN Clause".
- DEFAULT (PRIVATE | SHARED | NONE)
  See "DEFAULT Clause".
- FIRSTPRIVATE (*list*)
  See "FIRSTPRIVATE Clause".
- IF (*scalar_logical_expression*)
  Specifies that the enclosed code section is to be executed in parallel only if the *scalar_logical_expression* evaluates to .TRUE.. Otherwise, the parallel region is serialized. If this clause is not used, the region is executed as if an IF(.TRUE.) clause were specified.
  This clause is evaluated by the master thread before any data scope attributes take effect.
  Only a single IF clause can appear in the directive.

- NUM_THREADS (*scalar_integer_expression*)

  Specifies the number of threads to be used in a parallel region. The *scalar_integer_expressio*n must evaluate to a positive scalar integer value. Only a single NUM_THREADS clause can appear in the directive.
- PRIVATE (*list*)

  See "PRIVATE Clause".
- REDUCTION (*operator* | *intrinsic* : *list*)

  See "REDUCTION Clause".
- SHARED (*list*)

  See "SHARED Clause".

block

Is a structured block (section) of statements or constructs. You cannot branch into or out of the block (the parallel region).

Rules and Behavior

The PARALLEL and END PARALLEL directive pair must appear in the same routine in the executable section of the code.

The END PARALLEL directive denotes the end of the parallel region. There is an implied barrier at this point. Only the master thread of the team continues execution at the end of a parallel region.

The number of threads in the team can be controlled by the NUM_THREADS clause, the environment variable OMP_NUM_THREADS, or by calling the run-time library routine OMP_SET_NUM_THREADS from a serial portion of the program.

NUM_THREADS supersedes the OMP_SET_NUM_THREADS routine, which supersedes the OMP_NUM_THREADS environment variable. Subsequent parallel regions, however, are not affected unless they have their own NUM_THREADS clauses.

Once specified, the number of threads in the team remains constant for the duration of that parallel region.

If the dynamic threads mechanism is enabled by an environment variable or a library routine, then the number of threads requested by the NUM_THREADS clause is the maximum number to use in the parallel region.

The code contained within the dynamic extent of the parallel region is executed on each thread, and the code path can be different for different threads.

If a thread executing a parallel region encounters another parallel region, it creates a new team and becomes the master of that new team. By default, nested parallel regions are always serialized and executed by a team of one thread.

Examples

You can use the PARALLEL directive in coarse-grain parallel programs. In the following example, each thread in the parallel region decides what part of the global array X upon which to work based on the thread number:

```
c$OMP PARALLEL DEFAULT(PRIVATE) SHARED(X,NPOINTS)
      IAM = OMP_GET_THREAD_NUM()
      NP = OMP_GET_NUM_THREADS()
      IPOINTS = NPOINTS/NP
      CALL SUBDOMAIN(X,IAM,IPOINTS)
c$OMP END PARALLEL
```

Assuming you previously used the environment variable OMP_NUM_THREADS to set the number of threads to six, you can change the number of threads between parallel regions as follows:

```
      CALL OMP_SET_NUM_THREADS(3)
!$OMP PARALLEL
...
!$OMP END PARALLEL
      CALL OMP_SET_NUM_THREADS(4)
!$OMP PARALLEL DO
...
!$OMP END PARALLEL DO
```

See Also

- "OpenMP* Fortran Routines"
- Your user's guide for details on environment variables

PARALLEL DO Directive

The PARALLEL DO directive provides an abbreviated way to specify a parallel region containing a single DO directive.

The PARALLEL DO directive takes the following form:

c$OMP PARALLEL DO [*clause*[[,] *clause*] . . .]
  do_loop
[*c*$OMP END PARALLEL DO]

c

Is one of the following: C (or c), !, or * (see "Syntax Rules for Compiler Directives").

clause

Can be any of the clauses accepted by the DO or PARALLEL directives. See "DO Directive" and "PARALLEL Directive".

do_loop

Is a DO iteration (a DO loop). It cannot be a DO WHILE or a DO loop without loop control. The DO loop iteration variable must be of type integer.

You cannot branch out of a DO loop associated with a DO directive.

Rules and Behavior

If the END PARALLEL DO directive is not specified, the PARALLEL DO is assumed to end with the DO loop that immediately follows the PARALLEL DO directive. If used, the END PARALLEL DO directive must appear immediately after the end of the DO loop.

The semantics are identical to explicitly specifying a PARALLEL directive immediately followed by a DO directive.

Examples

In the following example, the loop iteration variable is private by default and it is not necessary to explicitly declare it. The END PARALLEL DO directive is optional:

```
c$OMP PARALLEL DO
      DO I=1,N
        B(I) = (A(I) + A(I-1)) / 2.0
      END DO
c$OMP END PARALLEL DO
```

The following example shows how to use the REDUCTION clause in a PARALLEL DO directive:

```
c$OMP PARALLEL DO DEFAULT(PRIVATE) REDUCTION(+: A,B)
      DO I=1,N
        CALL WORK(ALOCAL,BLOCAL)
        A = A + ALOCAL
        B = B + BLOCAL
      END DO
c$OMP END PARALLEL DO
```

See Also

"Rules for General Directives that Affect DO Loops"

PARALLEL SECTIONS Directive

The PARALLEL SECTIONS directive provides an abbreviated way to specify a parallel region containing a single SECTIONS directive. The semantics are identical to explicitly specifying a PARALLEL directive immediately followed by a SECTIONS directive.

The PARALLEL SECTIONS directive takes the following form:

c$OMP PARALLEL SECTIONS [*clause*[[,] *clause*] . . .]
[*c*$OMP SECTION]
block
[*c*$OMP SECTION
block] ...
c$OMP END PARALLEL SECTIONS

c

Is one of the following: C (or c), !, or * (see "Syntax Rules for Compiler Directives").

clause

Can be any of the clauses accepted by the PARALLEL or SECTIONS directives. See "PARALLEL Directive" and "SECTIONS Directive".

block

Is a structured block (section) of statements or constructs. You cannot branch into or out of the block.

The last section ends at the END PARALLEL SECTIONS directive.

Example

In the following example, subroutines XAXIS, YAXIS, and ZAXIS can be executed concurrently:

```
c$OMP PARALLEL SECTIONS
c$OMP SECTION
      CALL XAXIS
c$OMP SECTION
      CALL YAXIS
c$OMP SECTION
      CALL ZAXIS
c$OMP END PARALLEL SECTIONS
```

SECTIONS Directive

The SECTIONS directive specifies one or more blocks of code that must be divided among threads in the team. Each section is executed once by a thread in the team.

The SECTIONS directive takes the following form:

c$OMP SECTIONS [*clause*[[,] *clause*] . . .]
[*c*$OMP SECTION]
block
[*c*$OMP SECTION
block] ...
c$OMP END SECTIONS [NOWAIT]

c

Is one of the following: C (or c), !, or * (see "Syntax Rules for Compiler Directives").

clause

Is one of the following:

- FIRSTPRIVATE (*list*)
  See "FIRSTPRIVATE Clause".
- LASTPRIVATE (*list*)
  See "LASTPRIVATE Clause".
- PRIVATE (*list*)
  See "PRIVATE Clause".
- REDUCTION (*operator* | *intrinsic* : *list*)
  See "REDUCTION Clause".

block

Is a structured block (section) of statements or constructs. Any constituent section must also be a structured block.

You cannot branch into or out of the block.

Rules and Behavior

Each section of code is preceded by a SECTION directive, although the directive is optional for the first section. The SECTION directives must appear within the lexical extent of the SECTIONS and END SECTIONS directive pair.

The last section ends at the END SECTIONS directive. Threads that complete execution of their SECTIONs encounter an implied barrier at the END SECTIONS directive unless NOWAIT is specified.

SECTIONS directives must be encountered by all threads in a team or by none at all. It must also be encountered in the same order by all threads in a team.

Example

In the following example, subroutines XAXIS, YAXIS, and ZAXIS can be executed concurrently:

```
c$OMP PARALLEL
c$OMP SECTIONS
c$OMP SECTION
      CALL XAXIS
c$OMP SECTION
      CALL YAXIS
c$OMP SECTION
      CALL ZAXIS
c$OMP END SECTIONS
c$OMP END PARALLEL
```

SINGLE Directive

The SINGLE directive specifies that a block of code is to be executed by only one thread in the team. It takes the following form:

c$OMP SINGLE [*clause*[[,] *clause*] . . .]
  block
c$OMP END SINGLE [*modifier*]

c

Is one of the following: C (or c), !, or * (see "Syntax Rules for Compiler Directives").

clause

Is one of the following:

- FIRSTPRIVATE (*list*)
  See "FIRSTPRIVATE Clause".
- PRIVATE (*list*)
  See "PRIVATE Clause".

block

- Is a structured block (section) of statements or constructs. You cannot branch into or out of the block.

modifier

Is one of the following:

- COPYPRIVATE (list)
  See "COPYPRIVATE Clause".
- NOWAIT

Rules and Behavior

Threads in the team that are not executing this directive wait at the END SINGLE directive unless NOWAIT is specified.

SINGLE directives must be encountered by all threads in a team or by none at all. It must also be encountered in the same order by all threads in a team.

Example

In the following example, the first thread that encounters the SINGLE directive executes subroutines OUTPUT and INPUT:

```
c$OMP PARALLEL DEFAULT(SHARED)
      CALL WORK(X)
c$OMP BARRIER
c$OMP SINGLE
      CALL OUTPUT(X)
      CALL INPUT(Y)
c$OMP END SINGLE
      CALL WORK(Y)
c$OMP END PARALLEL
```

You should not make assumptions as to which thread executes the SINGLE section. All other threads skip the SINGLE section and stop at the barrier at the END SINGLE construct. If other threads can proceed without waiting for the thread executing the SINGLE section, you can specify NOWAIT in the END SINGLE directive.

THREADPRIVATE Directive

The THREADPRIVATE directive specifies named common blocks to be private (local) to a thread; they are global within the thread. It takes the following form:

c$OMP THREADPRIVATE(/*cb*/ [, /*cb*/]...)

c

Is one of the following: C (or c), !, or * (see "Syntax Rules for Compiler Directives").

cb

Is the name of the common block you want made private to a thread. Only named common blocks can be made thread private. Note that the slashes (/) are required.

Rules and Behavior

Each thread gets its own copy of the common block, so data written to the common block by one thread is not directly visible to other threads.

During serial portions and MASTER sections of the program, accesses are to the master thread copy of the common block. On entry to the first parallel region, data in the THREADPRIVATE common blocks should be assumed to be undefined unless a COPYIN clause is specified in the PARALLEL directive.

When a common block (which is initialized using DATA statements) appears in a THREADPRIVATE directive, each thread copy is initialized once prior to its first use. For subsequent parallel regions, data in THREADPRIVATE common blocks are guaranteed to persist only if the dynamic threads mechanism has been disabled and if the number of threads are the same for all the parallel regions.

A THREADPRIVATE common block or its constituent variables can appear only in a COPYIN clause. They are not permitted in a PRIVATE, FIRSTPRIVATE, LASTPRIVATE, SHARED, or REDUCTION clause. They are not affected by the DEFAULT clause.

Example

In the following example, the common blocks BLK1 and FIELDS are specified as thread private:

```
      COMMON /BLK/ SCRATCH
      COMMON /FIELDS/ XFIELD, YFIELD, ZFIELD

c$OMP THREADPRIVATE(/BLK/,/FIELDS/)
c$OMP PARALLEL DEFAULT(PRIVATE) COPYIN(/BLK1/,ZFIELD)
```

Scope and Association

15

Program entities are identified by names, labels, input/output unit numbers, operator symbols, or assignment symbols. For example, a variable, a derived type, or a subroutine is identified by its name.

Scope refers to the area in which a name is recognized. A scoping unit is the program or part of a program in which a name is defined or known. It can be any of the following:

- An entire executable program
- A single scoping unit
- A single statement (or part of a statement)

The region of the program in which a name is known and accessible is referred to as the scope of that name. These different scopes allow the same name to be used for different things in different regions of the program.

Association is the language concept that allows different names to refer to the same entity in a particular region of a program.

This chapter contains information on the following topics:

- "Scope"
- "Unambiguous Generic Procedure References"
- "Resolving Procedure References"
- "Association"

Scope

Program entities have the following kinds of scope (as shown in Table 15-1):

- Global

  Entities that are accessible throughout an executable program. The name of a global entity must be unique. It cannot be used to identify any other global entity in the same executable program.

- Scoping unit (Local scope)

  Entities that are declared within a scoping unit. These entities are local to that scoping unit. The names of local entities are divided into classes (see Table 15-1).

  A *scoping unit* is one of the following:

  — A derived-type definition

  — A procedure interface body (excluding any derived-type definitions and interface bodies contained within it)

  — A program unit or subprogram (excluding any derived-type definitions, interface bodies, and subprograms contained within it)

  A scoping unit that immediately surrounds another scoping unit is called the host scoping unit. Named entities within the host scoping unit are accessible to the nested scoping unit by host association. (For information about host association, see "Use and Host Association".)

  Once an entity is declared in a scoping unit, its name can be used throughout that scoping unit. An entity declared in another scoping unit is a different entity even if it has the same name and properties.

  Within a scoping unit, a local entity name that is not generic must be unique within its class. However, the name of a local entity in one class can be used to identify a local entity of another class.

  Within a scoping unit, a generic name can be the same as any one of the procedure names in the interface block.

  A component name has the same scope as the derived type of which it is a component. It can appear only within a component designator of a structure of that type.

  For information on interactions between local and global names, see Table 15-1.

- Statement

  Entities that are accessible only within a statement or part of a statement; such entities cannot be referenced in subsequent statements.

  The name of a statement entity can also be the name of a global or local entity in the same scoping unit; in this case, the name is interpreted within the statement as that of the statement entity.

Table 15-1 Scope of Program Entities

Entity	Scope	
Program units	Global	
Common blocks[1]	Global	
External procedures	Global	
Intrinsic procedures	Global[2]	
Module procedures	Local	Class I

Table 15-1 Scope of Program Entities

Entity	Scope	
Internal procedures	Local	Class I
Dummy procedures	Local	Class I
Statement functions	Local	Class I
Derived types	Local	Class I
Components of derived types	Local	Class II
Named constants	Local	Class I
Named constructs	Local	Class I
Namelist group names	Local	Class I
Generic identifiers	Local	Class I
Argument keywords in procedures	Local	Class III
Variables that can be referenced throughout a subprogram	Local	Class I
Variables that are dummy arguments in statement functions	Statement	
DO variables in an implied-DO list[3] of a DATA or FORALL statement, or an array constructor	Statement	
Intrinsic operators	Global	
Defined operators	Local	
Statement labels	Local	
External I/O unit numbers	Global	
Intrinsic assignment	Global[4]	
Defined assignment	Local	

1. Names of common blocks can also be used to identify local entities.
2. If an intrinsic procedure is not used in a scoping unit, its name can be used as a local entity within that scoping unit. For example, if intrinsic function COS is not used in a program unit, COS can be used as a local variable there.
3. The DO variable in an implied-DO list of an I/O list has local scope.
4. The scope of the assignment symbol (=) is global, but it can identify additional operations (see "Defining Generic Assignment").

Scoping units can contain other scoping units. For example, the following shows six scoping units:

```
MODULE MOD_1                        ! Scoping unit 1
   ...                              ! Scoping unit 1
CONTAINS                            ! Scoping unit 1
  FUNCTION FIRST                    ! Scoping unit 2
    TYPE NAME                       ! Scoping unit 3
    ...                             ! Scoping unit 3
    END TYPE NAME                   ! Scoping unit 3
```

```
    ...                          ! Scoping unit 2
  CONTAINS                       ! Scoping unit 2
    SUBROUTINE SUB_B             ! Scoping unit 4
      TYPE PROCESS               ! Scoping unit 5
      ...                        ! Scoping unit 5
      END TYPE PROCESS           ! Scoping unit 5
      INTERFACE                  ! Scoping unit 5
        SUBROUTINE SUB_A         ! Scoping unit 6
        ...                      ! Scoping unit 6
        END SUBROUTINE SUB_A     ! Scoping unit 6
      END INTERFACE              ! Scoping unit 5
    END SUBROUTINE SUB_B         ! Scoping unit 4
  END FUNCTION   FIRST           ! Scoping unit 2
END MODULE                       ! Scoping unit 1
```

See Also

- "Derived Data Types"
- "Use and Host Association"
- Chapter 9, "Intrinsic Procedures"
- Chapter 8, "Program Units and Procedures"
- "Defining Generic Names for Procedures" for details on user-defined generic procedures
- "Defining Generic Operators" for details on defined operations
- "Defining Generic Assignment" for details on defined assignment
- "PRIVATE and PUBLIC Attributes and Statements" for details on how the PRIVATE attribute can affect accessibility of entities

Unambiguous Generic Procedure References

When a generic procedure reference is made, a specific procedure is invoked. If the following rules are used, the generic reference will be unambiguous:

- Within a scoping unit, two procedures that have the same generic name must both be subroutines (or both be functions). One of the procedures must have a nonoptional dummy argument that is one of the following:
  - Not present by position or argument keyword in the other argument list
  - Is present, but has different type and kind parameters, or rank

- Within a scoping unit, two procedures that have the same generic operator must both have the same number of arguments or both define assignment. One of the procedures must have a dummy argument that corresponds by position in the argument list to a dummy argument of the other procedure that has a different type and kind parameters, or rank.

When an interface block extends an intrinsic procedure, operator, or assignment, the rules apply as if the intrinsic consists of a collection of specific procedures, one for each allowed set of arguments.

When a generic procedure is accessed from a module, the rules apply to all the specific versions, even if some of them are inaccessible by their specific names.

See Also

"Defining Generic Names for Procedures" for details on generic procedure names

Resolving Procedure References

The procedure name in a procedure reference is either established to be generic or specific, or is not established. The rules for resolving a procedure reference differ depending on whether the procedure is established and how it is established.

References to Generic Names

Within a scoping unit, a procedure name is established to be generic if any of the following is true:

- The scoping unit contains an interface block with that procedure name.
- The procedure name matches the name of a generic intrinsic procedure, and it is specified with the INTRINSIC attribute in that scoping unit.
- The procedure name is established to be generic in a module, and the scoping unit contains a USE statement making that procedure name accessible.
- The scoping unit contains no declarations for that procedure name, but the procedure name is established to be generic in a host scoping unit.

To resolve a reference to a procedure name established to be generic, the following rules are used in the order shown:

1. If an interface block with that procedure name appears in one of the following, the reference is to the specific procedure providing that interface:
   a. The scoping unit that contains the reference
   b. A module made accessible by a USE statement in the scoping unit

   The reference must be consistent with one of the specific interfaces of the interface block.

2. If the procedure name is specified with the INTRINSIC attribute in one of the following, the reference is to that intrinsic procedure:
   a. The same scoping unit
   b. A module made accessible by a USE statement in the scoping unit

   The reference must be consistent with the interface of that intrinsic procedure.
3. If the following is true, the reference is resolved by applying rules 1 and 2 to the host scoping unit:
   a. The procedure name is established to be generic in the host scoping unit
   b. There is agreement between the scoping unit and the host scoping unit as to whether the procedure is a function or subroutine name.
4. If none of the preceding rules apply, the reference must be to the generic intrinsic procedure with that name. The reference must be consistent with the interface of that intrinsic procedure.

Example

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
!
! shows how to use generic procedure references

USE UN_MOD
INTERFACE DUP
   MODULE PROCEDURE dup1, dup2, dup3
END INTERFACE

real a,b
integer c,d
character (len=2) state

a = 1.5
b = 2.32
c = 5
d = 47
state = 'WA'

call dup(a,b)
call dup(c,d)
print *, dup(state)       !actual output is 'S' only
END
```

Note that the function DUP3 only prints one character, since module UN_MOD specifies no length parameter for the function result.

If the dummy arguments x and y for DUP were declared as integers instead of reals, then any calls to DUP would be ambiguous. If this is the case, a compile-time error results.

The subroutine definitions, DUP1, DUP2, and DUP3, must have different names. The generic name is specified in the first line of the interface block, and in the example is DUP.

References to Specific Names

In a scoping unit, a procedure name is established to be specific if it is not established to be generic and any of the following is true:

- The scoping unit contains an interface body with that procedure name.
- The scoping unit contains an internal procedure, module procedure, or statement function with that procedure name.
- The procedure name is the same as the name of a generic intrinsic procedure, and it is specified with the INTRINSIC attribute in that scoping unit.

- The procedure name is specified with the EXTERNAL attribute in that scoping unit.
- The procedure name is established to be specific in a module, and the scoping unit contains a USE statement making that procedure name accessible.
- The scoping unit contains no declarations for that procedure name, but the procedure name is established to be specific in a host scoping unit.

To resolve a reference to a procedure name established to be specific, the following rules are used in the order shown:

1. If either of the following is true, the dummy argument is a dummy procedure and the reference is to that dummy procedure:
   a. The scoping unit is a subprogram, and it contains an interface body with that procedure name.
   b. The procedure name has been declared EXTERNAL, and the procedure name is a dummy argument of that subprogram.

   The procedure invoked by the reference is the one supplied as the corresponding actual argument.
2. If the scoping unit contains an interface body or the procedure name has been declared EXTERNAL, and Rule 1 does not apply, the reference is to an external procedure with that name.
3. If the scoping unit contains an internal procedure or statement function with that procedure name, the reference is to that entity.
4. If the procedure name has been declared INTRINSIC in the scoping unit, the reference is to the intrinsic procedure with that name.
5. If the scoping unit contains a USE statement that makes the name of a module procedure accessible, the reference is to that procedure. (The USE statement allows renaming, so the name referenced may differ from the name of the module procedure.)
6. If none of the preceding rules apply, the reference is resolved by applying these rules to the host scoping unit.

References to Nonestablished Names

In a scoping unit, a procedure name is not established if it is not determined to be generic or specific.

To resolve a reference to a procedure name that is not established, the following rules are used in the order shown:

1. If both of the following are true, the dummy argument is a dummy procedure and the reference is to that dummy procedure:
   a. The scoping unit is a subprogram.
   b. The procedure name is a dummy argument of that subprogram.

   The procedure invoked by the reference is the one supplied as the corresponding actual argument.
2. If both of the following are true, the procedure is an intrinsic procedure and the reference is to that intrinsic procedure:
   a. The procedure name matches the name of an intrinsic procedure.
   b. There is agreement between the intrinsic procedure definition and the reference of the name as a function or subroutine.
3. If neither of the preceding rules apply, the reference is to an external procedure with that name.

See Also

- "Function References"
- "USE Statement"
- "CALL Statement" for details on subroutine references
- "Defining Generic Names for Procedures" for details on generic procedure names

Association

Association allows different program units to access the same value through different names. Entities are associated when each is associated with the same storage location.

There are three kinds of association:

- "Name Association"
- "Pointer Association"
- "Storage Association"

Example 15-1 shows name, pointer, and storage association between an external program unit and an external procedure.

Example 15-1 Example of Name, Pointer, and Storage Association

```
! Scoping Unit 1: An external program unit
REAL A, B(4)
REAL, POINTER :: M(:)
REAL, TARGET :: N(12)
COMMON /COM/...
EQUIVALENCE (A, B(1))         ! Storage association between A and B(1)
M => N                        ! Pointer association
CALL P (actual-arg,...)
...
! Scoping Unit 2: An external procedure
SUBROUTINE P (dummy-arg,...)  ! Name and storage association between
                              !    these arguments and the calling
                              !    routine's arguments in scoping unit 1
  COMMON /COM/...             ! Storage association with common block COM
                              !    in scoping unit 1
  REAL Y
  CALL Q (actual-arg,...)
  CONTAINS
    SUBROUTINE Q (dummy-arg,...) ! Name and storage association between
                                 !   these arguments and the calling
                                 !   routine's arguments in host procedure
                                 !   P (subprogram Q has host association
                                 !   with procedure P)

     Y = 2.0*(Y-1.0)         ! Name association with Y in host procedure P
   ...
```

Name Association

Name association allows an entity to be accessed from different scoping units by the same name or by different names. There are three types of name association: argument, use, and host.

Argument Association

Arguments are the values passed to and from functions and subroutines through calling program argument lists.

Execution of a procedure reference establishes argument association between an actual argument and its corresponding dummy argument. The name of a dummy argument can be different from the name of its associated actual argument (if any).

When the procedure completes execution, the argument association is terminated.

See Also

"Argument Association"

Use and Host Association

Use association allows the entities in a module to be accessible to other scoping units. The mechanism for use association is the USE statement. The USE statement provides access to all public entities in the module, unless ONLY is specified. In this case, only the entities named in the ONLY list can be accessed.

Host association allows the entities in a host scoping unit to be accessible to an internal procedure, derived-type definition, or module procedure contained within the host. The accessed entities are known by the same name and have the same attributes as in the host. Entities that are local to a procedure are not accessible to its host.

Use or host association remains in effect throughout the execution of the executable program.

If an entity that is accessed by use association has the same nongeneric name as a host entity, the host entity is inaccessible. A name that appears in the scoping unit as an external name in an EXTERNAL statement is a global name, and any entity of the host that has this as its nongeneric name is inaccessible.

An interface body does not access named entities by host association, but it can access entities by use association.

If a procedure gains access to a pointer by host association, the association of the pointer with a target that is current at the time the procedure is invoked remains current within the procedure. This pointer association can be changed within the procedure. After execution of the procedure, the pointer association remains current, unless the execution caused the target to become undefined. If this occurs, the host associated pointer becomes undefined.

NOTE. *Implicit declarations can cause problems for host association. It is recommended that you use* IMPLICIT NONE *in both the host and the contained procedure, and that you explicitly declare all entities.*
When all entities are explicitly declared, local declarations override host declarations, and host declarations that are not overridden are available in the contained procedure.

The following example shows host and use association:

```
MODULE SHARE_DATA
  REAL Y, Z  END MODULE
PROGRAM DEMO
  USE SHARE_DATA         ! All entities in SHARE_DATA are available
  REAL B, Q              !   through use association.
  ...
  CALL CONS (Y)
CONTAINS
  SUBROUTINE CONS (Y)    ! Y is a local entity (dummy argument).
    REAL C, Y
    ...
    Y = B + C + Q + Z    ! B and Q are available through host association.
    ...                  !   C is a local entity, explicitly declared.  Z
  END SUBROUTINE CONS    !   is available through use association.
END PROGRAM DEMO
```

See Also

- "USE Statement"
- "Scope" for details on entities with local scope

Pointer Association

A pointer can be associated with a target. At different times during the execution of a program, a pointer can be undefined, associated with different targets, or be disassociated. The initial association status of a pointer is undefined. A pointer can become associated by the following:

- By pointer assignment (pointer => target)

  The target must be associated, or specified with the TARGET attribute. If the target is allocatable, it must be currently allocated.
- By allocation (successful execution of an ALLOCATE statement)

  The ALLOCATE statement must reference the pointer.

A pointer becomes disassociated if any of the following occur:

- The pointer is nullified by a NULLIFY statement.
- The pointer is deallocated by a DEALLOCATE statement.
- The pointer is assigned a disassociated pointer (or the NULL intrinsic function).

When a pointer is associated with a target, the definition status of the pointer is defined or undefined, depending on the definition status of the target. A target is undefined in the following cases:

- If it was never allocated
- If it is not deallocated through the pointer
- If a RETURN or END statement causes it to become undefined

If a pointer is associated with a definable target, the definition status of the pointer can be defined or undefined, according to the rules for a variable.

If the association status of a pointer is disassociated or undefined, the pointer must not be referenced or deallocated.

Whatever its association status, a pointer can always be nullified, allocated, or associated with a target. When a pointer is nullified, it is disassociated. When a pointer is allocated, it becomes associated, but is undefined. When a pointer is associated with a target, its association and definition status are determined by its target.

See Also

- "Pointer Assignments"
- "ALLOCATE Statement"
- "DEALLOCATE Statement"
- "NULLIFY Statement"
- "NULL"

Storage Association

Storage association is the association of two or more data objects. It occurs when two or more storage sequences share (or are aligned with) one or more *storage units*. Storage sequences are used to describe relationships among variables, common blocks, and result variables.

Storage Units and Storage Sequence

A *storage unit* is a fixed unit of physical memory allocated to certain data. A *storage sequence* is a sequence of storage units. The size of a storage sequence is the number of storage units in the storage sequence. A storage unit can be numeric, character, or unspecified.

A nonpointer scalar of type default real, integer, or logical occupies one numeric storage unit. A nonpointer scalar of type double precision real or default complex occupies two contiguous numeric storage units. In Intel® Fortran, one numeric storage unit corresponds to 4 bytes of memory.

A nonpointer scalar of type default character with character length 1 occupies one character storage unit. A nonpointer scalar of type default character with character length *len* occupies *len* contiguous character storage units. In Intel Fortran, one character storage unit corresponds to 1 byte of memory.

A nonpointer scalar of nondefault data type occupies a single unspecified storage unit. The number of bytes corresponding to the unspecified storage unit differs depending on the data type.

Table 15-2 lists the storage requirements (in bytes) for the intrinsic data types.

Table 15-2 Data Type Storage Requirements

Data Type	Storage Requirements (in bytes)
BYTE	1
LOGICAL	2, 4, or 8[1]
LOGICAL(1)	1
LOGICAL(2)	2
LOGICAL(4)	4
LOGICAL(8)	8
INTEGER	2, 4, or 8[1]
INTEGER(1)	1
INTEGER(2)	2
INTEGER(4)	4
INTEGER(8)	8
REAL	4, 8, or 16[2]
REAL(4)	4
DOUBLE PRECISION	8
REAL(8)	8
REAL(16)	16
COMPLEX	8, 16, or 32[2]
COMPLEX(4)	8
DOUBLE COMPLEX	16
COMPLEX(8)	16
COMPLEX(16)	32
CHARACTER	1
CHARACTER*len	len[3]

Table 15-2 Data Type Storage Requirements

Data Type	Storage Requirements (in bytes)
CHARACTER*(*)	assumed-length[4]

1. Depending on default integer, LOGICAL and INTEGER can have 2, 4, or 8 bytes. The default allocation is four bytes.
2. Depending on default real, REAL can have 4, 8, or 16 bytes and COMPLEX can have 8, 16, or 32 bytes. The default allocations are four bytes for REAL and eight bytes for COMPLEX.
3. The value of len is the number of characters specified. The largest valid value is2**31–1 on IA-32 processors; 2**63–1 on Intel® Itanium® processors. Negative values are treated as zero.
4. The assumed-length format *(*) applies to dummy arguments, PARAMETER statements, or character functions, and indicates that the length of the actual argument or function is used. (See "Assumed-Length Character Arguments" and your user's guide.)

A nonpointer scalar of sequence derived type occupies a sequence of storage sequences corresponding to the components of the structure, in the order they occur in the derived-type definition. (A sequence derived type has a SEQUENCE statement.)

A pointer occupies a single unspecified storage unit that is different from that of any nonpointer object and is different for each combination of type, type parameters, and rank.

The definition status and value of a data object affects the definition status and value of any storage-associated entity.

When two objects occupy the same storage sequence, they are totally storage-associated. When two objects occupy parts of the same storage sequence, they are partially associated. An EQUIVALENCE statement, a COMMON statement, or an ENTRY statement can cause total or partial storage association of storage sequences.

See Also

- "COMMON Statement"
- "ENTRY Statement"
- "EQUIVALENCE Statement"
- Your user's guide for details on the hardware representations of data types

Array Association

A nonpointer array occupies a sequence of contiguous storage sequences, one for each array element, in array element order.

Two or more arrays are associated when each one is associated with the same storage location. They are partially associated when part of the storage associated with one array is the same as part or all of the storage associated with another array.

If arrays with different data types are associated (or partially associated) with the same storage location, and the value of one array is defined (for example, by assignment), the value of the other array becomes undefined. This happens because an element of an array is considered defined only if the storage associated with it contains data of the same type as the array name.

An array element, array section, or whole array is defined by a DATA statement before program execution. (The array properties must be declared in a previous specification statement.) During program execution, array elements and sections are defined by an assignment or input statement, and entire arrays are defined by input statements.

See Also

- "Arrays"
- "DATA Statement"
- "Array Elements" for details on array element order

Deleted and Obsolescent Language Features

A

Fortran 90 identified some FORTRAN 77 features to be obsolescent. Fortran 95 deletes some of these features, and identifies a few more language features to be obsolescent. Features considered obsolescent may be removed from future revisions of the Fortran Standard.

You can specify a compiler option to have these features flagged.

NOTE. *Intel® Fortran fully supports features deleted from Fortran 95.*

This chapter contains information on the following topics:

- "Deleted Language Features in Fortran 95"
- "Obsolescent Language Features in Fortran 95"
- "Obsolescent Language Features in Fortran 90"

Deleted Language Features in Fortran 95

Some language features, considered redundant in FORTRAN 77, are not included in Fortran 95. However, they are still fully supported by Intel Fortran:

- ASSIGN and assigned GO TO statements
- Assigned FORMAT specifier
- Branching to an END IF statement from outside its IF block
- H edit descriptor
- PAUSE statement
- Real and double precision DO control variables and DO loop control expressions

For suggested methods to achieve the functionality of these features, see "Obsolescent Language Features in Fortran 90".

Obsolescent Language Features in Fortran 95

Some language features considered redundant in Fortran 90 are identified as obsolescent in Fortran 95.

Other methods are suggested to achieve the functionality of the following obsolescent features:

- Alternate returns

  To replace this functionality, it is recommended that you use an integer variable to return a value to the calling program, and let the calling program use a CASE construct to test the value and perform operations (see "CASE Constructs").
- Arithmetic IF

  To replace this functionality, it is recommended that you use an IF statement or construct (see "IF Construct and Statement").
- Assumed-length character functions

  To replace this functionality, it is recommended that you use one of the following:
  - An automatic character-length function, where the length of the function result is declared in a specification expression
  - A subroutine whose arguments correspond to the function result and the function arguments

  Dummy arguments of a function can still have assumed character length; this feature is not obsolescent.
- CHARACTER*(*) form of CHARACTER declaration

  To replace this functionality, it is recommended that you use the Fortran 90 forms of specifying a length selector in CHARACTER declarations (see "Declaration Statements for Character Types").
- Computed GO TO statement

  To replace this functionality, it is recommended that you use a CASE construct (see "CASE Constructs").
- DATA statements among executable statements

  This functionality has been included since FORTRAN 66, but is considered to be a potential source of errors.
- Fixed source form

  Newer methods of entering data have made this source form obsolescent and error-prone.

  The recommended method for coding is to use free source form (see "Free Source Form").
- Shared DO termination and termination on a statement other than END DO or CONTINUE

  To replace this functionality, it is recommended that you use an END DO statement (see "Forms for DO Constructs") or a CONTINUE statement (see "CONTINUE Statement").

- Statement functions

  To replace this functionality, it is recommended that you use an internal function (see "Internal Procedures").

Obsolescent Language Features in Fortran 90

Fortran 90 did not delete any of the features in FORTRAN 77, but some FORTRAN 77 features were identified as obsolescent.

Other methods are suggested to achieve the functionality of the following obsolescent features:

- Alternate return (labels in an argument list)

  To replace this functionality, it is recommended that you use an integer variable to return a value to the calling program, and let the calling program test the value and perform operations, using a computed GO TO statement (see "Computed GO TO Statement") or CASE construct (see "CASE Constructs").
- Arithmetic IF

  To replace this functionality, it is recommended that you use an IF statement or construct (see "IF Construct and Statement").
- ASSIGN and assigned GO TO statements

  These statements are usually used to simulate internal procedures (see "Internal Procedures"), which can now be coded directly.
- Assigned FORMAT specifier (label of a FORMAT statement assigned to an integer variable)

  To replace this functionality, it is recommended that you use character expressions to define format specifications (see "Format Specifications").
- Branching to an END IF statement from outside its IF block

  To replace this functionality, it is recommended that you branch to the statement following the END IF statement (see "IF Construct").
- H edit descriptor

  To replace this functionality, it is recommended that you use the character constant edit descriptor (see "Character String Edit Descriptors").
- PAUSE statement

  To replace this functionality, it is recommended that you use a READ statement that awaits input data (see "READ Statements").
- Real and double precision DO control variables and DO loop control expressions

  To replace this functionality, it is recommended that you use integer DO variables and expressions (see "DO Constructs").

- Shared DO termination and termination on a statement other than END DO or CONTINUE
  To replace this functionality, it is recommended that you use an END DO statement (see "Forms for DO Constructs") or a CONTINUE statement (see "CONTINUE Statement").

Additional Language Features

B

To facilitate compatibility with older versions of Fortran, Intel® Fortran provides the following additional language features:

- The "DEFINE FILE Statement"
- The "ENCODE and DECODE Statements"
- The "FIND Statement"
- The "INTERFACE TO Statement"
- "FORTRAN-66 Interpretation of the EXTERNAL Statement"
- "Alternative Syntax for the PARAMETER Statement"
- The "VIRTUAL Statement"
- "Alternative Syntax for Octal and Hexadecimal Constants"
- "Alternative Syntax for a Record Specifier"
- "Alternative Syntax for the DELETE Statement"
- "Alternative Form for Namelist External Records"
- The "Integer POINTER Statement"
- "Record Structures"

These language features are particularly useful in porting older Fortran programs to Fortran 95/90. However, you should avoid using them in new programs on these systems, and in new programs for which portability to other Fortran 95/90 implementations is important.

DEFINE FILE Statement

The DEFINE FILE statement establishes the size and structure of files with relative organization and associates them with a logical unit number. The DEFINE FILE statement is comparable to the OPEN statement. In situations where you can use the OPEN statement, OPEN is the preferable mechanism for creating and opening files.

The DEFINE FILE statement takes the following form:

DEFINE FILE *u*(*m*, *n*, *U*, *asv*) [, *u*(*m*, *n*, *U*, *asv*)] . . .

u

Is a scalar integer constant or variable that specifies the logical unit number.

m

Is a scalar integer constant or variable that specifies the number of records in the file.

n

Is a scalar integer constant or variable that specifies the length of each record in 16-bit words (2 bytes).

U

Specifies that the file is unformatted (binary); this is the only acceptable entry in this position.

asv

Is a scalar integer variable, called the associated variable of the file. At the end of each direct access I/O operation, the record number of the next higher numbered record in the file is assigned to *asv*. The *asv* must not be a dummy argument.

Rules and Behavior

The DEFINE FILE statement specifies that a file containing *m* fixed-length records, each composed of *n* 16-bit words, exists (or will exist) on the specified logical unit. The records in the file are numbered sequentially from 1 through *m*.

A DEFINE FILE statement does not itself open a file. However, the statement must be executed before the first direct access I/O statement referring to the specified file. The file is opened when the I/O statement is executed.

If this I/O statement is a WRITE statement, a direct access sequential file is opened, or created if necessary.

If the I/O statement is a READ or FIND statement, an existing file is opened, unless the specified file does not exist. If a file does not exist, an error occurs.

The DEFINE FILE statement establishes the variable *asv* as the associated variable of a file. At the end of each direct access I/O operation, the Fortran I/O system places in *asv* the record number of the record immediately following the one just read or written.

The associated variable always points to the next sequential record in the file (unless the associated variable is redefined by an assignment, input, or FIND statement). So, direct access I/O statements can perform sequential processing on the file by using the associated variable of the file as the record number specifier.

Example

In the following example, the DEFINE FILE statement specifies that the logical unit 3 is to be connected to a file of 1000 fixed-length records; each record is forty-eight 16-bit words long. The records are numbered sequentially from 1 through 1000 and are unformatted. After each direct access I/O operation on this file, the integer variable NREC will contain the record number of the record immediately following the record just processed.

```
DEFINE FILE 3(1000,48,U,NREC)
```

ENCODE and DECODE Statements

The ENCODE and DECODE statements translate data and transfer it between variables or arrays in internal storage. The ENCODE statement translates data from internal (binary) form to character form; the DECODE statement translates data from character to internal form. These statements are comparable to using internal files in formatted sequential WRITE and READ statements, respectively.

The ENCODE and DECODE statements take the following forms:

ENCODE (*c*, *f*, *b* [, IOSTAT=*i-var*] [, ERR=*label*]) [*io-list*]

DECODE (*c*, *f*, *b* [, IOSTAT=*i-var*] [, ERR=*label*]) [*io-list*]

c

Is a scalar integer expression. In the ENCODE statement, *c* is the number of characters (in bytes) to be translated to character form. In the DECODE statement, *c* is the number of characters to be translated to internal form.

f

Is a format identifier. An error occurs if more than one record is specified.

b

Is a scalar or array reference. If *b* is an array reference, its elements are processed in the order of subscript progression.

In the ENCODE statement, *b* receives the characters after translation to external form. If less than *c* characters are received, the remaining character positions are filled with blank characters. In the DECODE statement, *b* contains the characters to be translated to internal form.

i-var

Is a scalar integer variable that is defined as a positive integer if an error occurs and as zero if no error occurs (see "I/O Status Specifier").

label

Is the label of an executable statement that receives control if an error occurs.

io-list

Is an I/O list (see "I/O Lists").

In the ENCODE statement, the list contains the data to be translated to character form. In the DECODE statement, the list receives the data after translation to internal form.

The interaction between the format specifier and the I/O list is the same as for a formatted I/O statement.

Rules and Behavior

The number of characters that the ENCODE or DECODE statement can translate depends on the data type of *b*. For example, an INTEGER (2) array can contain two characters per element, so that the maximum number of characters is twice the number of elements in that array.

The maximum number of characters a character variable or character array element can contain is the length of the character variable or character array element.

The maximum number of characters a character array can contain is the length of each element multiplied by the number of elements.

Examples

In the following example, the DECODE statement translates the 12 characters in A to integer form (as specified by the FORMAT statement):

```
      DIMENSION K(3)
      CHARACTER*12 A,B
      DATA A/'123456789012'/
      DECODE(12,100,A) K
100   FORMAT(3I4)
      ENCODE(12,100,B) K(3), K(2), K(1)
```

The 12 characters are stored in array K:

```
K(1) = 1234
K(2) = 5678
K(3) = 9012
```

The ENCODE statement translates the values K(3), K(2), and K(1) to character form and stores the characters in the character variable B:

```
B = '901256781234'
```

See Also

- "Forms and Rules for Internal READ Statements"
- "Forms and Rules for Internal WRITE Statements"

FIND Statement

The FIND statement positions a direct access file at a particular record and sets the associated variable of the file to that record number. It is comparable to a direct access READ statement with no I/O list, and it can open an existing file. No data transfer takes place.

The FIND statement takes one of the following forms:

FIND ([UNIT=]*io-unit*, REC=*r* [, ERR=*label*] [, IOSTAT=*i-var*])

FIND (*io-unit* '*r* [, ERR=*label*] [, IOSTAT=*i-var*])

io-unit

Is a logical unit number. It must refer to a relative organization file (see "Unit Specifier").

r

Is the direct access record number. It cannot be less than one or greater than the number of records defined for the file (see "Record Specifier").

label

Is the label of the executable statement that receives control if an error occurs.

i-var

Is a scalar integer variable that is defined as a positive integer if an error occurs, and as zero if no error occurs (see "I/O Status Specifier").

Examples

In the following example, the FIND statement positions logical unit 1 at the first record in the file. The file's associated variable is set to one:

```
FIND(1, REC=1)
```

In the following example, the FIND statement positions the file at the record identified by the content of INDX. The file's associated variable is set to the value of INDX:

```
FIND(4, REC=INDX)
```

See Also

"Forms for Direct-Access READ Statements"

INTERFACE TO Statement

The INTERFACE TO statement identifies a subprogram and its actual arguments before it is referenced or called.

The INTERFACE TO statement takes the following form:

INTERFACE TO *subprogram-stmt*
[*formal-declarations*]
END

subprogram-stmt

Is a function or subroutine declaration statement.

formal-declarations

(Optional) Are type declaration statements (including optional attributes) for the arguments.

Rules and Behavior

The INTERFACE TO block defines an explicit interface, but it contains specifications for only the procedure declared in the INTERFACE TO statement. The explicit interface is defined only in the program unit that contains the INTERFACE TO block.

The recommended method for defining explicit interfaces is to use an INTERFACE block.

Example

Consider that a C function that has the following prototype:

```
extern void Foo (int i);
```

The following INTERFACE TO block declares the Fortran call to this function:

```
INTERFACE TO SUBROUTINE Foo [C.ALIAS: '_Foo'] (I)
  INTEGER*4 I
END
```

See Also

"Defining Explicit Interfaces" for details on INTERFACE blocks

FORTRAN-66 Interpretation of the EXTERNAL Statement

If you specify the compiler option indicating FORTRAN-66 semantics, the EXTERNAL statement is interpreted in a way that facilitates compatibility with older versions of Fortran. (The Fortran 95/90 interpretation is incompatible with previous Fortran standards and previous Compaq* implementations.)

The FORTRAN-66 interpretation of the EXTERNAL statement combines the functionality of the INTRINSIC statement with that of the EXTERNAL statement.

This lets you use subprograms as arguments to other subprograms. The subprograms to be used as arguments can be either user-supplied functions or Fortran 95/90 library functions.

The FORTRAN-66 EXTERNAL statement takes the following form:

EXTERNAL [*]*v* [, [*]*v*]...

*

Specifies that a user-supplied function is to be used instead of a Fortran 95/90 library function having the same name.

v

Is the name of a subprogram or the name of a dummy argument associated with the name of a subprogram.

Rules and Behavior

The FORTRAN-66 EXTERNAL statement declares that each name in its list is an external function name. Such a name can then be used as an actual argument to a subprogram, which then can use the corresponding dummy argument in a function reference or CALL statement.

However, when used as an argument, a complete function reference represents a value, not a subprogram name; for example, SQRT(B) in CALL SUBR(A, SQRT(B), C). It is not, therefore, defined in an EXTERNAL statement (as would be the incomplete reference SQRT).

Example

Example B-1 shows the FORTRAN-66 EXTERNAL statement:

Example B-1 Using the F66 External Statement

```
Main Program                          Subprograms
EXTERNAL SIN, COS, *TAN, SINDEG        SUBROUTINE TRIG(X,F,Y)
   .                                   Y = F(X)
   .                                   RETURN
   .                                   END
CALL TRIG(ANGLE, SIN, SINE)
   .
   .                                   FUNCTION TAN(X)
   .                                   TAN = SIN(X)/COS(X)
CALL TRIG(ANGLE, COS, COSINE)          RETURN
   .                                   END
```

Example B-1 Using the F66 External Statement

```
      .
      .
   CALL TRIG(ANGLE, TAN, TANGNT)       FUNCTION SINDEG(X)
      .                                SINDEG = SIN(X*3.1459/180)
      .                                RETURN
      .                                END
   CALL TRIG(ANGLED, SINDEG, SINE)
```

The CALL statements pass the name of a function to the subroutine TRIG. The function reference F(X) subsequently invokes the function in the second statement of TRIG. Depending on which CALL statement invoked TRIG, the second statement is equivalent to one of the following:

```
Y = SIN(X)
Y = COS(X)
Y = TAN(X)
Y = SINDEG(X)
```

The functions SIN and COS are examples of trigonometric functions supplied in the Fortran 95/90 library. The function TAN is also supplied in the library, but the asterisk (*) in the EXTERNAL statement specifies that the user-supplied function be used, instead of the library function. The function SINDEG is also a user-supplied function. Because no library function has the same name, no asterisk is required.

See Also

Chapter 9, “Intrinsic Procedures”

Alternative Syntax for the PARAMETER Statement

The PARAMETER statement discussed here is similar to the one discussed in “PARAMETER Attribute and Statement”; they both assign a name to a constant. However, this PARAMETER statement differs from the other one in the following ways:

- Its list is not bounded with parentheses.
- The form of the constant, rather than implicit or explicit typing of the name, determines the data type of the variable.

This PARAMETER statement takes the following form:

PARAMETER *c* = *expr* [, *c* = *expr*]...

c

Is the name of the constant.

expr

Is an initialization expression. It can be of any data type.

Rules and Behavior

Each name *c* becomes a constant and is defined as the value of expression *expr*. Once a name is defined as a constant, it can appear in any position in which a constant is allowed. The effect is the same as if the constant were written there instead of the name.

The name of a constant cannot appear as part of another constant, except as the real or imaginary part of a complex constant. For example:

```
PARAMETER I=3
PARAMETER M=I.25            ! Not allowed
PARAMETER N=(1.703, I)      ! Allowed
```

The name used in the PARAMETER statement identifies only the name's corresponding constant in that program unit. Such a name can be defined only once in PARAMETER statements within the same program unit.

The name of a constant assumes the data type of its corresponding constant expression. The data type of a parameter constant cannot be specified in a type declaration statement. Nor does the initial letter of the constant's name implicitly affect its data type.

Examples

The following are valid examples of this form of the PARAMETER statement:

```
PARAMETER PI=3.1415927, DPI=3.141592653589793238D0
PARAMETER PIOV2=PI/2, DPIOV2=DPI/2
PARAMETER FLAG=.TRUE., LONGNAME='A STRING OF 25 CHARACTERS'
```

See Also

"PARAMETER Attribute and Statement" for details on compile-time constant expressions

VIRTUAL Statement

The VIRTUAL statement is included for compatibility with PDP-11 Fortran. It has the same form and effect as the DIMENSION statement (see "DIMENSION Attribute and Statement").

Alternative Syntax for Octal and Hexadecimal Constants

To facilitate compatibility, you can use an alternative syntax for octal and hexadecimal constants. The following table shows this alternative syntax and equivalents:

Constant	Alternative Syntax	Equivalent
Octal	'0...7'O	O'0..7'
Hexadecimal	'0..F'X	Z'0..F'

You can use a quotation mark (") in place of an apostrophe in all the above syntax forms.

For information on the # syntax for integers not in base 10, see "Integer Data Types".

See Also

- "Octal Constants"
- "Hexadecimal Constants"

Alternative Syntax for a Record Specifier

To facilitate compatibility, you can specify the following form for a record specifier in an I/O control list:

'*r*

r

Is a numeric expression with a value that represents the position of the record to be accessed using direct access I/O.

The value must be greater than or equal to 1, and less than or equal to the maximum number of records allowed in the file. If necessary, a record number is converted to integer data type before being used.

If this nonkeyword form is used in an I/O control list, it must immediately follow the nonkeyword form of the io-unit specifier.

Alternative Syntax for the DELETE Statement

To facilitate compatibility, you can specify the following form of the DELETE statement when deleting records from a relative file:

DELETE (*io-unit*'*r* [, ERR=*label*] [, IOSTAT=*i-var*])

io-unit

Is the number of the logical unit containing the record to be deleted.

r

Is the positional number of the record to be deleted.

label

Is the label of an executable statement that receives control if an error condition occurs.

i-var

Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs.

This form deletes the direct access record specified by *r*.

See Also

"DELETE Statement"

Alternative Form for Namelist External Records

To facilitate compatibility, you can use the following form for an external record:

$*group-name object* = *value* [*object* = *value*]...$[END]

group-name

Is the name of the group containing the objects to be given values. The name must have been previously defined in a NAMELIST statement in the scoping unit.

object

Is the name (or subobject designator) of an entity defined in the NAMELIST declaration of the group name. The object name must not contain embedded blanks, but it can be preceded or followed by blanks.

value

Is a null value, a constant (or list of constants), a repetition of constants in the form r*c, or a repetition of null values in the form r*.

If more than one *object=value* or more than one value is specified, they must be separated by value separators.

A value separator is any number of blanks, or a comma or slash, preceded or followed by any number of blanks.

See Also

- "Rules for Namelist Sequential READ Statements" for details on namelist input
- "Rules for Namelist Sequential WRITE Statements" for details on namelist output

Integer POINTER Statement

The POINTER statement discussed here is different from the one discussed in "POINTER Attribute and Statement". It establishes pairs of variables and pointers, in which each pointer contains the address of its paired variable.

This POINTER statement takes the following form:

POINTER (*pointer*, *pointee*) [, (*pointer*, *pointee*)] . . .

pointer

Is a variable whose value is used as the address of the pointee.

pointee

Is a variable; it can be an array name or array specification.

Rules and Behavior

The following are *pointer* rules and behavior:

- Two pointers can have the same value, so pointer aliasing is allowed.
- When used directly, a pointer is treated like an integer variable. On Intel® Itanium® processors, a pointer occupies two numeric storage units, so it is a 64-bit quantity (INTEGER(8)). On IA-32 processors, a pointer occupies one numeric storage unit, so it is a 32-bit quantity (INTEGER(4)).
- A pointer cannot be a pointee.
- A pointer cannot appear in an ASSIGN statement and cannot have the following attributes:

ALLOCATABLE	INTRINSIC	POINTER
EXTERNAL	PARAMETER	TARGET

  A pointer can appear in a DATA statement with integer literals only.
- Integers can be converted to pointers, so you can point to absolute memory locations.
- A pointer variable cannot be declared to have any other data type.
- A pointer cannot be a function return value.
- You can give values to pointers by doing the following:
  — Retrieve addresses by using the LOC intrinsic function (or the %LOC built-in function)

— Allocate storage for an object by using the MALLOC intrinsic function (or by using `malloc(3f)` on Linux* systems)

For example:

Using %LOC:	**Using MALLOC:**
`INTEGER I(10)`	`INTEGER I(10)`
`INTEGER I1(10) /10*10/`	`POINTER (P,I)`
`POINTER (P,I)`	`P = MALLOC(40)`
`P = %LOC(I1)`	`I = 10`
`I(2) = I(2) + 1`	`I(2) = I(2) + 1`

- The value in a pointer is used as the pointee's base address.

The following are *pointee* rules and behavior:

- A pointee is not allocated any storage. References to a pointee look to the current contents of its associated pointer to find the pointee's base address.
- A pointee cannot be data-initialized or have a record structure that contains data-initialized fields.
- A pointee can appear in only one integer POINTER statement.
- A pointee array can have fixed, adjustable, or assumed dimensions.
- A pointee cannot appear in a COMMON, DATA, EQUIVALENCE, or NAMELIST statement, and it cannot have the following attributes:

ALLOCATABLE	OPTIONAL	SAVE
AUTOMATIC	PARAMETER	STATIC
INTENT	POINTER	

- A pointee cannot be:
  — A dummy argument
  — A function return value
  — A record field or an array element
  — Zero-sized
  — An automatic object
  — The name of a generic interface block
- If a pointee is of derived type, it must be of sequence type.

Record Structures

Intel Fortran record structures are similar to Fortran 95/90 derived types.

A *record structure* is an aggregate entity containing one or more elements. (Record elements are also called fields or components.) You can use records when you need to declare and operate on multi-field data structures in your programs.

Creating a record is a two-step process:

1. You must define the form of the record with a multistatement structure declaration.
2. You must use a RECORD statement to declare the record as an entity with a name. (More than one RECORD statement can refer to a given structure.)

The following sections discuss:

- "Structure Declarations"
- "RECORD Statement"
- "References to Record Fields"
- "Aggregate Assignment"

See Also

"Derived Data Types"

Structure Declarations

A *structure declaration* defines the field names, types of data within fields, and order and alignment of fields within a record. Fields and structures can be initialized, but records cannot be initialized.

A structure declaration takes the following form:

```
STRUCTURE [/structure-name/][field-namelist]
    field-declaration
    [field-declaration]
    . . .
    [field-declaration]
END STRUCTURE
```

structure-name

Is the name used to identify a structure, enclosed by slashes.

Subsequent RECORD statements use the structure name to refer to the structure. A structure name must be unique among structure names, but structures can share names with variables (scalar or array), record fields, PARAMETER constants, and common blocks.

Structure declarations can be nested (contain one or more other structure declarations). A structure name is required for the structured declaration at the outermost level of nesting, and is optional for the other declarations nested in it. However, if you wish to reference a nested structure in a RECORD statement in your program, it must have a name.

Structure, field, and record names are all local to the defining program unit. When records are passed as arguments, the fields in the defining structures within the calling and called subprograms must match in type, order, and dimension.

field-namelist

Is a list of fields having the structure of the associated structure declaration. A field namelist is allowed only in nested structure declarations.

field-declaration

Also called the declaration body. A *field-declaration* consists of any combination of the following:

- "Type Declarations"

  These are ordinary Fortran data type declarations.
- "Substructure Declarations"

  A field within a structure can be a substructure composed of atomic fields, other substructures, or a combination of both.
- "Union Declarations"

  A union declaration is composed of one or more mapped field declarations.
- PARAMETER statements

  PARAMETER statements can appear in a structure declaration, but cannot be given a data type within the declaration block.

  Type declarations for PARAMETER names must precede the PARAMETER statement and be outside of a STRUCTURE declaration, as follows:

```
   INTEGER*4 P
STRUCTURE /ABC/
   PARAMETER (P=4)
   REAL*4 F
END STRUCTURE
   REAL*4 A(P)
```

Rules and Behavior

Unlike type declaration statements, structure declarations do not create variables. Structured variables (records) are created when you use a RECORD statement containing the name of a previously declared structure. The RECORD statement can be considered as a kind of type declaration statement. The difference is that aggregate items, not single items, are being defined.

Within a structure declaration, the ordering of both the statements and the field names within the statements is important, because this ordering determines the order of the fields in records.

In a structure declaration, each field offset is the sum of the lengths of the previous fields, so the length of the structure is the sum of the lengths of its fields. The structure is packed; you must explicitly provide any alignment that is needed by including, for example, unnamed fields of the appropriate length.

By default, fields are aligned on natural boundaries; misaligned fields are padded as necessary. To avoid padding of records, you should lay out structures so that all fields are naturally aligned.

To pack fields on arbitrary byte boundaries, you must specify a compiler option. You can also specify alignment for fields by using the cDEC$ OPTIONS or cDEC$ PACK general directive.

A field name must not be the same as any intrinsic or user-defined operator (for example, EQ cannot be used as a field name).

Example

In the following example, the declaration defines a structure named APPOINTMENT. APPOINTMENT contains the structure DATE (field APP_DATE) as a substructure. It also contains a substructure named TIME (field APP_TIME, an array), a CHARACTER*20 array named APP_ MEMO, and a LOGICAL*1 field named APP_FLAG.

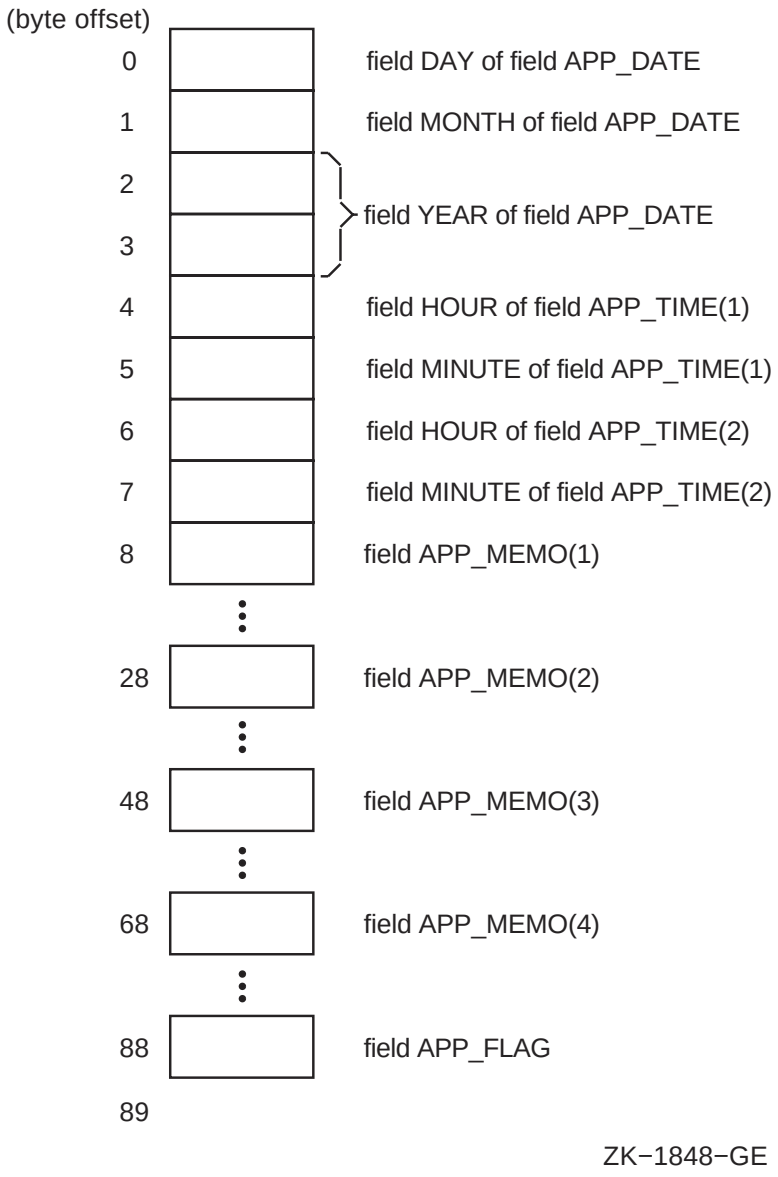
```
STRUCTURE /DATE/
   INTEGER*1 DAY, MONTH
  INTEGER*2 YEAR
END STRUCTURE

STRUCTURE /APPOINTMENT/
   RECORD /DATE/     APP_DATE
   STRUCTURE /TIME/  APP_TIME (2)
     INTEGER*1       HOUR, MINUTE
   END STRUCTURE
   CHARACTER*20      APP_MEMO (4)
   LOGICAL*1         APP_FLAG
END STRUCTURE
```

The length of any instance of structure APPOINTMENT is 89 bytes.

Figure B-1 shows the memory mapping of any record or record array element with the structure APPOINTMENT.

Figure B-1 Memory Map of Structure APPOINTMENT



See Also

- "OPTIONS Directive"
- "PACK Directive"
- Your user's guide for details on compiler options

Type Declarations

The syntax of a type declaration within a record structure is identical to that of a normal Fortran type statement.

The following rules and behavior apply to type declarations in record structures:

- %FILL can be specified in place of a field name to leave space in a record for purposes such as alignment. This creates an unnamed field.

  %FILL can have an array specification; for example:

  ```
  INTEGER %FILL (2,2)
  ```

  Unnamed fields cannot be initialized. For example, the following statement is invalid and generates an error message:

  ```
  INTEGER %FILL /1980/
  ```
- Initial values can be supplied in field declaration statements. Unnamed fields cannot be initialized; they are always undefined.
- Field names must always be given explicit data types. The IMPLICIT statement does not affect field declarations.
- Any required array dimensions must be specified in the field declaration statements. DIMENSION statements cannot be used to define field names.
- Adjustable or assumed sized arrays and assumed-length CHARACTER declarations are not allowed in field declarations.

Substructure Declarations

A field within a structure can itself be a structured item composed of other fields, other structures, or both. You can declare a substructure in two ways:

- By nesting structure declarations within other structure or union declarations (with the limitation that you cannot refer to a structure inside itself at any level of nesting).

  One or more field names must be defined in the STRUCTURE statement for the substructure, because all fields in a structure must be named. In this case, the substructure is being used as a field within a structure or union.

  Field names within the same declaration nesting level must be unique, but an inner structure declaration can include field names used in an outer structure declaration without conflict.
- By using a RECORD statement that specifies another previously defined record structure, thereby including it in the structure being declared.

See the example in the "Structure Declarations", for a sample structure declaration containing both a nested structure declaration (TIME) and an included structure (DATE).

Union Declarations

A union declaration is a multistatement declaration defining a data area that can be shared intermittently during program execution by one or more fields or groups of fields. A union declaration must be within a structure declaration.

Each unique field or group of fields is defined by a separate map declaration.

A union declaration takes the following form:

UNION
  map-declaration
  map-declaration
  [*map-declaration*]
  . . .
  [*map-declaration*]
END UNION

map-declaration

Takes the following form:

MAP
  field-declaration
  [*field-declaration*]
  . . .
  [*field-declaration*]
END MAP

field-declaration

Is a structure declaration or RECORD statement contained within a union declaration, a union declaration contained within a union declaration, or the declaration of a data field (having a data type) within a union. For a more detailed description of what can be specified in field declarations, see "Structure Declarations".

Rules and Behavior

As with normal Fortran type declarations, data can be initialized in field declaration statements in union declarations. However, if fields within multiple map declarations in a single union are initialized, the data declarations are initialized in the order in which the statements appear. As a result, only the final initialization takes effect and all of the preceding initializations are overwritten.

The size of the shared area established for a union declaration is the size of the largest map defined for that union. The size of a map is the sum of the sizes of the fields declared within it.

Manipulating data by using union declarations is similar to using EQUIVALENCE statements. The difference is that data entities specified within EQUIVALENCE statements are concurrently associated with a common storage location and the data residing there; with union declarations you can use one discrete storage location to alternately contain a variety of fields (arrays or variables).

With union declarations, only one map declaration within a union declaration can be associated at any point in time with the storage location that they share. Whenever a field within another map declaration in the same union declaration is referenced in your program, the fields in the prior map declaration become undefined and are succeeded by the fields in the map declaration containing the newly referenced field.

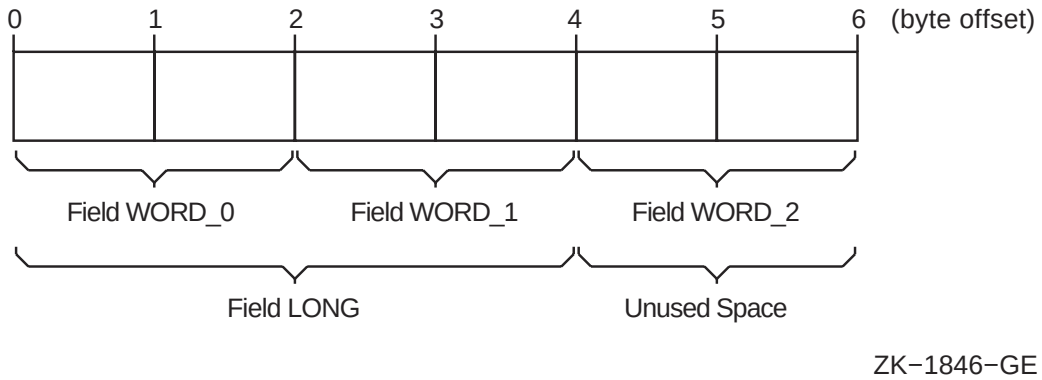
Example

In the following example, the structure WORDS_LONG is defined. This structure contains a union declaration defining two map fields. The first map field consists of three INTEGER*2 variables (WORD_0, WORD_1, and WORD_2), and the second, an INTEGER*4 variable, LONG:

```
STRUCTURE /WORDS_LONG/
   UNION
     MAP
       INTEGER*2  WORD_0, WORD_1, WORD_2
     END MAP
     MAP
       INTEGER*4  LONG
     END MAP
   END UNION
END STRUCTURE
```

The length of any record with the structure WORDS_LONG is 6 bytes. Figure B-2 shows the memory mapping of any record with the structure WORDS_LONG:

Figure B-2 Memory Map of Structure WORDS_LONG



RECORD Statement

A RECORD statement takes the following form:

RECORD /*structure-name*/*record-namelist*
[, /*structure-name*/*record-namelist*]
. . .
[, /*structure-name*/*record-namelist*]

structure-name

Is the name of a previously declared structure.

record-namelist

Is a list of one or more variable names, array names, or array specifications, separated by commas. All of the records named in this list have the same structure and are allocated separately in memory.

Rules and Behavior

You can use record names in COMMON and DIMENSION statements, but not in DATA, EQUIVALENCE, or NAMELIST statements.

Records initially have undefined values unless you have defined their values in structure declarations.

References to Record Fields

References to record fields must correspond to the kind of field being referenced. Aggregate field references refer to composite structures (and substructures). Scalar field references refer to singular data items, such as variables.

An operation on a record can involve one or more fields.

Record field references take one of the following forms:

Aggregate Field Reference:

record-name [.*aggregate-field-name*] . . .

Scalar Field Reference:

record-name [.*aggregate-field-name*] . . . .*scalar-field-name*

record-name

Is the name used in a RECORD statement to identify a record.

aggregate-field-name

Is the name of a field that is a substructure (a record or a nested structure declaration) within the record structure identified by the record name.

scalar-field-name

Is the name of a data item (having a data type) defined within a structure declaration.

Rules and Behavior

Records and record fields cannot be used in EQUIVALENCE statements. However, you can make fields of record structures equivalent to themselves by using the UNION and MAP statements in a structure declaration.

Records and record fields cannot be used in DATA statements, but individual fields can be initialized in the STRUCTURE definition.

An automatic array cannot be a record field.

A scalar field reference consists of the name of a record (as specified in a RECORD statement) and zero or more levels of aggregate field names followed by the name of a scalar field. A scalar field reference refers to a single data item (having a data type) and can be treated like a normal reference to a Fortran variable or array element.

You can use scalar field references in statement functions and in executable statements. However, they cannot be used in COMMON, SAVE, NAMELIST, or EQUIVALENCE statements, or as the control variable in an indexed DO-loop.

Type conversion rules for scalar field references are the same as those for variables and array elements.

An aggregate field reference consists of the name of a record (as specified in a RECORD statement) and zero or more levels of aggregate field names.

You can only assign an aggregate field to another aggregate field (record = record) if the records have the same structure. Intel Fortran supports no other operations (such as arithmetic or comparison) on aggregate fields.

Intel Fortran requires qualification on all levels. While some languages allow omission of aggregate field names when there is no ambiguity as to which field is intended, Intel Fortran requires all aggregate field names to be included in references.

You can use aggregate field references in unformatted I/O statements; one I/O record is written no matter how many aggregate and array name references appear in the I/O list. You cannot use aggregate field references in formatted, namelist, and list-directed I/O statements.

You can use aggregate field references as actual arguments and record dummy arguments. The declaration of the dummy record in the subprogram must match the form of the aggregate field reference passed by the calling program unit; each structure must have the same number and types of fields in the same order. The order of map fields within a union declaration is irrelevant.

Records are passed by reference. Aggregate field references are treated like normal variables. You can use adjustable arrays in RECORD statements that are used as dummy arguments.

NOTE. *Because periods are used in record references to separate fields, you should not use relational operators (*.EQ., .XOR.*), logical constants (*.TRUE., .FALSE.*), and logical expressions (*.AND., .NOT., .OR.*) as field names in structure declarations.*

Examples

The following examples show record and field references. Consider the following structure declarations:

Structure DATE:

```
STRUCTURE /DATE/
  INTEGER*1 DAY, MONTH
  INTEGER*2 YEAR
STRUCTURE
```

Structure APPOINTMENT:

```
STRUCTURE /APPOINTMENT/
  RECORD /DATE/      APP_DATE
```

```
  STRUCTURE /TIME/   APP_TIME(2)
    INTEGER*1        HOUR, MINUTE
  END STRUCTURE
  CHARACTER*20       APP_MEMO(4)
  LOGICAL*1          APP_FLAG
END STRUCTURE
```

The following RECORD statement creates a variable named NEXT_APP and a 10-element array named APP_LIST. Both the variable and each element of the array take the form of the structure APPOINTMENT.

```
RECORD /APPOINTMENT/    NEXT_APP,APP_LIST(10)
```

Each of the following examples of record and field references are derived from the previous structure declarations and RECORD statement:

Aggregate Field References

- The record NEXT_APP:
  `NEXT_APP`
- The field APP_DATE, a 4-byte array field in the record array APP_LIST(3):
  `APP_LIST(3).APP_DATE`

Scalar Field References

- The field APP_FLAG, a LOGICAL field of the record NEXT_APP:
  `NEXT_APP.APP_FLAG`
- The first character of APP_MEMO(1), a CHARACTER*20 field of the record NEXT_APP:
  `NEXT_APP.APP_MEMO(1)(1:1)`

See Also

- "RECORD Statement"
- "Structure Declarations", which also contains details on specification of fields within structure declarations
- "Union Declarations" for details on UNION and MAP statements
- Your user's guide for details on alignment of data

Aggregate Assignment

For aggregate assignment statements, the variable and expression must have the same structure as the aggregate they reference.

The aggregate assignment statement assigns the value of each field of the aggregate on the right of an equal sign to the corresponding field of the aggregate on the left. Both aggregates must be declared with the same structure.

Example

The following example shows valid aggregate assignments:

```
STRUCTURE /DATE/
  INTEGER*1 DAY, MONTH
  INTEGER*2 YEAR
END STRUCTURE

RECORD /DATE/ TODAY, THIS_WEEK(7)
STRUCTURE /APPOINTMENT/
  ...
  RECORD /DATE/ APP_DATE  END STRUCTURE

RECORD /APPOINTMENT/ MEETING

DO I =  1,7
  CALL GET_DATE (TODAY)
  THIS_WEEK(I) =     TODAY
  THIS_WEEK(I).DAY = TODAY.DAY + 1
END DO
MEETING.APP_DATE =   TODAY
```

The ASCII Character Set for Linux Systems

C

This appendix describes the ASCII character set that is available on Linux* systems. Other character sets are available on Windows* systems; for details, see the online documentation for those systems.

For details on the Fortran 95/90 character set, see "Character Sets".

The ASCII Character Set (L*X)

Figure C-1 represents the ASCII character set (characters with decimal values 0 through 127). The first half of each of the numbered columns identifies the character as you would enter it on a terminal or as you would see it on a printer. Except for SP and HT, the characters with names are nonprintable. In Figure C-1, the characters with names are defined as follows:

NUL	Null	DC1	Device Control 1 (XON)
SOH	Start of Heading	DC2	Device Control 2
STX	Start of Text	DC3	Device Control 1 (XOFF)
ETX	End of Text	DC4	Device Control 4
EOT	End of Transmission	NAK	Negative Acknowledge
ENQ	Enquiry	SYN	Synchronous Idle
ACK	Acknowledge	ETB	End of Transmission Block
BEL	Bell	CAN	Cancel
BS	Backspace	EM	End of Medium
HT	Horizontal Tab	SUB	Substitute
LF	Line Feed	ESC	Escape
VT	Vertical Tab	FS	File Separator
FF	Form Feed	GS	Group Separator
CR	Carriage Return	RS	Record Separator

SO	Shift Out	US	Unit Separator
SI	Shift In	SP	Space
DLE	Data Link Escape	DEL	Delete

The remaining half of each column identifies the character by the binary value of the byte; the value is stated in three radixes—octal, decimal, and hexadecimal. For example, the uppercase letter A has, under ASCII conventions, a storage value of hexadecimal 41 (a bit configuration of 01000001), equivalent to 101 in octal notation and 65 in decimal notation.

Figure C-1 Graphic Representation of the ASCII Character Set (L*X)

Row	Column / Bits (b8 b7 b6 b5 / b4 b3 b2 b1)	0		1		2		3		4		5		6		7	
		0 0 0 0		0 0 0 1		0 0 1 0		0 0 1 1		0 1 0 0		0 1 0 1		0 1 1 0		0 1 1 1	
0	0 0 0 0	NUL	0 0 0	DLE	20 16 10	SP	40 32 20	0	60 48 30	@	100 64 40	P	120 80 50	`	140 96 60	p	160 112 70
1	0 0 0 1	SOH	1 1 1	DC1 (XON)	21 17 11	!	41 33 21	1	61 49 31	A	101 65 41	Q	121 81 51	a	141 97 61	q	161 113 71
2	0 0 1 0	STX	2 2 2	DC2	22 18 12	"	42 34 22	2	62 50 32	B	102 66 42	R	122 82 52	b	142 98 62	r	162 114 72
3	0 0 1 1	ETX	3 3 3	DC3 (XOFF)	23 19 13	#	43 35 23	3	63 51 33	C	103 67 43	S	123 83 53	c	143 99 63	s	163 115 73
4	0 1 0 0	EOT	4 4 4	DC4	24 20 14	$	44 36 24	4	64 52 34	D	104 68 44	T	124 84 54	d	144 100 64	t	164 116 74
5	0 1 0 1	ENQ	5 5 5	NAK	25 21 15	%	45 37 25	5	65 53 35	E	105 69 45	U	125 85 55	e	145 101 65	u	165 117 75
6	0 1 1 0	ACK	6 6 6	SYN	26 22 16	&	46 38 26	6	66 54 36	F	106 70 46	V	126 86 56	f	146 102 66	v	166 118 76
7	0 1 1 1	BEL	7 7 7	ETB	27 23 17	´	47 39 27	7	67 55 37	G	107 71 47	W	127 87 57	g	147 103 67	w	167 119 77
8	1 0 0 0	BS	10 8 8	CAN	30 24 18	(	50 40 28	8	70 56 38	H	110 72 48	X	130 88 58	h	150 104 68	x	170 120 78
9	1 0 0 1	HT	11 9 9	EM	31 25 19	)	51 41 29	9	71 57 39	I	111 73 49	Y	131 89 59	i	151 105 69	y	171 121 79
10	1 0 1 0	LF	12 10 A	SUB	32 26 1A	*	52 42 2A	:	72 58 3A	J	112 74 4A	Z	132 90 5A	j	152 106 6A	z	172 122 7A
11	1 0 1 1	VT	13 11 B	ESC	33 27 1B	+	53 43 2B	;	73 59 3B	K	113 75 4B	[	133 91 5B	k	153 107 6B	{	173 123 7B
12	1 1 0 0	FF	14 12 C	FS	34 28 1C	,	54 44 2C	<	74 60 3C	L	114 76 4C	\	134 92 5C	l	154 108 6C	|	174 124 7C
13	1 1 0 1	CR	15 13 D	GS	35 29 1D	−	56 45 2D	=	75 61 3D	M	115 77 4D	]	135 93 5D	m	155 109 6D	}	175 125 7D
14	1 1 1 0	SO	16 14 E	RS	36 30 1E	.	56 46 2E	>	76 62 3E	N	116 78 4E	^	136 94 5E	n	156 110 6E	~	176 126 7E
15	1 1 1 1	SI	17 15 F	US	37 31 1F	/	57 47 2F	?	77 63 3F	O	117 79 4F	_	137 95 5F	o	157 111 6F	DEL	177 127 7F

Key

Character		
ESC	33	Octal
	27	Decimal
	1B	Hex

ZK−1752−GE

Data Representation Models

D

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).

The following intrinsic functions provide information on the data representation models:

Intrinsic function	Model	Value returned
BIT_SIZE	Bit	The number of bits (*s*) in the bit model
DIGITS	Integer or Real	The number of significant digits in the model for the argument
EPSILON	Real	The number that is almost negligible when compared to one
EXPONENT	Real	The value of the exponent part of a real argument
FRACTION	Real	The fractional part of a real argument
HUGE	Integer or Real	The largest number in the model for the argument
MAXEXPONENT	Real	The maximum exponent in the model for the argument
MINEXPONENT	Real	The minimum exponent in the model for the argument
NEAREST	Real	The nearest different machine-representable number in a given direction
PRECISION	Real	The decimal precision (real or complex) of the argument
RADIX	Integer or Real	The base of the model for the argument
RANGE	Integer or Real	The decimal exponent range of the model for the argument
RRSPACING	Real	The reciprocal of the relative spacing near the argument
SCALE	Real	The value of the exponent part (of the model for the argument) changed by a specified value
SET_EXPONENT	Real	The value of the exponent part (of the model for the argument) set to a specified value

Intrinsic function	Model	Value returned
SPACING	Real	The value of the absolute spacing of model numbers near the argument
TINY	Real	The smallest positive number in the model for the argument

For more information on the range of values for each data type (and kind), see your user's guide.

This appendix discusses the following topics:

The "Model for Integer Data"

The "Model for Real Data"

The "Model for Bit Data"

Model for Integer Data

In general, the model set for integers is defined as follows:

$$i = s \times \sum_{k=1}^{q} w_k \times r^{k-1}$$

The following values apply to this model set:

- i is the integer value.
- s is the sign (either +1 or –1).
- q is the number of digits (a positive integer).
- r is the radix (an integer greater than 1).
- w_k is a nonnegative number less than r.

The model for INTEGER(4) is as follows:

$$i = s \times \sum_{k=1}^{31} w_k \times 2^{k-1}$$

The following example shows the general integer model for $i = -20$ using a base (r) of 2:

$$i = (-1) \times (0 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 0 \times 2^3 + 1 \times 2^4)$$

$$i = (-1) \times (4 + 16)$$

$$i = -1 \times 20$$

$$i = -20$$

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; b = 2 in Intel® Fortran).
- p is the number of mantissa digits (an integer greater than 1). The number of digits differs depending on the real format, as follows:

REAL(4)	IEEE S_floating	24
REAL(8)	IEEE T_floating	53
REAL(16)	IEEE X_floating	113

- 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}
REAL(4)	IEEE S_floating	–125	128
REAL(8)	IEEE T_floating	–1021	1024
REAL(16)	IEEE X_floating	–16381	16384

- f_k is a nonnegative number less than b (f_1 is also nonzero).

For $x = 0$, its exponent e and digits f_k are defined to be zero.

The model set for single-precision real (REAL(4)) is defined as one of the following:

$$x = 0$$

$$x = s \times 2^{e} \times \left[1/2 + \sum_{k=2}^{24} f_k \times 2^{-k} \right], -125 \le e \le 128$$

The following example shows the general real model for $x = 20.0$ using a base (b) of 2:

$$x = 1 \times 2^{5} \times (1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3})$$

$$x = 1 \times 32 \times (.5 + .125)$$

$$x = 32 \times (.625)$$

$$x = 20.0$$

Model for Bit Data

The model set for bits (binary digits) interprets a nonnegative scalar data object of type integer as a sequence, as follows:

$$j = \sum_{k=0}^{s-1} w_k \times 2^{k}$$

The following values apply to this model set:

- j is the integer value.
- s is the number of bits.
- w_k is a bit value of 0 or 1.

The bits are numbered from right to left beginning with 0.

The following example shows the bit model for j = 1001 (integer 9) using a bit number (s) of 4:

```
1     0     0     1
|     |     |     |
w3    w2    w1    w0
```

$$j = (w_0 \times 2^{0}) + (w_1 \times 2^{1}) + (w_2 \times 2^{2}) + (w_3 \times 2^{3})$$

$$j = 1 + 0 + 0 + 8$$

$$j = 9$$

Run-Time Library Routines

E

Intel® Fortran provides the following run-time library routines, which are summarized in this appendix:

- "Module Routines"
- "OpenMP* Fortran Routines"

For more information on these routines, see the *Libraries Reference*.

Module Routines

Intel® Fortran provides library modules containing the following routines:

- Routines that help you write programs for graphics, QuickWin, and other applications (in modules IFQWIN, IFLOGM, and IFCORE):
  - "QuickWin Routines (W*32, W*64)"
  - "Graphics Routines (W*32, W*64)"
  - "Dialog Routines (W*32)"
  - "Miscellaneous Run-Time Routines"
- Routines that help you write programs that use Component Object Model (COM) and Automation servers (in modules IFCOM and IFAUTO):
  - "COM Routines (W*32)"
  - "AUTO Routines (W*32)"
- "Portability Routines" that help you port your programs to or from other systems, or help you perform basic I/O to serial ports on Windows* systems on IA-32 processors (in module IFPORT).
- "National Language Support Routines (W*32, W*64)" that help you write foreign language programs for international markets (in module IFNLS).
- "POSIX* Routines" that help you write Fortran programs that comply with the POSIX* Standard (in module IFPOSIX).

When you include the statement USE *module-name* in your program, these library routines are automatically linked to your program if called.

You can restrict what is accessed from a USE module by adding ONLY clauses to the USE statement.

This appendix summarizes the library routines.

See Also

- "USE Statement"
- The section on using libraries in Volume I of your user's guide

Portability Routines

To use a portability routine, add the following statement to the program unit containing the routine:

```
USE IFPORT
```

Table E-1 summarizes portability routines.

Table E-1 Summary of Portability Routines

Name	Description
Information Retrieval:	
FSTAT	Returns information about a logical file unit.
GETCWD	Returns the pathname of the current working directory.
GETENV	Searches the environment for a given string and returns its value if found.
GETGID	Returns the group ID of the user.
GETLOG	Returns the user's login name.
GETPID	Returns the process ID of the process.
GETUID	Returns the user ID of the user of the process.
HOSTNAM[1]	Returns the name of the user's host.
ISATTY	Checks whether a logical unit number is a terminal.
RENAME	Renames a file.
STAT, LSTAT	Returns information about a named file.
UNLINK	Deletes the file given by path.
Process Control:	
ABORT	Stops execution of the current process, clears I/O buffers, and writes a string to external unit 0.

Table E-1 Summary of Portability Routines

Name	Description
ALARM	Executes an external subroutine after waiting a specified number of seconds.
KILL	Sends a signal code to the process given by ID.
SIGNAL	Changes the action for signal.
SLEEP	Suspends program execution for a specified number of seconds.
SYSTEM	Executes a command in a separate shell.
Numeric Values and Conversion:	
BESJ0, BESJ1, BESJN, BESY0, BESY1, BESYN	Return single-precision values of Bessel functions of the first and second kind of orders 1, 2, and n, respectively.
BIC, BIS, BIT	Perform bit level clear, set, and test for integers.
CDFLOAT	Converts a COMPLEX(4) argument to DOUBLE PRECISION type.
COMPLINT, COMPLREAL, COMPLLOG	Return a BIT-WISE complement or logical .NOT. of the argument.
CSMG	Performs an effective BIT-WISE store under mask.
DBESJ0, DBESJ1, DBESJN, DBESY0, DBESY1, DBESYN	Return double-precision values of Bessel functions of the first and second kind of orders 1, 2, and n, respectively.
DFLOATI, DFLOATJ, DFLOATK	Convert an integer to double-precision real type.
DRAND, DRANDM	Return double-precision random values in the range 0 through 1.0.
DRANSET	Sets the seed for the random number generator.
IDFLOAT	Converts an INTEGER(4) argument to double-precision real type.
IFLOATI, IFLOATJ	Convert an integer to single-precision real type.
INMAX	Returns the maximum positive value for an integer.
INTC	Converts an INTEGER(4) argument to INTEGER(2) type.
IRAND, IRANDM	Return a positive integer in the range 0 through 2**31-1 or 2**15-1 if called without an argument.
IRANGET	Returns the current seed.
IRANSET	Sets the seed for the random number generator.
JABS	Computes an absolute value.
LONG	Converts an INTEGER(2) argument to INTEGER(4) type.
QRANSET	Sets the seed for a sequence of pseudo-random numbers.
RAND, RANDOM[2]	Return random values in the range 0 through 1.0.
RANF	Generates a random number between 0.0 and RAND_MAX.
RANGET	Returns the current seed.

Table E-1 Summary of Portability Routines

Name	Description
RANSET	Sets the seed for the random number generator.
SEED	Changes the starting point of the random number generator.
SHORT	Converts an INTEGER(4) argument to INTEGER(2) type.
SRAND	Seeds the random number generator used with IRAND and RAND.
Input and Output:	
ACCESS	Checks a file for accessibility according to mode.
CHMOD	Changes file attributes.
FGETC	Reads a character from an external unit.
FLUSH	Flushes the buffer for an external unit to its associated file.
FPUTC	Writes a character to an external unit.
FSEEK	Repositions a file on an external unit.
FTELL, FTELLI8	Return the offset, in bytes, from the beginning of the file.
GETC	Reads a character from unit 5.
GETPOS, GETPOSI8	Returns the offset, in bytes, from the beginning of the file.
PUTC	Writes a character to unit 6.
Date and Time:	
CLOCK	Returns current time in "hh:mm:ss" format using a 24-hour clock.
CLOCKX	Returns the processor clock to the nearest microsecond.
CTIME	Converts a system time to a 24-character ASCII string.
DATE	Returns the current system date.
DATE4	Returns the current system date.
DCLOCK	Returns the elapsed time in seconds since the start of the current process.
DTIME	Returns CPU time since later of (1) start of program, or (2) most recent call to DTIME.
ETIME	Returns elapsed CPU time since the start of program execution.
FDATE	Returns the current date and time as an ASCII string.
GETDAT	Returns the date.
GETTIM	Returns the time.
GMTIME	Returns Greenwich Mean Time as a 9-element integer array.
IDATE	Returns the date either as one 3-element array or three scalar parameters (month, day, year).

Table E-1 Summary of Portability Routines

Name	Description
IDATE4	Returns the date either as one 3-element array or three scalar parameters (month, day, year).
ITIME	Returns current time as a 3-element array (hour, minute, second).
JDATE	Returns current date as an 8-character string with the Julian date.
JDATE4	Returns current date as a 10-character string with the Julian date.
LTIME	Returns local time as a 9-element integer array.
RTC	Returns number of seconds since 00:00:00 GMT, Jan 1, 1970.
SECNDS	Returns number of seconds since midnight, less the value of its argument.
SETDAT	Sets the date.
SETTIM	Sets the time.
TIME	As a subroutine, returns time formatted as HH:MM:SS; as a function, returns time in seconds since 00:00:00 GMT, Jan 1, 1970.
TIMEF	Returns the number of seconds since the first time this function was called (or zero).
Error Handling:	
GETLASTERROR	Returns the last error set.
GETLASTERRORQQ	Returns the last error set by a run-time function or subroutine.
IERRNO	Returns the last code error.
SETERRORMODEQQ	Sets the mode for handling critical errors.
Program Call and Control:	
RAISEQQ	Sends an interrupt to the executing program, simulating an interrupt from the operating system.
RUNQQ	Calls another program and waits for it to execute.
SIGNALQQ	Controls signal handling.
SLEEPQQ	Delays execution of the program for a specified time.
System, Drive, and Directory:	
CHDIR	Changes the current working directory.
CHANGEDIRQQ	Makes the specified directory the current (default) directory.
CHANGEDRIVEQQ	Makes the specified drive the current drive.
DELDIRQQ	Deletes a specified directory.
GETDRIVEDIRQQ	Returns the current drive and directory path.
GETDRIVESIZEQQ	Returns the size of the specified drive.

Table E-1 Summary of Portability Routines

Name	Description
GETDRIVESQQ	Returns the drives available to the system.
GETENVQQ	Returns a value from the current environment.
MAKEDIRQQ	Creates a directory with the specified directory name.
SETENVQQ	Adds a new environment variable or sets the value of an existing one.
SYSTEMQQ	Executes a command by passing a command string to the operating system's command interpreter.
Speakers:	
BEEPQQ	Sounds the speaker for a specified duration in milliseconds at a specified frequency in Hertz.
File Management:	
DELFILESQQ	Deletes the specified files in a specified directory.
FINDFILEQQ	Searches for a file in the directories specified in the PATH environment variable.
FULLPATHQQ	Returns the full path for a specified file or directory.
GETFILEINFOQQ	Returns information about files with names that match a request string.
PACKTIMEQQ	Packs time values for use by SETFILETIMEQQ.
RENAMEFILEQQ	Renames a file.
SETFILEACCESSQQ	Sets file-access mode for the specified file.
SETFILETIMEQQ	Sets modification time for the specified file.
SPLITPATHQQ	Breaks a full path into four components.
UNPACKTIMEQQ	Unpacks a file's packed time and date value into its component parts.
Arrays:	
BSEARCHQQ	Performs a binary search for a specified element on a sorted one-dimensional array of intrinsic type.
SORTQQ	Sorts a one-dimensional array of intrinsic type.
Floating-Point Inquiry and Control:	
CLEARSTATUSFPQQ	Clears the exception flags in the floating-point processor status word.
GETCONTROLFPQQ	Returns the value of the floating-point processor control word.
GETSTATUSFPQQ	Returns the value of the floating-point processor status word.
LCWRQQ	Same as SETCONTROLFPQQ.
SCWRQQ	Same as GETCONTROLFPQQ.
SETCONTROLFPQQ	Sets the value of the floating-point processor control word.

Table E-1 Summary of Portability Routines

Name	Description
SSWRQQ	Same as GETSTATUSFPQQ.
IEEE* Functionality:	
IEEE_FLAGS	Sets, gets, or clears IEEE flags.
IEEE_HANDLER	Establishes a handler for IEEE exceptions.
Serial Port I/O: [3]	
SPORT_CANCEL_IO	Cancels any I/O in progress to the specified port.
SPORT_CONNECT	Establishes the connection to a serial port.
SPORT_GET_HANDLE	Returns the WIN32* handle associated with the communications port.
SPORT_GET_STATE	Returns the baud rate, parity, data bits setting, and stop bits setting of the communications port.
SPORT_GET_TIMEOUTS	Returns the user selectable timeouts for the serial port.
SPORT_PEEK_DATA	Returns information about the availability of input data.
SPORT_PEEK_LINE	Returns information about the availability of input records.
SPORT_PURGE	Executes a purge function on the specified port.
SPORT_READ_DATA	Reads available data from the port specified.
SPORT_READ_LINE	Reads a record from the port specified.
SPORT_RELEASE	Releases a serial port that has previously been connected.
SPORT_SET_STATE	Sets the baud rate, parity, data bits setting, and stop bits setting of the communications port.
SPORT_SET_TIMEOUTS	Sets the user selectable timeouts for the serial port.
SPORT_SHOW_STATE	Displays the state of a port.
SPORT_SPECIAL_FUNC	Executes a communications function on a specified port.
SPORT_WRITE_DATA	Outputs data to a specified port.
SPORT_WRITE_LINE	Outputs data to a specified port and follows it with a record terminator.
Miscellaneous:	
LNBLNK	Returns the index of the last non-blank character in a string.
QSORT	Returns a sorted version of a one-dimensional array of a specified number of elements of a named size.
RINDEX	Returns the index of the last occurrence of a substring in a string.
SCANENV	Scans the environment for the value of an environment variable.

Table E-1 Summary of Portability Routines

Name	Description
TTYNAM	Checks whether a logical unit is a terminal.

1. This routine can also be specified as HOSTNM.
2. There is a RANDOM function and a RANDOM subroutine in the portability library.
3. W*32 only

For more information on these routines, see your *Libraries Reference* or the Intel® Visual Fortran online *Reference*.

National Language Support Routines (W*32, W*64)

National Language Support (NLS) routines provide language localization and a multibyte character set (MBCS) to let you write applications in different languages. To use an NLS routine, add the following statement to the program unit containing the routine:

```
USE IFNLS
```

Table E-2 summarizes the NLS routines. Routine names are shown in mixed case to make the names easier to understand. When writing your applications, you can use any case.

Table E-2 Summary of NLS Routines (W*32, W*64)

Name	Description
Locale Setting and Inquiry:	
NLSEnumCodepages	Returns all the supported codepages on the system.
NLSEnumLocales	Returns all the languages and country combinations supported by the system.
NLSGetEnvironmentCodepage	Returns the codepage number for the system codepage or the console codepage.
NLSGetLocale	Returns the current language, country, and codepage.
NLSGetLocaleInfo	Returns requested information about the current local code set.
NLSSetEnvironmentCodepage	Changes the codepage for the current console.
NLSSetLocale	Sets the language, country, and codepage.
Formatting:	
NLSFormatCurrency	Formats a number string and returns the correct currency string for the current locale.
NLSFormatDate	Returns a correctly formatted string containing the date for the current locale.

Table E-2 Summary of NLS Routines (W*32, W*64)

Name	Description
NLSFormatNumber	Formats a number string and returns the correct number string for the current locale.
NLSFormatTime	Returns a correctly formatted string containing the time for the current locale.
MBCS Inquiry:	
MBCharLen	Returns the length of the first multibyte character in a string.
MBCurMax	Returns the longest possible multibyte character for the current codepage.
MBLead	Determines whether a given character is the first byte of a multibyte character.
MBLen	Returns the number of multibyte characters in a string, including trailing spaces.
MBLen_Trim	Returns the number of multibyte characters in a string, not including trailing spaces.
MBNext	Returns the string position of the first byte of the multibyte character immediately after the given string position.
MBPrev	Returns the string position of the first byte of the multibyte character immediately before the given string position.
MBStrLead	Performs a context sensitive test to determine whether a given byte in a character string is a lead byte.
MBCS Conversion:	
MBConvertMBToUnicode	Converts a character string from a multibyte codepage to a Unicode string.
MBConvertUnicodeToMB	Converts a Unicode string to a multibyte character string of the current codepage.
MBJISTToJMS	Converts a Japan Industry Standard (JIS) character to a Microsoft* Kanji (Shift JIS or JMS) character.
MBJMSTToJIS	Converts a Microsoft Kanji (Shift JIS or JMS) character to a Japan Industry Standard (JIS) character.
MBCS Fortran Equivalents:	
MBINCHARQQ	Same as INCHARQQ except that it can read a single multibyte character at once and returns the number of bytes read.
MBINDEX	Same as INDEX except that multibyte characters can be included in its arguments.
MBLGE, MBLGT, MBLLE, MBLLT, MBLEQ, MBLNE	Same as LGE, LGT, LLE, LLT, and the operators .EQ. and .NE. except that multibyte characters can be included in their arguments.

Table E-2 Summary of NLS Routines (W*32, W*64)

Name	Description
MBSCAN	Same as SCAN except that multibyte characters can be included in its arguments.
MBVERIFY	Same as VERIFY except that multibyte characters can be included in its arguments.

For more information on these routines, see your *Libraries Reference* or the Intel® Visual Fortran online *Reference*.

POSIX* Routines

Intel Fortran provides routines that implement the IEEE POSIX FORTRAN-77 language bindings.To use a POSIX routine, add the following statement to the program unit containing the routine:

```
USE IFPOSIX
```

Table E-3 summarizes the Intel Fortran POSIX library routines.

Table E-3 Summary of POSIX Routines

Name	Description
IPXFARGC	Returns the index of the last command-line argument.
IPXFCONST	Returns the value associated with a constant defined in the C POSIX standard.
IPXFLENTRIM	Returns the index of the last non-blank character in an input string.
IPXFWEXITSTATUS[1]	Returns the exit code of a child process.
IPXFWSTOPSIG[1]	Returns the number of the signal that caused a child process to stop.
IPXFWTERMSIG[1]	Returns the number of the signal that caused a child process to terminate.
PXF<TYPE>GET	Gets the value stored in a component (or field) of a structure.
PXF<TYPE>SET	Sets the value of a component (or field) of a structure.
PXFA<TYPE>GET	Gets the array values stored in a component (or field) of a structure.
PXFA<TYPE>SET	Sets the value of an array component (or field) of a structure.
PXFACCESS	Determines the accessibility of a file.
PXFALARM	Schedules an alarm.
PXFCALLSUBHANDLE	Calls the associated subroutine.
PXFCFGETISPEED[1]	Returns the input baud rate from a termios structure.
PXFCFGETOSPEED[1]	Returns the output baud rate from a termios structure.

Table E-3 Summary of POSIX Routines

Name	Description
PXFCFSETISPEED[1]	Sets the input baud rate in a termios structure.
PXFCFSETOSPEED[1]	Sets the output baud rate in a termios structure.
PXFCHDIR	Changes the current working directory.
PXFCHMOD	Changes the ownership mode of the file.
PXFCHOWN[1]	Changes the owner and group of a file.
PXFCLEARENV	Clears the process environment.
PXFCLOSE	Closes the file associated with the descriptor.
PXFCLOSEDIR	Closes the directory stream.
PXFCONST	Returns the value associated with a constant.
PXFCNTL[1]	Manipulates an open file descriptor.
PXFCREAT	Creates a new file or rewrites an existing file.
PXFCTERMID[1]	Generates a terminal pathname.
PXFDUP, PXFDUP2	Duplicates an existing file descriptor.
PXFE<TYPE>GET	Gets the value stored in an array element component (or field) of a structure.
PXFE<TYPE>SET	Sets the value of an array element component (or field) of a structure.
PXFEXECV, PXFEXECVE, PXFEXECVP	Execute a new process by passing command-line arguments.
PXFEXIT, PXFFASTEXIT	Exits from a process.
PXFFDOPEN	Opens an external unit.
PXFFFLUSH	Flushes a file directly to disk.
PXFFGETC	Reads a character from a file.
PXFFILENO	Returns the file descriptor associated with a specified unit.
PXFFORK[1]	Creates a child process that differs from the parent process only in its PID.
PXFFPATHCONF	Gets the value for a configuration option of an opened file.
PXFFPUTC	Writes a character to a file.
PXFFSEEK	Modifies a file position.
PXFFSTAT	Gets a file's status information.
PXFFTELL	Returns the relative position in bytes from the beginning of the file.
PXFGETARG	Gets the specified command-line argument.
PXFGETATTY	Tests whether a file descriptor is connected to a terminal.
PXFGETC	Reads a character from standard input unit 5.
PXFGETCWD	Returns the path of the current working directory.

Table E-3 Summary of POSIX Routines

Name	Description
PXFGETEGID[1]	Gets the effective group ID of the current process.
PXFGETENV	Gets the setting of an environment variable.
PXFGETEUID[1]	Gets the effective user ID of the current process.
PXFGETGID[1]	Gets the real group ID of the current process.
PXFGETGRGID[1]	Gets group information for the specified GID.
PXFGETGRNAM[1]	Gets group information for the named group.
PXFGETGROUPS[1]	Gets supplementary group IDs.
PXFGETLOGIN	Gets the name of the user.
PXFGETPGRP[1]	Gets the process group ID of the calling process.
PXFGETPID	Gets the process ID of the calling process.
PXFGETPPID	Gets the process ID of the parent of the calling process.
PXFGETPWNAM[1]	Gets password information for a specified name.
PXFGETPWUID[1]	Gets password information for a specified UID.
PXFGETSUBHANDLE	Returns a handle for a subroutine.
PXFGETUID[1]	Gets the real user ID of the current process.
PXFISBLK	Tests for a block special file.
PXFISCHR	Tests for a character file.
PXFISCONST	Tests whether a string is a valid constant name.
PXFISDIR	Tests whether a file is a directory.
PXFISFIFO	Tests whether a file is a special FIFO file.
PXFISREG	Tests whether a file is a regular file.
PXFKILL	Sends a signal to a specified process.
PXFLINK	Creates a link to a file or directory.
PXFLOCALTIME	Converts a given elapsed time in seconds to local time.
PXFLSEEK	Positions a file a specified distance in bytes.
PXFMKDIR	Creates a new directory.
PXFMKFIFO[1]	Creates a new FIFO.
PXFOPEN	Opens or creates a file.
PXFOPENDIR	Opens a directory and associates a stream with it.
PXFPATHCONF	Gets the value for a configuration option of an opened file.
PXFPAUSE	Suspends process execution.

Table E-3 Summary of POSIX Routines

Name	Description
PXFPIPE	Creates a communications pipe between two processes.
PXFPOSIXIO	Sets the current value of the POSIX I/O flag.
PXFPUTC	Outputs a character to logical unit 6 (stdout).
PXFREAD	Reads from a file.
PXFREADDIR	Reads the current directory entry.
PXFRENAME	Changes the name of a file.
PXFREWINDDIR	Resets the position of the stream to the beginning of the directory.
PXFRMDIR	Removes a directory.
PXFSETENV	Adds a new environment variable or sets the value of an environment variable.
PXFSETGID[1]	Sets the effective group ID of the current process.
PXFSETPGID[1]	Sets the process group ID.
PXFSETSID[1]	Creates a session and sets the process group ID.
PXFSETUID[1]	Sets the effective user ID of the current process.
PXFSIGACTION	Changes the action associated with a specific signal.
PXFSIGADDSET[1]	Adds a signal to a signal set.
PXFSIGDELSET[1]	Deletes a signal from a signal set.
PXFSIGEMPTYSET[1]	Empties a signal set.
PXFSIGFILLSET[1]	Fills a signal set.
PXFSIGISMEMBER[1]	Tests whether a signal is a member of a signal set.
PXFSIGPENDING[1]	Examines pending signals.
PXFSIGPROCMASK[1]	Changes the list of currently blocked signals.
PXFSIGSUSPEND[1]	Suspends the process until a signal is received.
PXFSLEEP	Forces the process to sleep.
PXFSTAT	Gets the status of a file.
PXFSTRUCTCOPY	Copies the contents of one structure to another.
PXFSTRUCTCREATE	Creates an instance of the specified structure.
PXFSTRUCTFREE	Deletes the instance of a structure.
PXFSYSCONF	Gets values for system limits or options.
PXFTCDRAIN[1]	Waits until all output written has been transmitted.
PXFTCFLOW[1]	Suspends the transmission or reception of data.
PXFTCFLUSH[1]	Discards terminal input data, output data, or both.

Table E-3 Summary of POSIX Routines

Name	Description
PXFTCGETATTR[1]	Reads current terminal settings.
PXFTCGETPGRP[1]	Gets the foreground process group ID associated with the terminal.
PXFTCSENDBREAK[1]	Sends a break to the terminal.
PXFTCSETATTR[1]	Writes new terminal settings.
PXFTCSETPGRP[1]	Sets the foreground process group associated with the terminal.
PXFTIME	Gets the system time.
PXFTIMES	Gets process times.
PXFTTYNAM[1]	Gets the terminal pathname.
PXFUCOMPARE	Compares two unsigned integers.
PXFUMASK	Sets a new file creation mask and gets the previous one.
PXFUNAME	Gets the operation system name.
PXFUNLINK	Removes a directory entry.
PXFUTIME	Sets file access and modification times.
PXFWAIT[1]	Waits for a child process.
PXFWAITPID[1]	Waits for a specific PID.
PXFWIFEXITED[1]	Determines if a child process has exited.
PXFWIFSIGNALED[1]	Determines if a child process has exited because of a signal.
PXFWIFSTOPPED[1]	Determines if a child process has stopped.
PXFWRITE	Writes to a file.

1. L*X only

For more information on these routines, see your *Libraries Reference* or the Intel® Visual Fortran online *Reference*.

QuickWin Routines (W*32, W*64)

QuickWin routines help you turn graphics programs into simple Windows* applications. To use a Quickwin routine, add the following statement to the program unit containing the routine:

```
USE IFQWIN
```

Graphic routines are also used in QuickWin applications (see "Graphics Routines (W*32, W*64)").

Table E-4 summarizes QuickWin routines.

Table E-4 Summary of QuickWin Routines (W*32, W*64)

Name	Description
Window Control and Inquiry:	
FOCUSQQ	Sets focus to specified window.
GETACTIVEQQ	Returns the unit number of the currently active child.
GETHWNDQQ	Converts the unit number into a Windows handle for functions that require it.
GETUNITQQ	Returns the unit number corresponding to the specified Windows handle.
GETWINDOWCONFIG	Returns current window properties.
GETWSIZEQQ	Returns the size and position of a window.
INQFOCUSQQ	Determines which window has focus.
SETACTIVEQQ	Makes a child window active, but does not give it focus.
SETWINDOWCONFIG	Sets current window properties.
SETWSIZEQQ	Sets the size and position of a window.
QuickWin Application Enhancement:	
ABOUTBOXQQ	Adds an About Box with customized text.
APPENDMENUQQ	Appends a menu item.
CLICKMENUQQ	Simulates the effect of clicking or selecting a menu item.
DELETEMENUQQ	Deletes a menu item.
GETEXITQQ	Returns the setting for a QuickWin application's exit behavior.
INCHARQQ	Reads a single character input from the keyboard and returns the ASCII value of that character without any buffering.
INITIALSETTINGS	Controls initial menu settings and initial frame window.
INSERTMENUQQ	Inserts a menu item.
MESSAGEBOXQQ	Displays a message box.
MODIFYMENUFLAGSQQ	Modifies a menu item's state.
MODIFYMENUROUTINEQQ	Modifies a menu item's callback routine.
MODIFYMENUSTRINGQQ	Modifies a menu item's text string.
PASSDIRKEYSQQ	Determines the behavior of direction and page keys.
REGISTERMOUSEEVENT	Registers the application defined routines to be called on mouse events.
SETEXITQQ	Sets a QuickWin application's exit behavior.

Table E-4 Summary of QuickWin Routines (W*32, W*64)

Name	Description
SETMESSAGEQQ	Changes any QuickWin message, including status bar messages, state messages, and dialog box messages.
SETMOUSECURSOR	Sets the mouse cursor for the window in focus.
SETWINDOWMENUQQ	Sets the menu to which a list of current child window names are appended.
UNREGISTERMOUSEEVENT	Removes the routine registered by REGISTERMOUSEEVENT.
WAITONMOUSEEVENT	Blocks a return until a mouse event occurs.
Color Conversion:	
INTEGERTORGB	Converts an RGB color value to its red, green, and blue components.
RGBTOINTEGER	Converts integers specifying red, green, and blue color into an RGB integer (for use in RGB routines).

For more information on these routines, see your *Libraries Reference* or the Intel® Visual Fortran online *Reference*.

Graphics Routines (W*32, W*64)

The graphics routines can be used in Standard Graphics applications and in Quickwin applications (see "QuickWin Routines (W*32, W*64)"). To use a graphics routine, do the following:

- Add the following statement to the program unit containing the routine:

  USE IFQWIN
- Choose the QuickWin Graphics or Standard Graphics program type.

Table E-5 summarizes graphics routines.

Table E-5 Summary of Graphics Routines (W*32, W*64)

Name	Description
Color Control or Inquiry:[1]	
FLOODFILL	Fills an area using the current index and fill mask; fill starting point uses viewport coordinates.
FLOODFILL_W	Fills an area using the current index and fill mask; fill starting point uses window coordinates.
FLOODFILLRGB	Fills an area using the current RGB color and fill mask; fill starting point uses viewport coordinates.
FLOODFILLRGB_W	Fills an area using the current RGB color and fill mask; fill starting point uses viewport coordinates.

Table E-5 Summary of Graphics Routines (W*32, W*64)

Name	Description
GETBKCOLOR	Returns current background color index for both text and graphics.
GETBKCOLORRGB	Returns current background RGB color value for both text and graphics.
GETCOLOR	Returns the current graphics color index.
GETCOLORRGB	Returns the current graphics color RGB value.
GETPIXEL	Returns the color index of a pixel; pixel is located using viewport coordinates.
GETPIXEL_W	Returns the color index of a pixel; pixel is located using window coordinates.
GETPIXELRGB	Returns the RGB color value of a pixel; pixel is located using viewport coordinates.
GETPIXELRGB_W	Returns the RGB color value of a pixel; pixel is located using window coordinates.
GETPIXELS	Returns the color indexes of multiple pixels.
GETPIXELSRGB	Returns the RGB color values of multiple pixels.
GETTEXTCOLOR	Returns the current text color index.
GETTEXTCOLORRGB	Returns the RGB color value of the current text.
REMAPALLPALETTERGB	Remaps an entire palette to an RGB color.
REMAPPALETTERGB	Remaps one color index to an RGB color.
SETBKCOLOR	Sets current background color index for both text and graphics.
SETBKCOLORRGB	Sets current background RGB color value for both text and graphics.
SETCOLOR	Sets the current graphics color index.
SETCOLORRGB	Sets the current graphics color to an RGB value.
SETPIXEL	Sets a pixel to the current graphics color index; pixel is located using viewport coordinates.
SETPIXEL_W	Sets a pixel to the current graphics color index; pixel is located using window coordinates.
SETPIXELRGB	Sets a pixel to an RGB color value; pixel is located using viewport coordinates.
SETPIXELRGB_W	Sets a pixel to an RGB color value; pixel is located using window coordinates.
SETPIXELS	Sets the color indexes of multiple pixels.
SETPIXELSRGB	Sets multiple pixels to an RGB color.
SETTEXTCOLOR	Sets the current text color index.
SETTEXTCOLORRGB	Sets the current text color to an RGB value.

Table E-5 Summary of Graphics Routines (W*32, W*64)

Name	Description
Figure Characteristics:	
GETFILLMASK	Returns the current fill mask.
GETLINESTYLE	Returns the current line style.
GETWRITEMODE	Returns the logical write mode used when drawing lines.
SETCLIPRGN	Masks part of the screen; it does not change the viewport coordinates.
SETFILLMASK	Sets the current fill mask.
SETLINESTYLE	Sets the current line style.
SETWRITEMODE	Sets the logical write mode used when drawing lines.
Coordinate Conversion and Settings:	
GETPHYSCOORD	Converts viewpoint coordinates to physical coordinates.
GETVIEWCOORD	Converts physical coordinates to viewport coordinates.
GETVIEWCOORD_W	Converts window coordinates to viewport coordinates.
GETWINDOWCOORD	Converts viewport coordinates to window coordinates.
SETVIEWORG	Moves the viewport coordinate origin (0,0) to a specified physical point.
SETVIEWPORT	Redefines viewport bounds to the specified limits and sets the viewport coordinate origin to the upper-left corner of this region.
SETWINDOW	Defines a window bound by specified window coordinates.
Graphics Drawing:	
ARC	Draws an arc using viewport coordinates.
ARC_W	Draws an arc using window coordinates.
CLEARSCREEN	Clears the screen, viewport, or text window.
ELLIPSE	Draws an ellipse or circle using viewport coordinates.
ELLIPSE_W	Draws an ellipse or circle using window coordinates.
GETARCINFO	Returns the endpoints of the most recently drawn arc or pie.
GETCURRENTPOSITION	Returns the viewport coordinates of the current graphics-output position.
GETCURRENTPOSITION_W	Returns the window coordinates of the current graphics-output position.
GRSTATUS	Returns the status (success or failure) of the most recently called graphics routine.
LINETO	Draws a line from the current graphics-output position to a specified point using viewport coordinates.
LINETO_W	Draws a line from the current graphics-output position to a specified point using window coordinates.

Table E-5 Summary of Graphics Routines (W*32, W*64)

Name	Description
LINETOAR	Draws a line between points in one array and corresponding points in another array.
LINETOAREX	Similar to LINETOAR, but also lets you specify color and line style.
MOVETO	Moves the current graphics-output position to a specified point using viewport coordinates.
MOVETO_W	Moves the current graphics-output position to a specified point using window coordinates.
PIE	Draws a pie-slice-shaped figure using viewport coordinates.
PIE_W	Draws a pie-slice-shaped figure using window coordinates.
POLYBEZIER	Draws a Bezier curve using viewport coordinates.
POLYBEZIER_W	Draws a Bezier curve using window coordinates.
POLYBEZIERTO	Draws a Bezier curve using viewport coordinates.
POLYBEZIERTO_W	Draws a Bezier curve using window coordinates.
POLYGON	Draws a polygon using viewport coordinates.
POLYGON_W	Draws a polygon using window coordinates.
POLYLINEQQ	Draws a line between successive points in an array.
RECTANGLE	Draws a rectangle using viewport coordinates.
RECTANGLE_W	Draws a rectangle using window coordinates.
Character-Based Text Display:	
DISPLAYCURSOR	Sets the cursor on or off.
GETTEXTPOSITION	Returns the current text-output position.
GETTEXTWINDOW	Returns the boundaries of the current text window.
OUTTEXT	Sends text to the screen at the current position.
SCROLLTEXTWINDOW	Scrolls the contents of a text window.
SETTEXTCURSOR	Sets the height and width of the text cursor for the window in focus.
SETTEXTPOSITION	Sets the current text-output position.
SETTEXTWINDOW	Sets the boundaries of the current text window.
WRAPON	Turns line wrapping on or off.
Font-Based Character Display:	
GETFONTINFO	Returns the current font characteristics.
GETGTEXTEXTENT	Returns the width of specified text in the current font.
GETGTEXTROTATION	Returns the current orientation of the font text output by OUTGTEXT.

Table E-5 Summary of Graphics Routines (W*32, W*64)

Name	Description
INITIALIZEFONTS	Initializes the font library.
OUTGTEXT	Sends text in the current font to the screen at the current position.[2]
SETFONT	Finds one font that matches a specified set of characteristics and makes it the current font used by OUTGTEXT.
SETGTEXTROTATION	Sets the orientation angle of font text output in degrees.
Image Transfers in Memory:	
GETIMAGE	Stores a screen image using viewport coordinates.
GETIMAGE_W	Stores a screen image using window coordinates.
IMAGESIZE	Returns a viewport-coordinate image size in bytes.
IMAGESIZE_W	Returns a window-coordinate image size in bytes.
PUTIMAGE	Retrieves a viewport-coordinate image from memory and displays it.
PUTIMAGE_W	Retrieves a window-coordinate image from memory and displays it.
Image Loading and Saving:	
LOADIMAGE	Reads a Windows bitmap file (.BMP) from disk and displays it as specified viewport coordinates.
LOADIMAGE_W	Reads a Windows bitmap file (.BMP) from disk and displays it as specified window coordinates.
SAVEIMAGE	Saves an image from a specified part of the screen and saves it as a Windows bitmap file; screen location is specified using viewport coordinates.
SAVEIMAGE_W	Saves an image from a specified part of the screen and saves it as a Windows bitmap file; screen location is specified using window coordinates.

1. RGB is Red-Green-Blue
2. OUTGTEXT allows use of special fonts; OUTTEXT does not

For more information on these routines, see your *Libraries Reference* or the Intel® Visual Fortran online *Reference*.

Dialog Routines (W*32)

Dialog routines let you add dialog boxes to Windows, QuickWin, and console applications. To activate a dialog box, add the following statement to the application's relevant program unit:

```
USE IFLOGM
```

Table E-6 summarizes the dialog routines.

Table E-6 Summary of Dialog Routines (W*32)

Name	Description
DLGEXIT	Closes an open dialog box.
DLGFLUSH	Updates the display of a dialog box.
DLGGET	Returns the value of a control variable.
DLGGETCHAR	Returns the value of a character control variable.
DLGGETINT	Returns the value of an integer control variable.
DLGGETLOG	Returns the value of a logical control variable.
DLGINIT	Initializes a dialog box.
DLGINITWITHRESOURCEHANDLE	Initializes a dialog box.
DLGISDLGMESSAGE	Determines whether a message is intended for a modeless dialog box.
DLGISDLGMESSAGEWITHDLG	Determines whether a message is intended for a specific modeless dialog box.
DLGMODAL	Displays a dialog box.
DLGMODALWITHPARENT	Displays a dialog box and indicates the parent window.
DLGMODELESS	Displays a modeless dialog box.
DLGSENDCTRLMESSAGE	Sends a message to a dialog box control.
DLGSET	Assigns a value to a control variable.
DLGSETCHAR	Assigns a value to a character control variable.
DLGSETCTRLEVENTHANDLER	Assigns user-written event handlers to ActiveX* controls in a dialog box.
DLGSETINT	Assigns a value to an integer control variable.
DLGSETLOG	Assigns a value to a logical control variable.
DLGSETRETURN	Sets the return value for DLGMODAL.
DLGSETSUB	Assigns a defined callback routine to a control.
DLGSETTITLE	Sets the title of a dialog box.
DLGUNINIT	Deallocates memory for an initialized dialog box.

For more information on these routines, see your *Libraries Reference* or the Intel® Visual Fortran online *Reference*.

Miscellaneous Run-Time Routines

Intel Fortran provides several miscellaneous routines for applications. To use for_rtl_init_ and for_rtl_finish_, you must call them from a main program written in C. To use the other routines, add the following statement to the program unit containing the routine:

USE IFCORE

Table E-7 summarizes these run-time routines:

Table E-7 Summary of Miscellaneous Run-Time Routines

Name	Description
Keyboards and Speakers:	
GETCHARQQ	Returns the next keyboard keystroke.
GETSTRQQ	Reads a character string from the keyboard using buffered input.
PEEKCHARQQ	Checks the buffer to see if a keystroke is waiting.
File Management:	
COMMITQQ	Executes any pending write operations for the file associated with the specified unit to the file's physical device.
Error Handling:	
GERROR	Returns the IERRNO error code as a string variable.
PERROR	Returns an error message, preceded by a string, for the last error detected.
Floating-Point Inquiry and Control:	
FOR_GET_FPE	Returns the current settings of floating-point exception flags.
FOR_SET_FPE	Sets the floating-point exception flags.
GETEXCEPTIONPTRSQQ[1]	Returns a pointer to C run-time exception information pointers appropriate for use in signal handlers established with SIGNALQQ or direct calls to the C rtl signal( ) routine.
Run-Time Environment:	
for_rtl_finish_	Cleans up the Fortran run-time environment.
for_rtl_init_	Initializes the Fortran run-time environment.
Reentrancy Mode Control:	
FOR_SET_REENTRANCY	Controls the type of reentrancy protection that the Run-Time Library exhibits.
Traceback:	
TRACEBACKQQ	Generates a stack trace.

Table E-7 Summary of Miscellaneous Run-Time Routines

Name	Description
Memory assignment:	
FOR_DESCRIPTOR_ASSIGN[1]	Creates an array descriptor in memory.

1. W*32, W*64

For more information on these routines, see your *Libraries Reference* or the Intel® Visual Fortran online *Reference*.

COM Routines (W*32)

COM routines help you work with COM objects. To use a COM routine, add the following statement to the program unit containing the routine:

```
USE IFCOM
```

Some of the routines may also require the statement USE IFWINTY.

Table E-8 summarizes COM routines. Routine names are shown in mixed case to make the names easier to understand. When writing your applications, you can use any case.

Table E-8 Summary of COM Routines (W*32)

Name	Description
COMAddObjectReference	Adds a reference to an object's interface.
COMCLSIDFromProgID	Passes a programmatic identifier and returns the corresponding class identifier.
COMCLSIDFromString	Passes a class identifier string and returns the corresponding class identifier.
COMCreateObjectByGUID	Passes a class identifier, creates an instance of an object, and returns a pointer to the object's interface.
COMCreateObjectByProgID	Passes a programmatic identifier, creates an instance of an object, and returns a pointer to the object's IDispatch interface.
COMGetActiveObjectByGUID	Passes a class identifier and returns a pointer to the interface of a currently active object.
COMGetActiveObjectByProgID	Passes a programmatic identifier and returns a pointer to the IDispatch interface of a currently active object.
COMGetFileObject	Passes a file name and returns a pointer to the IDispatch interface of an automation object that can manipulate the file.
COMInitialize	Initializes the COM library.

Table E-8 Summary of COM Routines (W*32)

Name	Description
COMIsEqualGUID	Determines whether two globally unique identifiers (GUIDs) are the same.
COMQueryInterface	Passes an interface identifier and returns a pointer to an object's interface.
COMReleaseObject	Indicates that the program is done with a reference to an object's interface.
COMStringFromGUID	Passes a globally unique identifier (GUID) and returns a string of printable characters.
COMUninitialize	Uninitializes the COM library.

For more information on these routines, see your *Libraries Reference* or the Intel® Visual Fortran online *Reference*.

AUTO Routines (W*32)

AUTO routines help you work with automation objects. To use an AUTO routine, add the following statement to the program unit containing the routine:

```
USE IFAUTO
```

Some of the routines may also require the statement USE IFWINTY.

Table E-9 summarizes AUTO routines. Routine names are shown in mixed case to make the names easier to understand. When writing your applications, you can use any case.

Table E-9 Summary of AUTO Routines (W*32)

Name	Description
AUTOAddArg	Passes an argument name and value and adds the argument to the argument list data structure.
AUTOAllocateInvokeArgs	Allocates an argument list data structure that holds the arguments to be passed to AUTOInvoke.
AUTODeallocateInvokeArgs	Deallocates an argument list data structure.
AUTOGetExceptInfo	Retrieves the exception information when a method has returned an exception status.
AUTOGetProperty	Passes the name or identifier of the property and gets the value of the automation object's property.
AUTOGetPropertyByID	Passes the member ID of the property and gets the value of the automation object's property into the argument list's first argument.

Table E-9 Summary of AUTO Routines (W*32)

Name	Description
AUTOGetPropertyInvokeArgs	Passes an argument list data structure and gets the value of the automation object's property specified in the argument list's first argument.
AUTOInvoke	Passes the name or identifier of an object's method and an argument list data structure and invokes the method with the passed arguments.
AUTOSetProperty	Passes the name or identifier of the property and a value, and sets the value of the automation object's property.
AUTOSetPropertyByID	Passes the member ID of the property and sets the value of the automation object's property, using the argument list's first argument.
AUTOSetPropertyInvokeArgs	Passes an argument list data structure and sets the value of the automation object's property specified in the argument list's first argument.

For more information on these routines, see your *Libraries Reference* or the Intel® Visual Fortran online *Reference*.

OpenMP* Fortran Routines

Intel Fortran provides OpenMP* Fortran library routines you can use for directed parallel decomposition. To use an OpenMP Fortran routine, add the following statement to the program unit containing the routine:

```
USE OMP_LIB
```

For more information on a specific routine, see your user's guide or the appropriate reference page; for example, for more information on OMP_SET_LOCK, see `omp_set_lock(3f)`.

Table E-10 summarizes the Intel Fortran OpenMP Fortran API run-time library routines. These routines are all external procedures.

Table E-10 Summary of OpenMP Fortran Routines

Name	Description
OMP_SET_NUM_THREADS	Sets the number of threads to use for the next parallel region.
OMP_GET_NUM_THREADS	Gets the number of threads currently in the team executing the parallel region from which the routine is called.
OMP_GET_MAX_THREADS	Gets the maximum value that can be returned by calls to the OMP_GET_NUM_THREADS function.
OMP_GET_THREAD_NUM	Gets the thread number, within the team, in the range from zero to OMP_GET_NUM_THREADS minus one.

Table E-10 Summary of OpenMP Fortran Routines

Name	Description
OMP_GET_NUM_PROCS	Gets the number of processors that are available to the program.
OMP_IN_PARALLEL	Informs whether or not a region is executing in parallel.
OMP_SET_DYNAMIC	Enables or disables dynamic adjustment of the number of threads available for execution of parallel regions.
OMP_GET_DYNAMIC	Informs whether or not dynamic thread adjustment is enabled.
OMP_SET_NESTED	Enables or disables nested parallelism.
OMP_GET_NESTED	Informs whether or not nested parallelism is enabled.
OMP_INIT_LOCK	Initializes a lock for use in subsequent calls.
OMP_DESTROY_LOCK	Disassociates a lock variable from any locks.
OMP_SET_LOCK	Makes the executing thread wait until the specified lock is available.
OMP_UNSET_LOCK	Releases the executing thread from ownership of a lock.
OMP_TEST_LOCK	Tries to set the lock associated with a lock variable.
OMP_INIT_NEST_LOCK	Initializes a nested lock for use in subsequent calls.
OMP_DESTROY_NEST_LOCK	Disassociates a lock variable from a nested lock.
OMP_SET_NEST_LOCK	Makes the executing thread wait until the specified nested lock is available.
OMP_UNSET_NEST_LOCK	Releases the executing thread from ownership of a nested lock if the nesting count is zero.
OMP_TEST_NEST_LOCK	Tries to set the nested lock associated with a lock variable.
OMP_GET_WTIME	Returns a double-precision value equal to the elapsed wallclock time (in seconds) relative to an arbitrary reference time.
OMP_GET_WTICK	Returns a double-precision value equal to the number of seconds between successive clock ticks.
Intel® Fortran Extensions:	
KMP_GET_STACKSIZE_S[1]	Returns the number of bytes that will be allocated for each parallel thread to use as its private stack.
KMP_SET_STACKSIZE_S[2]	Sets the number of bytes that will be allocated for each parallel thread to use as its private stack.
KMP_GET_BLOCKTIME	Returns the number of milliseconds that a thread should wait, after completing the execution of a parallel region, before sleeping.
KMP_SET_BLOCKTIME	Sets the number of milliseconds that a thread should wait, after completing the execution of a parallel region, before sleeping.
KMP_MALLOC	Allocates a memory block of a specified size (in bytes) from the thread-local heap.

Table E-10 Summary of OpenMP Fortran Routines

Name	Description
KMP_CALLOC	Allocates an array of a specified number of elements and size from the thread-local heap.
KMP_REALLOC	Reallocates a memory block at a specified address and of a specified size from the thread-local heap.
KMP_FREE	Frees a memory block at a specified address from the thread-local heap.

1. For backwards compatibility, this can also be specified as KMP_GET_STACKSIZE.
2. For backwards compatibility, this can also be specified as KMP_SET_STACKSIZE.

See Also

- "OpenMP* Fortran Compiler Directives"
- Volume II of your user's guide for details on OpenMP Fortran routines

Summary of Language Extensions

F

This appendix summarizes the Intel® Fortran language extensions to the ANSI/ISO Fortran 95 Standard.

Most extensions are available on all supported operating systems. However, some extensions are limited to one or more platforms. If an extension is limited, it is labeled.

Source Forms

The following are extensions to the methods and rules for source forms:

- Tab-formatting as a method to code lines (see "Tab-Format Lines")
- The letter D as a debugging statement indicator in column 1 of fixed or tab source form (see "Fixed and Tab Source Forms")
- An optional statement field width of 132 columns for fixed or tab source form (see "Fixed and Tab Source Forms")
- An optional sequence number field for fixed source form (see "Fixed-Format Lines")
- Up to 511 continuation lines in a source program (see "Source Forms")

Names

The following are extensions to the rules for names (see "Names"):

- Names can contain up to 63 characters
- The dollar sign ($) is a valid character in names, and can be the first character

Character Sets

The following are extensions to the standard character set:

- The Tab (<Tab>) character (see "Character Sets")

- For Windows systems (see online Help):
  - Special ASCII and ANSI character sets
  - Key code sets

Intrinsic Data Types

The following table lists data-type extensions and an extended way to specify data types (see "Intrinsic Data Types"):

BYTE	INTEGER*1	REAL*16
DOUBLE COMPLEX	INTEGER*2	COMPLEX*8
LOGICAL*1	INTEGER*4	COMPLEX*16
LOGICAL*2	INTEGER*8	COMPLEX*32
LOGICAL*4	REAL*4	
LOGICAL*8	REAL*8	

Constants

C strings are allowed in character constants as an extension (see "C Strings in Character Constants").

Hollerith constants are allowed as an extension (see "Hollerith Constants").

Expressions and Assignment

When operands of different intrinsic data types are combined in expressions, conversions are performed as necessary (see "Data Type of Numeric Expressions").

Binary, octal, hexadecimal, and Hollerith constants can appear wherever numeric constants are allowed (see "Binary, Octal, Hexadecimal, and Hollerith Constants").

The following are extensions allowed in logical expressions (see "Logical Expressions"):

- .XOR. as a synonym for .NEQV.
- Integers as valid logical items
- Logical operators applied to integers bit-by-bit

Specification Statements

The following specification attributes and statements are extensions:

- AUTOMATIC and STATIC (see "AUTOMATIC and STATIC Attributes and Statements")
- VOLATILE (see "VOLATILE Attribute and Statement")

Execution Control

The following control statements are extensions to Fortran 95 (see Chapter 7, "Execution Control"):

- ASSIGN
- Assigned GO TO
- PAUSE

These are older Fortran features that have been deleted in Fortran 95. Intel Fortran fully supports these features.

Compilation Control Statements

The following statement and option are extensions that can influence compilation:

- /[NO]LIST, which can be specified for an INCLUDE statement (see "INCLUDE Statement")
- The OPTIONS statement (see "OPTIONS Statement")

Built-In Functions

The following built-in functions are extensions:

- %VAL, %REF, and %LOC, which facilitate references to non-Fortran procedures (see "References to Non-Fortran Procedures")
- %FILL, which can be used in record structure type definitions (see "Type Declarations")

I/O Statements

The following I/O statements are extensions:

- The ACCEPT statement (see "ACCEPT Statement")
- The TYPE statement, which is a synonym for the PRINT statement (see "PRINT and TYPE Statements")
- The REWRITE statement (see "REWRITE Statement")

I/O Formatting

The following are extensions allowed in I/O formatting:

- The Q edit descriptor (see "Character Count Editing (Q)")
- The dollar sign ($) edit descriptor (see "Dollar Sign ($) and Backslash (\) Editing" and carriage control character (see "Printing of Formatted Records")
- The backslash (\) edit descriptor (see "Dollar Sign ($) and Backslash (\) Editing")
- The ASCII NUL carriage control character (see "Printing of Formatted Records")
- Variable format expressions (see "Variable Format Expressions")
- The H edit descriptor (see "H Editing")

  This is an older Fortran feature that has been deleted in Fortran 95. Intel Fortran fully supports this feature.

File Operation Statements

The following statement specifiers and statements are extensions (see Chapter 12, "File Operation I/O Statements"):

- 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:
  - BINARY (W*32, W*64)
  - BLOCKSIZE
  - BUFFERED
  - CARRIAGECONTROL
  - CONVERT
  - DEFAULTFILE
  - FORM values: 'UNKNOWN', 'BINARY' (W*32, W*64)
  - IOFOCUS (W*32, W*64)
  - MODE as a synonym for ACTION
  - ORGANIZATION
  - RECORDTYPE
  - SHARE (W*32, W*64)
- OPEN statement specifiers:
  - ACCESS values: 'APPEND'
  - ASSOCIATEVARIABLE
  - BLOCKSIZE

  - BUFFERCOUNT
  - BUFFERED
  - CARRIAGECONTROL
  - CONVERT
  - DEFAULTFILE
  - DISPOSE (or DISP)
  - FORM value: 'BINARY' (W*32, W*64)
  - IOFOCUS (W*32, W*64)
  - MAXREC
  - MODE as a synonym for ACTION
  - NAME as a synonym for FILE
  - ORGANIZATION
  - READONLY
  - RECORDSIZE as a synonym for RECL
  - RECORDTYPE
  - SHARE (W*32, W*64*)
  - SHARED
  - TITLE (W*32, W*64)
  - TYPE as a synonym for STATUS
  - USEROPEN
- UNLOCK statement

Compiler Directives

The following general directives are extensions (see "General Compiler Directives"):

- ALIAS
- ATTRIBUTES
- DECLARE and NODECLARE
- DEFINE and UNDEFINE
- DISTRIBUTE POINT
- FIXEDFORMLINESIZE
- FREEFORM and NOFREEFORM
- IDENT
- IF and IF DEFINED
- INTEGER

- IVDEP
- LOOP COUNT
- MESSAGE
- OBJCOMMENT
- OPTIONS
- PACK
- PARALLEL and NOPARALLEL
- PREFETCH and NOPREFETCH
- PSECT
- REAL
- STRICT and NOSTRICT
- SWP and NOSWP (i64 only)
- TITLE and SUBTITLE
- UNROLL and NOUNROLL
- VECTOR ALIGNED and VECTOR UNALIGNED (i32 only)
- VECTOR ALWAYS and NOVECTOR (i32 only)
- VECTOR NONTEMPORAL (i32 only)

The following OpenMP Fortran parallel directives are extensions (see "OpenMP* Fortran Compiler Directives"):

- ATOMIC
- BARRIER
- CRITICAL
- DO
- FLUSH
- MASTER
- ORDERED
- PARALLEL
- PARALLEL DO
- PARALLEL SECTIONS
- SECTIONS
- SINGLE
- THREADPRIVATE

Intrinsic Procedures

The following intrinsic procedures are extensions (see Chapter 9, "Intrinsic Procedures"):

A to D			
ACOSD	BIAND	COSD	DCMPLX
ACOSH	BIEOR	COTAN	DCONJG
AIMAX0	BIOR	COTAND	DCOSD
AIMIN0	BITEST	CQABS	DCOTAN
AJMAX0	BIXOR	CQCOS	DCOTAND
AJMIN0	BJTEST	CQEXP	DERF
AKMAX0	BKTEST	CQLOG	DERFC
AKMIN0	BMOD	CQSIN	DFLOAT
AND	BMVBITS	CQSQRT	DFLOTI
ASIND	BNOT	CQTAN	DFLOTJ
ASINH	BSHFT	CTAN	DFLOTK
ATAN2D	BSHFTC	DACOSD	DIMAG
ATAND	BSIGN	DACOSH	DNUM
ATANH	CACHESIZE	DASIND	DREAL
BABS	CDABS	DASINH	DSHIFTL
BADDRESS	CDCOS	DATAN2D	DSHIFTR
BBCLR	CDEXP	DATAND	DSIND
BBITS	CDLOG	DATANH	DTAND
BBSET	CDSIN	DATE	
BBTEST	CDSQRT	DBLE	
BDIM	CDTAN	DBLEQ	

E to I			
EOF	HIXOR	IIDINT	IMVBITS
ERF	HMOD	IIDNNT	ININT
ERFC	HMVBITS	IIEOR	INOT
ERRSNS	HNOT	IIFIX	INT1
EXIT	HSHFT	IINT	INT2
FLOATI	HSHFTC	IIOR	INT4
FLOATJ	HSIGN	IIQINT	INT8
FLOATK	HTEST	IIQNNT	INT_PTR_KIND

FP_CLASS	IADDR	IISHFT	INUM
FREE	IARG	IISHFTC	IQINT
GETARG	IARGC	IISIGN	IQNINT
HABS	IARGPTR	IIXOR	ISHA
HBCLR	IBCHNG	IJINT	ISHC
HBITS	IDATE	ILEN	ISHL
HBSET	IIABS	IMAG	ISNAN
HDIM	IIAND	IMAX0	IXOR
HFIX	IIBCLR	IMAX1	IZEXT
HIAND	IIBITS	IMIN0	
HIEOR	IIBSET	IMIN1	
HIOR	IIDIM	IMOD	
J to P			
JFIX	JIXOR	KIDINT	KNOT
JIABS	JMAX0	KIDNNT	KZEXT
JIAND	JMAX1	KIEOR	LEADZ
JIBCLR	JMIN0	KIFIX	LOC
JIBITS	JMIN1	KINT	LSHIFT
JIBSET	JMOD	KIOR	LSHFT
JIDIM	JMVBITS	KIQINT	MALLOC
JIDINT	JNINT	KIQNNT	MCLOCK
JIDNNT	JNOT	KISHFT	MM_PREFETCH
JIEOR	JNUM	KISHFTC	MULT_HIGH
JIFIX	JZEXT	KISIGN	NARGS
JINT	KDIM	KMAX0	NUMARG
JIOR	KIABS	KMAX1	OR
JIQINT	KIAND	KMIN0	POPCNT
JIQNNT	KIBCLR	KMIN1	POPPAR
JISHFT	KIBITS	KMOD	
JISHFTC	KIBSET	KMVBITS	
JISIGN	KIDIM	KNINT	
Q to Z			
QABS	QCOTAN	QNUM	SHIFTR

QACOS	QCOTAND	QREAL	SIND
QACOSD	QDIM	QSIGN	SIZEOF
QACOSH	QERF	QSIN	SNGLQ
QASIN	QERFC	QSIND	TAND
QASIND	QEXP	QSINH	TIME
QASINH	QEXT	QSQRT	TRAILZ
QATAN	QEXTD	QTAN	XOR
QATAN2	QFLOAT	QTAND	ZABS
QATAN2D	QIMAG	QTANH	ZCOS
QATAND	QINT	RAN	ZEXP
QATANH	QLOG	RANDU	ZEXT
QCMPLX	QLOG10	RNUM	ZLOG
QCONJG	QMAX1	RSHIFT	ZSIN
QCOS	QMIN1	RSHFT	ZSQRT
QCOSD	QMOD	SECNDS	ZTAN
QCOSH	QNINT	SHIFTL	

The argument KIND is an extension available in the following intrinsic procedures:

COUNT	LEN	SCAN	VERIFY
ICHAR	LEN_TRIM	SHAPE	
INDEX	MAXLOC	SIZE	
LBOUND	MINLOC	UBOUND	

Additional Language Features

The following are language extensions that facilitate compatibility with other versions of Fortran:

- The DEFINE FILE statement (see "DEFINE FILE Statement")
- The ENCODE and DECODE statements (see "ENCODE and DECODE Statements")
- The FIND statement (see "FIND Statement")
- The INTERFACE TO statement (see "INTERFACE TO Statement")
- FORTRAN-66 Interpretation of the EXTERNAL Statement (see "FORTRAN-66 Interpretation of the EXTERNAL Statement")
- An alternative syntax for the PARAMETER statement (see "Alternative Syntax for the PARAMETER Statement")

- The VIRTUAL statement (see "VIRTUAL Statement")
- The AND, OR, XOR, IMAG, LSHIFT, RSHIFT intrinsics (see Table 9-3)
- An alternative syntax for octal and hexadecimal constants (see "Alternative Syntax for Octal and Hexadecimal Constants")
- An alternative syntax for an I/O record specifier (see "Alternative Syntax for a Record Specifier")
- An alternate syntax for the DELETE statement (see "Alternative Syntax for the DELETE Statement")
- An alternative form for namelist external records (see "Alternative Form for Namelist External Records")
- An integer POINTER statement (see "Integer POINTER Statement")
- Record structures (see "Record Structures")

Run-Time Library Routines

The following run-time library routines are available as extensions (see Appendix F, "Summary of Language Extensions"):

- Modules routines
- OpenMP Fortran routines

Glossary

This glossary contains terms that are commonly used in this manual and your user's guide. The terms and short descriptions are informative and are not part of the standard definition of the Fortran 95/90 programming language.

A

absolute pathname

A directory path specified in fixed relationship to the root directory. On Linux* systems, the first character is a slash (/). On Windows* systems, the first character is a backslash (\).

actual argument

A value (a variable, expression, or procedure) passed from a calling program unit to a subprogram (function or subroutine). *See also* **dummy argument**.

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. The array's rank is specified at compile time, but its bounds are determined at run time. 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.)

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

Can be either of the following:

- An actual argument—A variable, expression, or procedure passed from a calling program unit to a subprogram. *See also* **actual argument**.
- A 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. *See also* **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. The name is used in a subprogram definition. Argument keywords can be used when the subprogram is invoked to associate dummy arguments with actual arguments, so that the subprogram arguments can 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 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. *Contrast with* **scalar**. *See also* **bounds**, **conformable**, **shape**, **size**, **whole array**, and **zero-sized array**.

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 (individual) 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, or 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 statement

Usually, a statement that assigns (stores) the value of an expression on the right of an equal sign to the storage location of the variable to the left of the equal sign. In the case of Fortran 95/90 pointers, the storage location is assigned, not the pointer itself.

association

The relationship that allows an entity to be referenced by different names in one scoping unit or by the same or different names in more than one scoping unit. The principal kinds of association are argument, host, pointer, storage, and use association. *See also* **argument association**, **host association**, **pointer association**, **storage association**, and **use association**.

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. The rank of the array is the number of colons (:) specified in parentheses.

assumed-size array

A dummy array whose size (only) is assumed from its associated actual argument. The upper bound of its last dimension is specified by an asterisk (*). All other extents (if any) must be specified.

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.

Most attributes can be alternatively specified in statements. For example, the DIMENSION statement has the same meaning as the DIMENSION attribute appearing in a type declaration statement.

automatic array

An explicit-shape array that is a local variable in a subprogram. It is not a dummy argument, and has bounds that are nonconstant specification expressions. The bounds (and shape) are determined at entry to the procedure by evaluating the bounds expressions. *See also* **automatic object**.

automatic object

A local data object that is created upon entry to a subprogram and disappears when the execution of the subprogram is completed. There are two kinds of automatic objects: arrays (of any data type) and objects of type CHARACTER. Automatic objects cannot be saved or initialized.

An automatic object is not a dummy argument, but is declared with a specification expression that is not a constant expression. The specification expression can be the bounds of the array or the length of the character object.

B

background process

On Linux systems, a process for which the command interpreter is not waiting. Its process group differs from that of its controlling terminal, so it is blocked from most terminal access. *Contrast with* **foreground process**.

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. The functions IBSET, IBCLR, BTEST, and IBITS, and the subroutine MVBITS operate on bit fields.

blank common

A common block (one or more contiguous areas of storage) without a name. Common blocks are defined by a COMMON statement.

block

In general, a group of related items treated as a physical 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.

bottleneck

The slowest process in an executing program. This process determines the maximum speed of execution.

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**.

breakpoint

A critical point in a program, at which execution is stopped so that you can see if the program variables contain the correct values. Breakpoints are often used to debug programs.

built-in procedure

See **intrinsic procedure**.

byte

A group of 8 contiguous bits (binary digits) starting on an addressable boundary.

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 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 initiated by another process (the parent). The child process can operate independently from the parent process. Also, the parent process can suspend or terminate without affecting the child process. *See also* **parent 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. *See also* **blank common** and **named common block**.

compilation unit

The source file or files that are compiled together to form a single object file, possibly using interprocedural optimization across source files.

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".)

compiler option

An option (or flag) that can be used on the compiler command line to override the default behavior of the Intel® Fortran compiler.

complex constant

A constant that is a pair of real or integer constants representing a complex number; the pair is separated by a comma and enclosed in parentheses. The first constant represents the real part of the number; the second constant represents the imaginary part. The following types of complex constants are available on all systems: COMPLEX(KIND=4), COMPLEX(KIND=8), and COMPLEX(KIND=16).

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

Part of a derived-type definition. There must be at least one component (intrinsic or derived type) in every derived-type definition.

concatenate

The combination of two items into one by placing one of the items after the other. In Fortran 95/90, the concatenation operator (//) is used to combine character items. *See also* **character expression**.

conformable

Pertains to dimensionality. Two arrays are conformable if they have the same shape. A scalar is conformable with any array.

conformance

See **shape conformance**.

conservative automatic inlining

The inline expansion of small procedures, with conservative heuristics to limit extra code.

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 series of statements starting with a DO, SELECT CASE, IF, WHERE, or FORALL statement and ending with a corresponding terminal 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 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).

D

data abstraction

A style of programming in which you define types to represent objects in your program, define a set of operations for objects of each type, and restrict the operations to only this set, making the types abstract. The Fortran 95/90 modules, derived types, and defined operators, support this programming paradigm.

data edit descriptor

A repeatable format descriptor that causes the transfer or conversion of data to or from its internal representation. In FORTRAN 77, this term was called a field descriptor.

data entity

A data object that has a data type. It is the result of the evaluation of an expression, or the result of the execution of a function reference (the function result).

data item

A unit of data (or value) to be processed. Includes constants, variables, arrays, character substrings, or records.

data object

A constant, variable, or subobject (part) 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 declaration

See **type declaration statement**.

data type length specifier

The form *n appended to Intel Fortran-specific data type names. For example, in REAL*4, the *4 is the data type length specifier.

declaration

See **specification statement**.

default character

The kind for character constants if no kind type parameter is specified. Currently, the only kind parameter for character constants is CHARACTER(1), the default character kind.

default complex

The kind for complex constants if no kind type parameter is specified. The default complex kind is affected by compiler options that specify real size. If no compiler option is specified, default complex is COMPLEX(4) (COMPLEX*8). *See also* **default real**.

default integer

The kind for integer constants if no kind type parameter is specified. The default integer kind is affected by compiler options that specify integer size. If no compiler option is specified, default integer is INTEGER(4) (INTEGER*4).

If a compiler 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 of the integer is the smallest integer kind which can hold the integer.

default logical

The kind for logical constants if no kind type parameter is specified. The default logical kind is affected by compiler options that specify integer size. If no compiler option is specified, default logical is LOGICAL(4) (LOGICAL*4). *See also* **default integer**.

default real

The kind for real constants if no kind type parameter is specified. The default real kind is determined by the compiler option specifying real size. If no compiler option is specified, default real is REAL(4) (REAL*4).

If a real constant is encountered that is outside the range for the default, an error occurs.

deferred-shape array

An array pointer (an array with the POINTER attribute) or an allocatable array (an array with the ALLOCATABLE attribute). The size in each dimension is determined by pointer assignment or when the array is allocated.

The array specification contains a colon (:) for each dimension of the array. No bounds are specified.

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.

defined

For a data object, the property of having or being given a valid value.

defined assignment

An assignment statement that is not intrinsic, but is defined by a subroutine and an ASSIGNMENT(=) interface block. *See also* **derived type** and **interface block**.

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* **derived type** and **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 a data object) that can be defined and referenced separately from other parts of the data 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. *See also* **extent**.

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**.

directive

See **compiler directive**.

DLL

See **Dynamic Link Library**.

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's guide. *See also* **denormalized number**.

driver program

A program that is the user interface to the language compiler. It accepts command 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. *See also* **actual argument**.

dummy array

A dummy argument that is an array.

dummy procedure

A dummy argument that is specified as a procedure or appears in a procedure reference. The corresponding actual argument must be a procedure.

Dynamic Link Library (DLL)

A separate source module compiled and linked independently of the applications that use it. Applications access the DLL through procedure calls. The code for a DLL is not included in the user's executable image, but the compiler automatically modifies the executable image to point to DLL procedures at run time.

E

edit descriptor

A descriptor in a format specification. It can be a data edit descriptor, control edit descriptor, or string edit descriptor. *See also* **control edit descriptor**, **data edit descriptor**, and **string edit descriptor**.

element

See **array element**.

elemental

Pertains to an intrinsic operation, intrinsic procedure, or assignment statement that is independently applied to either of the following:

- The elements of an array
- Corresponding elements of a set of conformable arrays and scalars

end-of-file

The condition that exists when all records in a file open for sequential access have been read.

entity

A general term referring to any Fortran 95/90 concept; for example, a constant, a variable, a program unit, a statement label, a common block, a construct, an I/O unit and so forth.

environment variable

A symbolic variable that represents some element of the operating system, such as a path, a filename, or other literal data.

error number

An integer value denoting an I/O error condition, obtained by using the IOSTAT specifier in an I/O statement.

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

A data reference or a computation formed from operators, operands, and parentheses. The result of an expression is either a scalar value or an array of scalar values.

extension

See **language extension**.

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 other subprogram. A module is not a subprogram.

F

field

Can be either of the following:

- A set of contiguous characters, considered as a single item, in a record or line.
- A substructure of a STRUCTURE declaration.

field descriptor

See **data edit descriptor**.

field separator

The comma (,) or slash (/) that separates edit descriptors in a format specification.

field width

The total number of characters in the field. *See also* **field**, the first definition.

file

A collection of logically related records. If the file is in internal storage, it is an internal file; if the file is on an input /output device, it is an external file.

file access

The way records are accessed (and stored) in a file. The Fortran 95/90 file access modes are sequential and direct.

file handle

A unique identifier that the system assigns to a file when the file is opened or created. A file handle is valid until the file is closed.

file organization

The way records in a file are physically arranged on a storage device. Fortran 95/90 files can have sequential or relative organization.

fixed-length record type

A file format in which all the records are the same length.

foreground process

On Linux systems, a process for which the command interpreter is waiting. Its process group is the same as that of its controlling terminal, so the process is allowed to read from or write to the terminal. *Contrast with* **background process**.

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**.

full pathname

See **absolute pathname**.

function

A series of statements that perform some operation and return a single value (through the function or result name) to the calling program unit. A function is invoked by a function reference in a main program unit or a subprogram unit.

In Fortran 95/90, a function can be used to define a new operator or extend the meaning of an intrinsic operator symbol. The function is invoked by the appearance of the new or extended operator in the expression (along with the appropriate operands). For example, the symbol * can be defined for logical operands, extending its intrinsic definition for numeric operands. *See also* **function subprogram**, **statement function**, and **subroutine**.

function reference

Used in an expression to invoke a function, it consists of the function name and its actual arguments. A function reference returns a value (through the function or result name) that is used to evaluate the calling expression.

function result

The result value associated with a particular execution or call to a function. This result can be of any data type (including derived type) and can be array-valued. In a FUNCTION statement, the RESULT option can be used to give the result a name different from the function name. This option is required for a recursive function that directly calls itself.

function subprogram

A sequence of statements beginning with a FUNCTION (or optional OPTIONS) statement that is not in an interface block and ending with the corresponding END statement. *See also* **function**.

G

generic identifier

A generic name, operator, or assignment specified in an INTERFACE statement that is associated with all of the procedures within the interface block. Also called a generic specification.

global entity

An entity (a program unit, common block, or external procedure) that can be used with the same meaning throughout the executable program. A global entity has global scope; it is accessible throughout an executable program. *See also* **local entity**.

global section

A data structure (for example, global COMMON) or shareable image section potentially available to all processes in the system.

H

handle

A value (often, but not always, a 32-bit integer) that identifies some operating system resource, for example, a window or a process. The handle value is returned from an operating system call when the resource is created; your program then passes that value as an argument to subsequent operating system routines to identify which resource is being accessed.

Your program should consider the handle value a "private" type and not try to interpret it as having any specific meaning (for example, an address).

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.

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.

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. To provide the information needed to resolve the external references to DLL functions, the linker uses an import library when building an executable module of a process.

index

Can be either 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.

initialize

The assignment of an initial value to a variable.

initialization expression

A form of constant expression that is used to specify an initial value for an entity.

inlining

An optimization that replaces a subprogram reference (CALL statement or function invocation) with the replicated code of the subprogram.

input/output (I/O)

The data that a program reads or writes. Also, devices to read and write data.

inquiry function

An intrinsic function whose result depends on properties of the principal argument, not the value of the argument.

integer constant

A constant that is a whole number with no decimal point. It can have a leading sign and is interpreted as a decimal number.

intent

An attribute of a dummy argument that is not a pointer or procedure. It indicates whether the argument is used to transfer data into the procedure, out of the procedure, or both.

interactive process

A process that must periodically get user input to do its work. *Contrast with* **background process**.

interface

See **procedure interface**.

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) contained in a main program or another 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.

intrinsic

Describes entities defined by the Fortran 95/90 language (such as data types and procedures). Intrinsic entities can be used freely in any scoping unit.

intrinsic procedure

A subprogram supplied as part of the Fortran 95/90 library that performs array, mathematical, numeric, character, bit manipulation, and other miscellaneous functions. Intrinsic procedures are automatically available to any Fortran 95/90 program unit (unless specifically overridden by an EXTERNAL statement or a procedure interface block). Also called a built-in or library procedure.

invoke

To call upon; used especially with reference to subprograms. For example, to invoke a function is to execute the function.

I/O

See **input/output**.

iteration count

The number of executions of the DO range, which is determined as follows:

```
[(terminal value - initial value + increment value) / increment value]
```

K

keyword

See **argument keyword** and **statement keyword**.

kind type parameter

Indicates the range of an intrinsic data type; for example: INTEGER(KIND=2). For real and complex types, it also indicates precision. If a specific kind parameter is not specified, the kind is the default for that type (for example, default integer). *See also* **default character**, **default complex**, **default integer**, **default logical**, and **default real**.

L

label

An integer, from 1 to 5 digits long, that precedes a statement and identifies it. For example, labels can be used to refer to a FORMAT statement or branch target statement.

language extension

An Intel Fortran language element or interpretation that is not part of the Fortran 95 standard.

lexical token

A sequence of one or more characters that have an indivisible interpretation. A lexical token is the smallest meaningful unit (a basic language element) of a Fortran 95/90 statement; for example, constants and statement keywords.

library routines

Files that contain functions, subroutines, and data that can be used by Fortran programs.

For example: one library contains routines that handle the various differences between Fortran and C in argument passing and data types; another contains run-time functions and subroutines for Windows graphics and QuickWin applications.

Some library routines are intrinsic (automatically available) to Fortran; others may require a specific USE statement to access the module defining the routines. *See also* **intrinsic procedure**.

linker

A system program that creates an executable program from one or more object files (or modules) produced by a language compiler or assembler. The linker resolves external references, acquires referenced library routines, and performs other processing required to create Linux and Windows 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; its value is directly specified in a program. *See also* **named 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 Intel 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

A level of optimization enabling optimizations within the source program unit and recognition of common expressions. *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.

lower bound

See **bounds**.

M

main program

The first program unit to receive control when a program is run; it exercises control over subprograms. The main program usually contains a PROGRAM statement (or does not contain a SUBROUTINE, FUNCTION, or BLOCK DATA statement). *Contrast with* **subprogram**.

makefile

On Linux systems, an argument to the `make` command containing a sequence of entries that specify dependencies. On Windows systems, a file passed to the NMAKE utility containing a sequence of entries that specify dependencies. The contents of a makefile override the system built-in rules for maintaining, updating, and regenerating groups of programs.

For more information on makefiles on Linux systems, see `make(1)`. For more information on makefiles on Windows systems, see the online help on the NMAKE utility.

many-one array section

An array section with a vector subscript having two or more elements with the same value.

master thread

In an OpenMP Fortran program, the thread that creates a team of threads when a parallel region (PARALLEL directive construct) is encountered. The statements in the parallel region are then executed in parallel by each thread in the team. At the end of the parallel region, the team threads synchronize and only the master thread continues execution. *See also* **thread**.

message file

A Linux catalog that contains the diagnostic message text of errors that can occur during program execution (run time).

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 that is not an internal procedure and is contained in a module. 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.

multitasking

The ability of an operating system to execute several programs (tasks) at once.

multithreading

The ability of an operating system to execute different parts of a program, called threads, simultaneously.

If the system supports parallel processing, multiple processors may be used to execute the threads.

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).

A name can contain letters, digits, underscores (_), and the dollar sign ($) special character. The first character must be a letter or a dollar sign. In FORTRAN 77, this term was called a symbolic name.

name association

Pertains to argument, host, or use association. *See also* **argument association**, **host association**, and **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. *See also* **literal 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**.

NaN

Not-a-Number. The condition that results from a floating-point operation that has no mathematical meaning; for example, zero divided by zero.

natural boundary

The virtual address of a data item that is the multiple of the size of its data type. For example, a REAL(8) (REAL*8) data item aligned on natural boundaries has an address that is a multiple of eight.

naturally aligned record

A record that is aligned on a hardware-specific natural boundary; each field is naturally aligned. (For more information, see your user's guide.) *Contrast with* **packed record**.

nesting

The placing of one entity (such as a construct, subprogram, format specification, or loop) inside another entity of the same kind. For example, nesting a loop within another loop (a nested loop), or nesting a subroutine within another subroutine (a nested subroutine).

nonexecutable statement

A Fortran 95/90 statement that describes program attributes, but does not cause any action to be taken when the program is executed.

numeric expression

A numeric constant, variable, or function value, or combination of these, joined by numeric operators and parentheses, so that the entire expression can be evaluated to produce a single numeric value. For example, `-L` or `X + (Y - 4.5)*Z`.

numeric operator

A symbol designating an arithmetic operation. In Fortran 95/90, the symbols +, –, *, /, and ** are used to designate addition, subtraction, multiplication, division, and exponentiation, respectively.

numeric storage unit

The unit of storage for holding a non-pointer scalar value of type default real, default integer, or default logical. One numeric storage unit corresponds to 4 bytes of memory.

O

object

See **data object**.

object file

The binary output of a language processor (such as the assembler or compiler), which can either be executed or used as input to the linker.

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. For example, in `I .NE. J`, .NE. is the operator. *Contrast with* **operand**.

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). This kind of argument does not have to be associated with an actual argument when its procedure is invoked.

order of subscript progression

A characteristic of a multidimensional array in which the leftmost subscripts vary most rapidly.

overflow

An error condition occurring when an arithmetic operation yields a result that is larger than the maximum value in the range of a data type.

P

packed record

A record that starts on an arbitrary byte boundary; each field starts in the next unused byte. *Contrast with* **naturally aligned record**.

pad

The filling of unused positions in a field or character string with dummy data (such as zeros or blanks).

parallel processing

The simultaneous use of more than one processor (CPU) to execute a program.

parameter

Can be either of the following:

- In general, any quantity of interest in a given situation; often used in place of the term "argument".
- A Fortran 95/90 named constant.

parent process

A process that initiates and controls another process (child). The parent process defines the environment for the child process. Also, the parent process can suspend or terminate without affecting the child process. *See also* **child process**.

pathname

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, Windows 2000 on IA-32 processors).

pointer

Is one of the following:

- A Fortran 95/90 pointer

  A data object that has the POINTER attribute. A Fortran 95/90 pointer does not contain data, but *points* to a scalar or array variable where data is stored. 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**.
- An integer pointer

  A data object that contains the address of its paired variable.

pointer association

The association of storage space to a Fortran 95/90 pointer by means of a target. A pointer is associated with a target after pointer assignment or the valid execution of an ALLOCATE statement.

precision

The number of significant digits in a real number. *See also* **double-precision constant**, **kind type parameter**, and **single-precision constant**.

primary

The simplest form of an expression. A primary can be any of the following data objects:

- A constant

- A constant subobject (parent is a constant)
- A variable (scalar, structure, array, or pointer; an array cannot be assumed size)
- An array constructor
- A structure constructor
- A function reference
- An expression in parentheses

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. *See also* **subprogram**.

procedure interface

The statements that specify the name and characteristics of a procedure, the name and characteristics of each dummy argument, and the generic identifier (if any) by which the procedure can be referenced. The characteristics of a procedure are fixed, but the remainder of the interface can change in different scoping units.

If these properties are all known within the scope of the calling program, the procedure interface is explicit; otherwise it is implicit (deduced from its reference and declaration).

program

A set of instructions that can be compiled and executed by itself. Program blocks contain a declaration and an executable section.

program section

A particular common block or local data area for a particular routine containing equivalence groups.

program unit

The fundamental component of an executable program. A sequence of statements and optional comments that can be a main program, a procedure, an external program, or a block data program unit.

Q

quadword

Four contiguous words (64 bits) starting on any addressable byte boundary. Bits are numbered 0 to 63. (Bit 63 is used as the sign bit.) A quadword is identified by the address of the word containing the low-order bit (bit 0). The value of a signed quadword integer is in the range –2**63 to 2**63–1.

R

random access

See **direct access**.

rank

The number of dimensions in 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(KIND=4)), double precision (REAL(KIND=8)), or quad precision (REAL(KIND=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 = =, /=, <, <=, >, and >= (.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. Intel Fortran 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 Linux systems, the top-level directory in the file system; it is represented by a slash (/).

On Windows 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.

routine

A subprogram; a function or procedure. *See also* **function**, **subroutine**, and **procedure**.

run time

The time during which a computer executes the statements of a program.

S

saved object

A variable that retains its association status, allocation status, definition status, and value after execution of a RETURN or END statement in the scoping unit containing the declaration.

scalar

Pertaining to data items with a rank of zero. A single data object of any intrinsic or derived data type. *Contrast with* **array**. *See also* **rank-one object**.

scalar memory reference

A reference to a scalar variable, scalar record field, or array element that resolves into a single data item (having a data type) and can be assigned a value with an assignment statement. It is similar to a scalar reference, but it excludes constants, character substrings, and expressions.

scalar reference

A reference to a scalar variable, scalar record field, derived-type component, array element, constant, character substring, or expression that resolves into a single data item having a data type. *Contrast with* **scalar memory reference**.

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 cannot overlap, though one scoping unit can contain another scoping unit. (The outer scoping unit is called the host scoping unit.)

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's guide. *See also* **denormalized number**.

size

The total number of elements in an array; the product of the extents.

source file

A program or portion of a program library, such as an object file, or image file.

specification expression

A restricted expression that is of type integer and has a scalar value. This type of expression appears only in the declaration of array bounds and character lengths.

specification statement

A nonexecutable statement that provides information about the data used in the source program. Such a statement can be used to allocate and initialize variables, arrays, records, and structures, and define other characteristics of names used in a program.

statement

An instruction in a programming language that represents a step in a sequence of actions or a set of declarations. In Fortran 95/90, an ampersand (&) can be used to continue a statement from one line to another, and a semicolon (;) can be used to separate several statements on one line.

There are two main classes of statements: executable and nonexecutable. *See also* **executable statement** and **nonexecutable statement**.

statement function

A computing procedure defined by a single statement in the same program unit in which the procedure is referenced.

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. *See also* **statement function**.

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 that can optionally be 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 function or subroutine that can be invoked from another program unit to perform a specific task. A subprogram can define more than one procedure if it contains an ENTRY statement. *Contrast with* **main program**. *See also* **procedure**.

subroutine

A procedure that can return many values, a single value, or no value to the calling program unit (through arguments). A subroutine is invoked by a CALL statement in another program unit.

In Fortran 95/90, a subroutine can also be used to define a new form of assignment (defined assignment), which is different from those intrinsic to Fortran 95/90. Such assignments are invoked with an ASSIGNMENT(=) interface block rather than the CALL statement. *See also* **function**, **statement function**, and **subroutine subprogram**.

subroutine subprogram

A sequence of statements starting with a SUBROUTINE (or optional OPTIONS) statement and ending with the corresponding END statement. *See also* **subroutine**.

subscript

A scalar integer expression (enclosed in parentheses and appended to the array name) indicating the position of an array element. The number of subscripts is the rank of the array. *See also* **array element**.

subscript triplet

An item in a section subscript list specifying a range of values for the array section. A subscript triplet contains at least one colon and has three optional parts: a lower bound, an upper bound, and a stride. *Contrast with* **vector subscript**. *See also* **array section** and **section subscript**.

substring

A contiguous portion of a scalar character string. Do not confuse this with the substring selector in an array section, where the result is another array section, not a substring.

symbolic name

See **name**.

syntax

The formal structure of a statement or command string.

T

target

The named data object associated with a pointer (in the form pointer-object => target). A target is specified in a TARGET statement or in a type declaration statement that contains the TARGET attribute. *See also* **pointer** and **pointer association**.

thread

Part of a program that can run at the same time as other parts, usually with some form of communication and/or synchronization among the threads. *See also* **multithreading**.

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

See **data type**.

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. In Fortran 95/90, a type declaration statement may also specify attributes for the variables. Also called a type declaration or type specification.

type parameter

Defines an intrinsic data type. The type parameters are kind and length. The kind type parameter (KIND=) specifies the range for the integer data type, the precision and range for real and complex data types, and the machine representation method for the character and logical data types. The length type parameter (LEN=) specifies the length of a character string. *See also* **kind type parameter**.

U

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)`.

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 other scalar object of non-default intrinsic type.

upper bound

See **bounds**.

use association

The process by which the entities in a module are made accessible to other scoping units. This association is specified by a USE statement in the scoping unit. *See also* **module**.

user-defined assignment

See **defined assignment**.

user-defined operator

See **defined operation**.

user-defined type

See **derived type**.

V

variable

A data object (stored in a memory location) whose value can change during program execution. A variable can be a named data object, an array element, an array section, a structure component, or a substring. *Contrast with* **constant**.

variable format expression

A numeric expression enclosed in angle brackets (< >) that can be used in a FORMAT statement. If necessary, it is converted to integer type before use.

variable-length record type

A file format in which records may be of different lengths.

vector subscript

A rank-one array of integer values used as a section subscript to select elements from a parent array. Unlike a subscript triplet, a vector subscript specifies values (within the declared bounds for the dimension) in an arbitrary order. *Contrast with* **subscript triplet**. *See also* **array section** and **section subscript**.

W

whole array

An array reference (for example, in a type declaration statement) that consists of the array name alone, without subscript notation. Whole array operations affect every element in the array. *See also* **array**.

Z

zero-sized array

An array with at least one dimension that has at least one extent of zero. A zero-sized array has a size of zero and contains no elements. *See also* **array**.

Index

Symbols

!
See Exclamation point character
!$OMP prefix, 14-1
!DEC$ prefix, 14-1
!DIR$ prefix, 14-1
!MS$ prefix, 14-1
"
See Quotation mark character
%
See Built-in functions
%FILL built-in function
using in record structure, B-18
%LOC built-in function, 8-44
%REF built-in function, 8-43
See also your user's guide
%VAL built-in function, 8-43
See also your user's guide
&
See Ampersand character
(/.../)
See Array constructors
*
See Asterisk character
**
See Exponential operator
+
See Addition operator
–
See Subtraction operator
.AND., 4-8
.EQ., 4-7
See also your user's guide
.EQV., 4-8
.GE., 4-7
See also your user's guide
.GT., 4-7
See also your user's guide
.LE., 4-7
See also your user's guide
.LT., 4-7
See also your user's guide
.NE., 4-7
See also your user's guide
.NEQV., 4-8
.NOT., 4-8
See also your user's guide
.OR., 4-8
.XOR., 4-8
/
See Slash character
//
See also Concatenation operator
See Blank common blocks
/=
as relational operator, 4-7
::
See also Type declaration statement, 5-2

See Double colon separator, 5-2
<
as relational operator, 4-7
<=
as relational operator, 4-7
==
as relational operator, 4-7
=>
See Pointer assignment statements
>
as relational operator, 4-7
>=
as relational operator, 4-7
?
See Question mark character
[...]
See Array constructors
\
See Backslash character
'
See Apostrophe character

A

A edit descriptor, 11-26
input processing, 11-26
output processing, 11-27
ABS function, 9-18
Absolute spacing
function returning, 9-138
Absolute value
function computing, 9-18
function returning, 9-134
ACCEPT statements, 10-28
Access
modes of record, 10-2
Access methods, 10-2
ACCESS specifier
in INQUIRE statements, 12-8
in OPEN statements, 12-24
Accessibility of modules, 5-51
ACHAR function, 9-19
ACOS function, 9-20
ACOSD function, 9-20
ACOSH function, 9-21
ACTION specifier
in INQUIRE statements, 12-9
in OPEN statements, 12-25
Actual arguments, 8-30
definition of, 8-30
function returning pointer to list of, 9-66
functions not allowed as, 9-2
passing to procedures, 9-2
using aggregate field references as, B-23
Addition operator, 4-2
precedence of, 4-11
See also Unary operators
Address
function allocating, 9-91
function returning, 9-32
subroutine freeing allocated, 9-62
subroutine prefetching data from, 9-103
Adjustable arrays, 5-12, 5-13
in RECORD statements, B-23
ADJUSTL function, 9-21
ADJUSTR function, 9-22
ADVANCE specifier, 10-8
Advancing I/O, 10-8, 10-9
See also your user's guide
Aggregate assignment, B-24
example of, B-25
Aggregate field references, B-22
examples of, B-23
AIMAG function, 9-22
AIMAX0 function, 9-93
AIMIN0 function, 9-99
AINT function, 9-23
AJMAX0 function, 9-93
AJMIN0 function, 9-99
AKMAX0 function, 9-93
AKMIN0 function, 9-99
ALIAS
directive, 14-5
option for ATTRIBUTES directive, 14-8

ALIGN
option for ATTRIBUTES directive, 14-9
OPTIONS directive option, 14-27
PSECT directive option, 14-33
Alignment
See your user's guide
ALL function, 9-23
Allocatable arrays, 5-16, 5-17
allocation of, 6-3
deallocation of, 6-6
dynamically allocating and deallocating, 6-1
function to determine status of, 9-24
See also Arrays
ALLOCATABLE attribute and statement, 5-17
attributes compatible with, 5-5
examples of, 5-18
ALLOCATABLE option for ATTRIBUTES directive, 14-9
ALLOCATE statement, 6-2
examples of, 6-2, 6-4
ALLOCATED function, 6-3, 9-24
Allocating virtual memory
allocatable array, 6-4
Allocation
of allocatable arrays, 6-2, 6-3
of pointer targets, 6-2, 6-4
ALLOW_NULL
option for ATTRIBUTES directive, 14-9
ALOG function, 9-89
ALOG10 function, 9-90
Alphabetic characters
case sensitivity for, 2-6
Alternate entry points, 8-53
Alternate return
alternative for, A-2
arguments, 8-37
examples of, 7-34
specifier, 7-7, 8-25, 8-37
AMAX0 function, 9-93
AMAX1 function, 9-93
AMIN0 function, 9-99
AMIN1 function, 9-99
AMOD function, 9-105
Ampersand character (&)
as continuation indicator in free source form, 2-10
AND function, 9-65
ANINT function, 9-25
ANSI standard
conformance to, 1-2
language extensions to, F-1
ANY function, 9-25
Apostrophe character (')
as delimiter for character strings, 3-15
See also Character constants
See also Character strings
APOSTROPHE value
for INQUIRE (DELIM), 12-12
for OPEN (DELIM), 12-31
Append access
specifying for sequential files, 12-24
APPEND value
for INQUIRE (POSITION), 12-16
for OPEN (ACCESS), 12-25
for OPEN (POSITION), 12-35
Arccosine
function returning hyperbolic, 9-21
function returning in degrees, 9-20
function returning in radians, 9-20
Arcsine
function returning hyperbolic, 9-27
function returning in degrees, 9-27
function returning in radians, 9-26
Arctangent
function returning hyperbolic, 9-32
function returning in degrees, 9-31
function returning in degrees (complex), 9-31
function returning in radians, 9-29
function returning in radians (complex), 9-30
Argument association, 8-30, 15-10
Argument intent, 5-41
Argument keywords
argument association using, 8-31
BACK, 9-3
DIM, 9-3
in function references, 8-23

in intrinsic procedures, 9-3
in subroutine references, 7-7
KIND, 9-3
MASK, 9-3

Argument keywords in procedures
scope of, 15-2

Argument passing
defaults for, 8-43
using %VAL and %REF, 8-43

Argument presence function, 9-115

Argument-list functions, 8-43
See also Built-in functions

Arguments
actual, 8-30
aggregate field references as, B-23
alternate return, 8-37
array, 8-33
associating array elements with, 15-16
association of procedure, 8-30
assumed-length character, 8-35
assumed-shape, 8-33
character constants as, 8-36
defaults for %VAL and %REF functions, 8-43
dummy, 8-30, 8-37
Hollerith constants as, 8-36
intent of, 5-41
list of
effect in CALL statement, 7-8
of generic intrinsic functions, 9-2
optional, 8-32
pointer, 8-34
See also your user's guide
subroutine returning command-line, 9-62
using external and dummy procedures as, 5-38
using intrinsic procedures as, 5-43

Arithmetic expressions, 4-2

Arithmetic IF statement, 7-6
alternative for, A-2

Arithmetic shift
function performing left, 9-52, 9-133
function performing left or right, 9-77
function performing right, 9-52, 9-133

Array assignment statements, 4-20

Array assignments
masking in, 4-23

Array components, 3-23
examples of, 3-25

Array constructors, 3-44
implied-do loops in, 3-45

Array declaration statements, 5-10

Array declarators, 5-10

Array descriptor
data items passing, 5-14, 5-16, 5-50

Array elements, 3-35, 3-38
association of, 15-15
function returning location of maximum, 9-95
function returning location of minimum, 9-100
function returning maximum value of, 9-97
function returning minimum value of, 9-102
function returning product of, 9-116
function returning sum of, 9-140
order of, 3-39
references to, 3-39
storage of, 3-39

Array expressions, 4-20

Array functions
categories of, 9-4
for construction, 9-98, 9-113, 9-138, 9-148
for inquiry, 9-24, 9-83, 9-132, 9-136, 9-147
for location, 9-95, 9-100
for manipulation, 9-42, 9-54, 9-127, 9-146
for reduction, 9-23, 9-25, 9-40, 9-97, 9-102, 9-116, 9-140

Array name, 3-38
operations on, 3-35
unsubscripted in a DATA statement, 5-25

Array pointers, 5-16

Array properties, 3-35

Array sections, 3-35, 3-41
assigning values to, 4-20
many-one, 3-44, 4-20
restrictions to vector subscripts in, 3-44

Array specifications, 5-10

Array structure component, 3-25

Array transposition, 9-146

Array variables, 4-20

ARRAY_VISUALIZER (W*32)
    option for ATTRIBUTES directive, 14-10
Arrays, 3-35
    adjustable, 5-13
    allocatable, 5-16, 5-17
        example of, 6-4
    allocating allocatable, 6-2, 6-3
    as automatic objects, 5-12
    as components in derived types, 3-21
    as operands in expressions, 4-2
    as structure components, 3-25
    as variables, 3-33
    assignment of, 4-20
    associating group name with, 5-45
    association of, 15-15
    assumed-shape, 5-14
    assumed-size, 5-15
    automatic, 5-12
    bounds of, 3-36
    components of, 3-38
    conformable, 3-36
    constructing multidimensional, 3-46
    constructing one-dimensional, 3-44
    data type of, 3-36
    deallocating allocatable, 6-5, 6-6
    declaration of
        using ALLOCATABLE, 5-17
        using COMMON, 5-22
        using DIMENSION, 5-27
        using POINTER, 5-50
        using TARGET, 5-55
        using type declaration, 5-3
    deferred-shape, 5-16
    defining constants for, 3-44
    dummy argument, 5-13
    element order in, 3-39
    elements in, 3-38
    establishing with subprogram references, 15-16
    explicit-shape, 5-11
    extending, 9-127, 9-138
    extent of, 3-36
    function combining, 9-98
    function counting true elements in MASK, 9-40
    function determining all true in, 9-23
    function determining any true in, 9-25
    function packing, 9-113
    function performing circular shift of, 9-42
    function performing dot-product multiplication of, 9-50
    function performing end-off shift on, 9-54
    function performing matrix multiplication of, 9-92
    function returning location of maximum value in, 9-95
    function returning location of minimum value in, 9-100
    function returning lower bounds of, 9-83
    function returning maximum value of elements in, 9-97
    function returning minimum value of elements in, 9-102
    function returning shape of, 9-132
    function returning size (extent) of, 9-136
    function returning upper bounds of, 9-147
    function to add a dimension to, 9-138
    function to determine allocation of, 9-24
    function to replicate, 9-138
    function to reshape, 9-127
    function transposing rank-two, 9-146
    function unpacking, 9-148
    functions for geometric location, 9-4
    functions to construct, 9-4
    functions to determine properties of, 9-4
    functions to manipulate, 9-4
    functions to reduce, 9-4
    in I/O lists, 10-9
    initializing elements with DATA statements, 5-24
    intrinsic assignment of, 4-20
    intrinsic functions for, 9-4
    making equivalent, 5-31
    many-one, 3-44, 4-20
    number of storage elements for, 5-28
    properties of, 3-35
    rank of, 3-36
    referencing, 4-2
    section subscript in, 3-41
    section subscript list in, 3-35
    sections in, 3-41
    shape of, 3-36
    size of, 3-36
    size of dummy, 5-15
    specifications for, 5-10
    specifying the values in, 3-44

storage of, 3-39
subscript list in, 3-35
subscript triplets in, 3-42
vector subscripts in, 3-43
volatile, 5-57
whole, 3-38
zero-size, 3-36, 5-12

ASCII character set (L*X), C-1

ASCII constants
assigned in DATA statements, 5-26

ASIN function, 9-26

ASIND function, 9-27

ASINH function, 9-27

ASIS value
for INQUIRE (POSITION), 12-16
for OPEN (POSITION), 12-35

ASSIGN statement, 7-4
alternative for, A-3
establishing assigned GO TO, 7-5
examples of, 7-4
See also your user's guide

Assigned FORMAT specifier
alternative for, A-3

Assigned GO TO statement, 7-5
alternative for, A-3
establishing labels for, 7-4
examples of, 7-6

Assignment
array, 4-20
character, 4-18
defined, 4-21
derived-type, 4-19
element array (FORALL), 4-26
generic, 8-51
intrinsic, 4-16
logical, 4-18
masked-array (WHERE), 4-23
generalization of, 4-26
numeric, 4-17
pointer, 4-22

ASSIGNMENT interface specifier
for subroutines, 4-21, 8-47, 8-51

Assignment statements, 4-15
kinds of, 4-15

Assignment symbol
scope of, 15-2

ASSOCIATED function, 6-5, 9-28

ASSOCIATEVARIABLE specifier
in OPEN statements, 12-25

Association, 15-9
argument, 8-30, 15-10
common, 5-21
equivalence, 5-29
of arrays, 15-15
pointer, 15-12
storage, 15-13
full, 15-15
partial, 15-15
use and host, 15-11

ASSUME
OPTIONS statement option, 13-3

Assumed-length character
arguments, 3-34, 8-35
example of, 8-35
See also your user's guide
functions, 8-19
alternative for, A-2

Assumed-shape
arguments, 8-33
requiring explicit interface, 8-46
arrays, 5-14

Assumed-size
arguments, 8-33
arrays, 5-15
and subscript triplets, 3-42
as whole array references, 5-15
example of, 5-16
in SIZE function, 9-136
in UBOUND function, 9-147
restrictions to using, 5-16

Asterisk character (*)
as comment line indicator, 2-11
as dummy argument, 8-35, 8-37
as multiplication operator, 4-2
precedence of, 4-11
using to specify assumed-length character function, 8-19

ATAN function, 9-29

ATAN2 function, 9-30
ATAN2D function, 9-31
ATAND function, 9-31
ATANH function, 9-32
ATOMIC directive, 14-52
  example of, 14-53
Attributes
  ALLOCATABLE, 5-17
  AUTOMATIC, 5-18
  DIMENSION, 5-27
  EXTERNAL, 5-38
  in type declaration statements, 5-3
  INTENT, 5-41
  INTRINSIC, 5-43
  OPTIONAL, 5-46
  PARAMETER, 5-48
  POINTER, 5-50
  PRIVATE, 5-51
  PUBLIC, 5-51
  SAVE, 5-54
  STATIC, 5-18
  summary of compatible, 5-5
  TARGET, 5-55
  VOLATILE, 5-57
ATTRIBUTES directive, 14-5
  ALIAS option, 14-8
  ALIGN option, 14-9
  ALLOCATABLE option, 14-9
  ALLOW_NULL option, 14-9
  ARRAY_VISUALIZER option (W*32), 14-10
  C option, 14-10
  DECORATE option, 14-12
  DEFAULT option, 14-12
  DLLEXPORT option (W*32, W*64), 14-13
  DLLIMPORT option (W*32, W*64), 14-13
  EXTERN option, 14-13
  FORCEINLINE option, 14-14
  IGNORE_LOC option, 14-13
  INLINE option, 14-14
  NO_ARG_CHECK option, 14-14
  NOINLINE option, 14-14
  NOMIXED_STR_LEN_ARG option, 14-15
  REFERENCE option, 14-15
  STDCALL option, 14-10
  VALUE option, 14-15
  VARYING option, 14-16
Automatic arrays, 5-12
AUTOMATIC attribute and statement, 5-18
  attributes compatible with, 5-5
  examples of, 5-20
Automatic objects
  array as, 5-12
  in character declarations, 5-8
Automatic variables, 5-18
Automation object routines (W*32), E-24

B

B edit descriptor, 11-11
  input processing, 11-11
  output processing, 11-11
BABS function, 9-18
BACK keyword
  in intrinsics, 9-3
Background process
  temporarily suspending, 7-32
Backslash character (\)
  as edit descriptor, 11-37
Backslash editing, 11-37
BACKSPACE statement, 12-2
  See also REWIND statement
BADDRESS function, 9-32
BARRIER directive, 14-53
  example of, 14-53
Base of model
  function returning, 9-120
BBCLR function, 9-67
BBITS function, 9-68
BBSET function, 9-69
BBTEST function, 9-33
BIAND function, 9-65
BIEOR function, 9-71
BIG_ENDIAN value
  for INQUIRE (CONVERT), 12-11
  for OPEN (CONVERT), 12-28
Binary constants, 3-28
  assigning with DATA statement, 5-25

data type assignments of, 3-31
examples of, 3-28

Binary digits
See Bits

Binary operations, 4-3
defined, 8-50

Binary operators
definition of, 4-3

Binary patterns
functions that shift, 9-16

BINARY specifier (W*32, W*64)
in INQUIRE statements, 12-9

Binary transfer of data
function performing, 9-145

BINARY value (W*32, W*64)
for INQUIRE (FORM), 12-13
for OPEN (FORM), 12-32

Binary values
transferring, 11-11

BIOR function, 9-76

Bit constants, 3-28

Bit fields, 9-16
function to extract, 9-68
functions operating on, 9-16
operating on general, 9-17
references to, 9-16
subroutine to copy, 9-107

Bit functions, 9-16
categories of, 9-4

Bit model, D-4

Bit patterns
function performing circular shift on, 9-80
function performing logical shift on, 9-79

Bit position ranges
for intrinsics operating on 1-bit fields, 9-17

Bit representation of integers
functions returning, 9-5

Bit size
determining, 9-4

Bit subfields
referencing, 9-16

BIT_SIZE function, 9-33

BITEST function, 9-33

Bits
function arithmetically shifting left, 9-52, 9-133
function arithmetically shifting left or right, 9-77
function arithmetically shifting right, 9-52, 9-133
function clearing to zero, 9-67
function logically shifting left or right, 9-81
function performing exclusive OR on, 9-71
function performing inclusive OR on, 9-76
function performing logical AND on, 9-65
function returning number of, 9-33
function reversing value of, 9-67
function rotating left or right, 9-78
function setting to 1, 9-69
function to extract sequences of, 9-68
function to test, 9-33
model for data, D-4

BIXOR function, 9-71

BJTEST function, 9-33

BKTEST function, 9-33

Blank characters
effect in character expressions, 4-7
effect in statement label fields, 2-8
in fixed and tab source form, 2-11
in free source form, 2-9

Blank common blocks, 5-21
See also Common blocks

Blank editing (BN,BZ), 11-33
BN, 11-34
BZ, 11-34

Blank padding, 11-29, 12-34

BLANK specifier
in INQUIRE statements, 12-9
in OPEN statements, 11-3, 12-26

Block data program unit, 2-1, 8-1, 8-10
effect of using DATA statement in, 8-11
forcing linker to search libraries, 5-38
in EXTERNAL statement, 8-11
statement declaring as external, 5-38

BLOCK DATA statement, 8-10
example of, 8-12

Block DO construct, 7-15
examples of, 7-16

Block IF statement, 7-26
See also IF construct
Blocks
contained in constructs, 7-1
DO loops in, 7-1
interface, 8-46
BLOCKSIZE specifier
in INQUIRE statements, 12-10
in OPEN statements, 12-26
interaction with BUFFERCOUNT, 12-27
BMOD function, 9-105
BMVBITS subroutine, 9-107
BN edit descriptor, 11-34
BNOT function, 9-111
Bounds
function returning lower, 9-83
function returning upper, 9-147
in an array, 3-36
Branch specifiers
in data transfer, 10-7
Branch statements, 7-2
Branch target statements
definition of, 7-2
in data transfer, 10-7
Branching
statements, 7-2
to END IF statement, 7-27
to SELECT CASE statement, 7-13
BSHFT function, 9-79
BSHFTC function, 9-80
BSIGN function, 9-134
BTEST function, 9-33
BUFFERCOUNT specifier
in OPEN statements, 12-26
BUFFERED specifier
in INQUIRE statements, 12-10
in OPEN statements, 12-27
Built-in functions, 8-43
%FILL, B-18
%LOC, 8-44
%REF, 8-43
%VAL, 8-43
See also Intrinsic procedures
See also Non-Fortran procedures
See also your user's guide
BYTE
data type, 3-3
in type declaration statements, 5-2, 5-6
BZ edit descriptor, 11-34

C

C character as comment line indicator, 2-11
C option for ATTRIBUTES directive, 14-10
C strings, 3-16
c$OMP ATOMIC directive, 14-52
c$OMP BARRIER directive, 14-53
c$OMP CRITICAL directive, 14-54
c$OMP DO directive, 14-55
c$OMP FLUSH directive, 14-59
c$OMP MASTER directive, 14-60
c$OMP ORDERED directive, 14-61
c$OMP PARALLEL directive, 14-62
c$OMP PARALLEL DO directive, 14-64
c$OMP PARALLEL SECTIONS directive, 14-66
c$OMP prefix, 14-1
c$OMP SECTION directive, 14-67
c$OMP SECTIONS directive, 14-67
c$OMP SINGLE directive, 14-68
c$OMP THREADPRIVATE directive, 14-69
CABS function, 9-18
Cache
function returning size of a level in memory, 9-34
subroutine prefetching data on, 9-103
CACHESIZE function (i64), 9-34
CALL statement, 7-7
examples of, 7-8
See also your user's guide
using to invoke a function, 7-8
Carriage control
characters, 11-42
editing, 11-42
CARRIAGECONTROL specifier
in INQUIRE statements, 12-10

in OPEN statements, 12-28
CASE construct
examples of, 7-13
CASE constructs, 7-9
flow of control in, 7-11
CASE DEFAULT statement, 7-11
Case index, 7-10
determining a match, 7-10
Case sensitivity, 2-6
See also your user's guide
Case values
examples of, 7-11
range of, 7-10
CCOS function, 9-38
CDABS function, 9-18
CDCOS function, 9-38
cDEC$ ALIAS directive, 14-5
cDEC$ ATTRIBUTES directive, 14-5
ALIAS option, 14-8
ALIGN option, 14-9
ALLOCATABLE option, 14-9
ALLOW_NULL option, 14-9
ARRAY_VISUALIZER option (W*32), 14-10
C option, 14-10
DECORATE option, 14-12
DEFAULT option, 14-12
DLLEXPORT option (W*32, W*64), 14-13
DLLIMPORT option (W*32, W*64), 14-13
EXTERN option, 14-13
FORCEINLINE option, 14-14
IGNORE_LOC option, 14-13
INLINE option, 14-14
NO_ARG_CHECK option, 14-14
NOINLINE option, 14-14
NOMIXED_STR_LEN_ARG option, 14-15
REFERENCE option, 14-15
STDCALL option, 14-10
VALUE option, 14-15
VARYING option, 14-16
cDEC$ DECLARE directive, 14-16
cDEC$ DEFINE directive, 14-17
cDEC$ DISTRIBUTE POINT directive, 14-18
cDEC$ ELSE directive, 14-20
cDEC$ ELSEIF directive, 14-20
cDEC$ ENDIF directive, 14-20
cDEC$ FIXEDFORMLINESIZE directive, 14-19
cDEC$ FREEFORM directive, 14-20
cDEC$ IDENT directive, 14-20
cDEC$ IF DEFINED directive, 14-20
cDEC$ IF directive, 14-20
cDEC$ INTEGER directive, 14-22
cDEC$ IVDEP directive, 14-23
cDEC$ LOOP COUNT directive, 14-25
cDEC$ MESSAGE directive, 14-26
cDEC$ NODECLARE directive, 14-16
cDEC$ NOFREEFORM directive, 14-20
cDEC$ NOPARALLEL directive, 14-31
cDEC$ NOPREFETCH directive, 14-32
cDEC$ NOSTRICT directive, 14-36
cDEC$ NOSWP directive (i64), 14-37
cDEC$ NOUNROLL directive, 14-39
cDEC$ NOVECTOR directive (i32), 14-40
cDEC$ OBJCOMMENT directive, 14-26
cDEC$ OPTIONS directive, 14-27
cDEC$ PACK directive, 14-30
cDEC$ PARALLEL directive, 14-31
cDEC$ PREFETCH directive, 14-32
cDEC$ prefix, 14-1
cDEC$ PSECT directive, 14-33
cDEC$ REAL directive, 14-35
cDEC$ STRICT directive, 14-36
cDEC$ SUBTITLE directive, 14-38
cDEC$ SWP directive (i64), 14-37
cDEC$ TITLE directive, 14-38
cDEC$ UNDEFINE directive, 14-17
cDEC$ UNROLL directive, 14-39
cDEC$ VECTOR ALIGNED directive (i32), 14-39
cDEC$ VECTOR ALWAYS directive (i32), 14-40
cDEC$ VECTOR NONTEMPORAL directive (i32), 14-41
cDEC$ VECTOR UNALIGNED directive (i32), 14-39
CDEXP function, 9-59

cDIR$ prefix, 14-1
CDLOG function, 9-89
CDSIN function, 9-135
CDSQRT function, 9-139
CDTAN function, 9-142
CEILING function, 9-35
CEXP function, 9-59
CHAR function, 9-35
Character
  arguments
    assumed-length, 8-35
    passed length of, 3-34
  expressions
    in relational expressions, 4-7, 4-8
  function returning, 9-19, 9-35
  function returning position of, 9-64, 9-69, 9-70
  function to check for all in set, 9-149
  operations, 4-6
  printable, 2-6
  See also Lowercase letters
  See also Uppercase letters
  substrings, 3-17
    making equivalent, 5-33
Character arguments
  assumed-length, 8-35
Character assignment statements, 4-18
Character constants, 3-15
  as arguments, 8-36
  as edit descriptors, 11-39
  assigned with DATA statements, 5-26
  C strings in, 3-16
  continuation in fixed and tab source, 2-12
  continuation in free source, 2-10
  default kind for, 3-15
  delimiters in, 3-15
  length of, 3-16
  See also your user's guide
  uppercase and lowercase letters in, 2-6
Character count
  editing, 11-38
  specifier, 10-9
CHARACTER data type, 3-14
  C strings, 3-16
  constants, 3-15
  conversion rules with DATA, 5-25
  default kind, 3-15
  definition of, 3-3
  in type declaration statements, 5-2, 5-8
  storage, 15-14
  storage requirements, 15-14
  substrings, 3-17
Character editing, 11-26
Character expressions, 4-6
  as format specifications, 11-4
  function returning length of, 9-85
  operator in, 4-6
Character functions
  categories of, 9-4
  definition of conversion, 9-4
  definition of string-handling, 9-4
Character operands
  comparing, 4-8
Character set
  ASCII (L*X), C-1
  extensions to Fortran 95, F-1
  Fortran 95/90, 2-5
CHARACTER statement, 5-8
Character storage unit, 15-14
Character string edit descriptors, 11-38 thru 11-40
Character strings
  as edit descriptors, 11-38, 11-39
  directive specifying for output, 14-26
  function adjusting to left, 9-21
  function adjusting to right, 9-22
  function returning length minus trailing blanks, 9-85
  function to check for all characters in, 9-149
  function to concatenate copies of, 9-126
  function to scan for characters in, 9-129
  function to trim blanks from, 9-147
  in list-directed records, 10-16
  in namelist records, 10-19
  of different lengths, 3-34
  without delimiters, 10-19, 10-27
Character substrings, 3-17
  effect of assigning values to, 4-19
  examples of, 3-18
  positions within parent string, 3-18

Character type declaration statements, 5-8
    automatic objects in, 5-8
Character type functions, 8-19
Character values
    transferring, 11-26
Character-oriented I/O, 10-9
CHECK
    OPTIONS statement option, 13-3
Circular shift
    function performing, 9-80
    of arrays
        function performing, 9-42
Clauses
    COPYIN, 14-44
    COPYPRIVATE, 14-44
    DEFAULT, 14-45
    FIRSTPRIVATE, 14-46
    IF, 14-62
    LASTPRIVATE, 14-46
    ORDERED, 14-56
    REDUCTION, 14-47
    SCHEDULE, 14-56
    SHARED, 14-49
CLOG function, 9-89
CLOSE statement, 12-3
CMPLX function, 9-36
cMS$ prefix, 14-1
Code access
    OpenMP directive restricting to one thread, 14-54
Code execution
    OpenMP directive dividing among threads, 14-67
    OpenMP directive restricting to master thread, 14-60
    OpenMP directive restricting to one thread in a team, 14-68
    OpenMP directive specifying sequential, 14-61
Colon character
    as edit descriptor, 11-37
    in array specification, 3-43, 3-46, 5-14, 5-16
Colon editing, 11-37
Column positions
    in fixed source form, 2-13
Columns
    for fields in fixed source form, 2-13
    position of comment indicator, 2-11
    position of debugging indicator, 2-12
COM object routines (W*32), E-23
Combining arrays, 9-98
Comma character
    as a field separator, 11-29
Command-line arguments
    function returning index of, 9-66
    function returning number of, 9-66, 9-108
    subroutine returning, 9-62
Comment indicators, 2-7
    in fixed and tab source form, 2-11
    in free source form, 2-10
Comments, 2-7
    allowable characters in, 2-6
    in continued statements, 2-7
    in namelist input, 10-20
Common block association, 5-21
Common blocks, 5-21
    agreement of data types in, 5-22
    arrays in, 5-22
    blank, 5-21
    data types of variables in, 5-22
    defaults and PSECT modifications, 14-34
    directive modifying alignment of data in, 14-27
    directive modifying characteristics of, 14-33
    effect of including in SAVE statement, 5-55
    effect of sharing names in, 5-22
    EQUIVALENCE interaction in, 5-35
    establishing and initializing values in, 8-10
    example of module containing, 8-9
    extending, 5-35
    initialized size in DATA, 5-23
    named, 5-21, 5-22, 8-10
    pointers in, 5-22
    scope of, 15-2
    variables in, 5-21
    volatile, 5-57
COMMON statement, 5-21
    examples of, 5-23
    interaction with EQUIVALENCE, 5-35
    using record structure names in, B-21
    using to define storage areas, 5-21
Comparison character functions, 9-4

Compatibility
features for language version, B-1
See also your user's guide
summary of language, 1-2
Compilation control statements, 13-1
Compiler directives, 14-1
affecting DO loops, 14-4
ALIAS, 14-5
ATOMIC, 14-52
ATTRIBUTES, 14-5
BARRIER, 14-53
CRITICAL, 14-54
DECLARE, 14-16
DEFINE, 14-17
DISTRIBUTE POINT, 14-18
DO, 14-55
FIXEDFORMLINESIZE, 14-19
FLUSH, 14-59
FREEFORM, 14-20
general, 14-2
syntax of prefix, 14-1
IDENT, 14-20
IF, 14-20
IF DEFINED, 14-20
INTEGER, 14-22
IVDEP, 14-23
LOOP COUNT, 14-25
MASTER, 14-60
MESSAGE, 14-26
NODECLARE, 14-16
NOFREEFORM, 14-20
NOPARALLEL loop, 14-31
NOPREFETCH, 14-32
NOSTRICT, 14-36
NOSWP (i64), 14-37
NOUNROLL, 14-39
NOVECTOR (i32), 14-40
OBJCOMMENT, 14-26
OpenMP Fortran, 14-42
syntax of prefix, 14-1
OPTIONS, 14-27
ORDERED, 14-61
PACK, 14-30
parallel
See OpenMP* Fortran compiler directives
PARALLEL DO, 14-64
PARALLEL loop, 14-31
PARALLEL OpenMP Fortran, 14-62
PARALLEL SECTIONS, 14-66
PREFETCH, 14-32
PSECT, 14-33
REAL, 14-35
SECTIONS, 14-67
See also General compiler directives
See also OpenMP* Fortran compiler directives
SINGLE, 14-68
STRICT, 14-36
SUBTITLE, 14-38
SWP (i64), 14-37
THREADPRIVATE, 14-69
TITLE, 14-38
UNDEFINE, 14-17
UNROLL, 14-39
VECTOR ALIGNED (i32), 14-39
VECTOR ALWAYS (i32), 14-40
VECTOR NONTEMPORAL (i32), 14-41
VECTOR UNALIGNED (i32), 14-39
Compiler limits
See your user's guide
Compiler options
overriding with OPTIONS statements, 13-3
See also Command line in your user's guide
Complementary error function
function returning, 9-57
Complex constants, 3-11
See also COMPLEX(16)
See also COMPLEX(4)
See also COMPLEX(8)
COMPLEX data type, 3-2, 3-10
constants, 3-11, 3-12, 3-13
default kind, 3-11
function converting to double-precision real, 9-46
function converting to quad-precision real, 9-117
in type declaration statements, 5-2, 5-6
See also COMPLEX(4)
storage, 15-14
Complex data types, 3-10 thru 3-14
Complex editing, 11-14, 11-24
input processing, 11-24
output processing, 11-24

Complex expressions
using relational operators in, 4-8
Complex numbers
function determining imaginary part of, 9-22
function resulting in conjugate of, 9-37
Complex operands
comparing, 4-8
Complex values
transferring, 11-14, 11-24
COMPLEX(16)
constants, 3-13
data type, 3-10
function converting to, 9-117
See also your user's guide
function converting to quad-precision real, 9-119
storage requirements, 15-14
COMPLEX(4)
constants, 3-11
data type, 3-10
See also your user's guide
See also COMPLEX data type
storage requirements, 15-14
COMPLEX(8)
constants, 3-12
data type, 3-10
See also your user's guide
See also DOUBLE COMPLEX data type
storage requirements, 15-14
COMPLEX*16 constants
See COMPLEX(8)
COMPLEX*32 constants
See COMPLEX(16)
COMPLEX*8 constants
See COMPLEX(4)
Component selector, 3-23
Components
arrays as derived-type, 3-21
derived-type, 3-19, 3-20
referencing, 3-23
of array structures, 3-38
Components of derived types
scope of, 15-2
Computation functions
definition of, 9-4
Computed GO TO statement, 7-3
alternative for, A-2
Concatenation of strings
function performing, 9-126
Concatenation operator (//)
precedence of, 4-11
See also your user's guide
using for long character constants
in fixed and tab source, 2-12
using in expressions, 4-6
Conditional compilation
general directive creating symbolic variable for, 14-17
general directive specifying, 14-20
specifying for OpenMP directives, 14-49
Conditional DO statement, 7-23
Conformable arrays, 3-36
CONJG function, 9-37
Conjugate
function calculating, 9-37
Conjunction
logical, 4-8
Connecting files, 12-20
Constant expressions, 4-11
Constants
array, 3-44
binary, 3-28
character, 3-15
complex, 3-11
definition of, 3-1
hexadecimal, 3-29
Hollerith, 3-30
in list-directed records, 10-16
in namelist records, 10-19
integer, 3-4
literal, 3-1
logical, 3-14
named, 3-1, 5-48
nondecimal numeric, 3-28
octal, 3-29
ranges for
See your user's guide
real, 3-7
Constructors
array, 3-44

structure, 3-26
examples of, 3-27
Constructs
CASE, 7-9
DO, 7-14
FORALL, 4-26
IF, 7-26
named, 7-1
nested, 7-29
See also Nested constructs
WHERE, 4-23
CONTAINS statement, 8-4, 8-29, 8-53
in main programs, 8-3
Continuation indicator, 2-7
in fixed source form, 2-11
in free source form, 2-10
in tab source form, 2-11
Continuation line
in debugging statements, 2-12
number allowed, 2-7
restriction in included files, 13-2
CONTINUE statement, 7-14, 7-16
Control characters
in printing, 11-42
Control constructs
blocks in, 7-1
named, 7-1
CASE, 7-9
DO, 7-15
FORALL, 4-26
IF, 7-26
WHERE, 4-23
Control edit descriptors, 11-30
for blanks, 11-33
forms for, 11-30
positional, 11-31
repeating, 11-40
sign, 11-33
Control lists
I/O, 10-3
See also I/O control list
Control statements, 7-1
extensions to, F-3
Control transfer
statements allowing, 7-1
with arithmetic IF statement, 7-6
with branch statements, 7-2
with CALL statement, 7-7
with CASE construct, 7-9
with DO construct, 7-14
with DO WHILE statement, 7-23
with END statement, 7-25
with GO TO statement
assigned, 7-5
computed, 7-3
unconditional, 7-2
with IF construct, 7-26
with logical IF statement, 7-31
with RETURN statement, 7-33
Control-list specifiers
defining variable for character count, 10-9
for advancing or nonadvancing I/O, 10-8
for transfer of control, 10-7
identifying the I/O status, 10-6
identifying the record number, 10-6
identifying the unit, 10-4
indicating the format, 10-5
indicating the namelist group, 10-6
keywords for, 10-3
mixed form, 10-4
nonkeyword form, 10-4
Conversion
function performing logical, 9-91
function performing real, 9-125
function resulting in complex type, 9-36
function resulting in COMPLEX(16) type, 9-117
function resulting in double-complex type, 9-47
function resulting in double-precision type, 9-46, 9-48, 9-49, 9-51
function resulting in integer type, 9-73, 9-82
function resulting in INTEGER(2) type, 9-76
function resulting in quad-precision type, 9-117, 9-118
function resulting in real type, 9-128
function resulting in REAL(16) type, 9-119
rules for numeric assignments, 4-17
to higher precision, 4-6
to nearest integer, 9-35, 9-60

Conversion character functions, 9-4
Conversion of data
    rules for numeric assignment statements, 4-17
    to or from internal representation, 11-6
    using FORMAT statements, 11-1
CONVERT
    OPTIONS statement option, 13-3
CONVERT specifier
    alternatives for, 12-29
    in INQUIRE statements, 12-11
    in OPEN statements, 12-28
        See also your user's guide
Converting unformatted numeric files, 12-11, 12-28
COPYIN clause, 14-44
    for THREADPRIVATE common blocks, 14-70
    in PARALLEL directive, 14-62
    in PARALLEL DO directive, 14-65
    in PARALLEL SECTIONS directive, 14-66
COPYPRIVATE clause, 14-44
    in SINGLE directive, 14-68
COS function, 9-38
COSD function, 9-38
COSH function, 9-39
Cosine
    function returning hyperbolic, 9-39
    function with argument in degrees, 9-38
    function with argument in radians, 9-38
COTAN function, 9-39
COTAND function, 9-40
Cotangent
    function with argument in degrees, 9-40
    function with argument in radians, 9-39
COUNT function, 9-40
CPU_TIME subroutine, 9-42
CQABS function, 9-18
CQCOS function, 9-38
CQEXP function, 9-59
CQLOG function, 9-89
CQSIN function, 9-135
CQSQRT function, 9-139
CQTAN function, 9-142
CRAY value
    for INQUIRE (CONVERT), 12-11
    for OPEN (CONVERT), 12-28
CRAY*-style pointers
    See Integer pointers
CRITICAL directive, 14-54
    example of, 14-55
CSHIFT function, 9-42
CSIN function, 9-135
CSQRT function, 9-139
CTAN function, 9-142
Current date
    subroutines returning, 9-44, 9-71
Cycle
    beginning new one in DO constructs, 7-24
CYCLE statement, 7-14, 7-24

D

D character
    as debugging statement indicator, 2-12
D edit descriptor, 11-16
    input processing, 11-17
    output processing, 11-17
DABS function, 9-18
DACOS function, 9-20
DACOSD function, 9-20
DACOSH function, 9-21
DASIN function, 9-26
DASIND function, 9-27
DASINH function, 9-27
Data abstraction
    example of, 8-10
Data conversion
    rules for numeric assignment statements, 4-17
    to or from internal representation, 11-6
    using FORMAT statements, 11-1
Data edit descriptors, 11-6
    default field widths for, 11-28
    forms for, 11-6
    integer, 11-9
    real, 11-14

repeating, 11-7
rules for numeric, 11-8

Data editing
specifying format for, 10-5

Data objects
assigning initial values to, 5-24
associating with group name, 5-45
directive specifying properties of, 14-5
in common block
defining storage of, 5-21
providing initial values for, 8-10
retaining properties of, 5-54
See also Data in your user's guide
specifying as pointers, 5-50
storage association of, 5-29
unpredictable values of, 5-57

Data representation
bit model, D-4
integer model, D-2
real model, D-3

Data representation models, D-1
intrinsic functions providing data for, D-1

Data scope attribute clauses, 14-44
COPYIN, 14-44
COPYPRIVATE, 14-44
DEFAULT, 14-45
DEFAULT NONE, 14-45
DEFAULT PRIVATE, 14-45
DEFAULT SHARED, 14-45
FIRSTPRIVATE, 14-46
LASTPRIVATE, 14-46
PRIVATE, 14-46
REDUCTION, 14-47
SHARED, 14-49

DATA statement, 5-24
effect in block data program unit, 8-11
examples of, 5-26
implied-DO list in, 5-24
list of constants in, 5-24
See also your user's guide
unsubscripted array name in, 5-25
using to define arrays, 15-16

Data transfer
from direct-access files
input, 10-24
output, 10-35
from internal files
input, 10-26
output, 10-36
from sequential files
input, 10-13
output, 10-29
function for binary, 9-145

Data transfer statements, 10-2
ADVANCE specifier in, 10-8
branch specifiers in, 10-7
components of, 10-2
control list in, 10-3
control specifiers in, 10-2
FMT specifier in, 10-5
I/O list in, 10-9
implied-DO lists in, 10-12
input, 10-13
ACCEPT, 10-28
READ, 10-13
IOSTAT specifier in, 10-6
list items in, 10-10
NML specifier in, 10-6
output
PRINT and TYPE, 10-38
REWRITE, 10-39
WRITE, 10-29
REC specifier in, 10-6
SIZE specifier in, 10-9
UNIT specifier in, 10-4

Data translation
direct-access statements
READ, 10-25
REWRITE, 10-39
WRITE, 10-36
internal statements
READ, 10-26
WRITE, 10-36
sequential statements
ACCEPT, 10-28
PRINT and TYPE, 10-38
READ, 10-15
WRITE, 10-30

Data type declaration statements, 3-34, 5-2
See also Type declaration statements

Data types, 3-1 thru 3-33
  character
    conversion rules with DATA statement, 5-25
    kind parameter for, 3-15
  complex
    kind parameters for, 3-10
  conventions for determining
    in numeric expressions, 4-6
  conversion in numeric assignment statements, 4-17
  derived, 3-19
    defining, 3-20
  determining for expressions, 4-6
  determining in numeric expressions, 4-5
  examples of assigning, 3-5
  implicit, 3-35
  integer
    kind parameters for, 3-4
  logical
    kind parameters for, 3-14
  numeric
    conversion rules with DATA statement, 5-25
  of named constants, 5-48
  overriding default for names, 5-39
  ranking in numeric expressions, 4-5
  real
    kind parameters for, 3-6
  resulting from logical operations, 4-9
  specifying explicit, 3-34
  specifying for variables, 3-34
  storage requirements for, 15-14
DATAN function, 9-29
DATAN2 function, 9-30
DATAN2D function, 9-31
DATAND function, 9-31
DATANH function, 9-32
Date
  subroutines returning current, 9-44
  subroutines to return current, 9-44, 9-71
Date and time
  subroutine returning, 9-44
DATE subroutine, 9-44
DATE_AND_TIME subroutine, 9-44
DBLE function, 9-46
DBLEQ function, 9-46
DCMPLX function, 9-47
DCONJG function, 9-37
DCOS function, 9-38
DCOSD function, 9-38
DCOSH function, 9-39
DCOTAN function, 9-39
DCOTAND function, 9-40
DDIM function, 9-49
DEALLOCATE statement, 6-5
  examples of, 6-6
Deallocation
  of allocatable arrays, 6-5, 6-6
  of pointer targets, 6-5, 6-7
Debug statements, 2-12
Debugging
  directive specifying string for, 14-26
Decimal exponent
  function returning range of, 9-124
Decimal point
  moving in real and complex values, 11-34
Decimal precision
  function returning, 9-114
Declaration statements, 5-1
  See also Type declaration statements
Declarations, 5-1
  array, 5-10
  character type, 5-8
  derived-type, 5-10
  numeric and logical type, 5-6
  record structure, B-14
    nesting, B-15
  record substructure, B-18
  union, B-19
DECLARE directive, 14-16
DECODE statement, B-3
DECORATE
  option for ATTRIBUTES directive, 14-12
DEFAULT
  clause, 14-45
    in PARALLEL directive, 14-62
    in PARALLEL DO directive, 14-65
    in PARALLEL SECTIONS directive, 14-66

option for ATTRIBUTES directive, 14-12
Default initialization
of derived-type components, 3-22
DEFAULT NONE clause, 14-45
DEFAULT PRIVATE clause, 14-45
DEFAULT SHARED clause, 14-45
DEFAULTFILE specifier
in INQUIRE statements, 12-7
in OPEN statements, 12-30
Defaults
caused by implicit typing, 3-35
for accessibility of modules, 5-52
for argument passing, 8-43
for character constants
See "default character" in the Glossary
for complex constants
See "default complex" in the Glossary
for integer constants
See "default integer" in the Glossary
for interpretation of blanks, 11-8
for list-directed output, 10-31
for logical constants
See "default logical" in the Glossary
for names, 3-35
statement overriding, 5-39
for OPEN statement specifiers, 12-20
for real constants
See "default real" in the Glossary
widths for data edit descriptors, 11-28
Deferred-shape arrays, 5-16
DEFINE directive, 14-17
example of, 14-17
DEFINE FILE statement, B-1
compared to OPEN statement, B-1
Defined assignment, 4-21
intent of arguments in subroutines specifying, 5-42
scope of, 15-2
Defined operations, 4-10, 8-50
binary, 4-10
unary, 4-10
Defined operators, 4-10
intent of dummy arguments in functions specifying, 5-42
scope of, 15-2
Defined variables, 3-33
DELETE statement, 12-4
alternative form for relative files, B-10
examples of, 12-5
DELETE value
for CLOSE statements, 12-3
for OPEN (DISPOSE), 12-31
DELIM specifier
in INQUIRE statements, 12-12
in OPEN statements, 12-30
DENYNONE value (W*32, W*64)
for INQUIRE (SHARE), 12-19
for OPEN (SHARE), 12-38
DENYRD value (W*32, W*64)
for INQUIRE (SHARE), 12-19
for OPEN (SHARE), 12-38
DENYRW value (W*32, W*64)
for INQUIRE (SHARE), 12-19
for OPEN (SHARE), 12-38
DENYWR value (W*32, W*64)
for INQUIRE (SHARE), 12-19
for OPEN (SHARE), 12-38
Dependence analysis
directive assisting, 14-23
DERF function, 9-57
DERFC function, 9-57
Derived data types, 3-19
arrays as components of, 3-27
assignment statements, 4-19
default initialization of, 3-21, 3-22
defining, 3-20
directive specifying starting address of items in, 14-30
in I/O lists, 10-10
pointers as components of, 3-27
referencing, 4-2
referencing components in, 3-23
scope of, 15-2
scope of component, 3-21
scope of type, 3-21
sequence, 3-22
specifying scalar values of, 3-26
volatile objects of, 5-57
Derived types, 3-19
See also Derived data types

Derived-type assignment statements, 4-19
Derived-type components, 3-19, 3-20
    arrays as, 3-21
    attributes in, 3-21
    initialization in, 3-21
    referencing, 3-23
    statements in, 3-20
Derived-type declaration statements, 5-10
Derived-type definitions, 3-19, 3-20
    default initialization in, 3-21, 3-22
    examples of, 3-24
DEXP function, 9-59
DFAUTO routines (W*32), E-24
DFCOM routines (W*32), E-23
DFLOAT function, 9-48
DFLOTI function, 9-48
DFLOTJ function, 9-48
DFLOTK function, 9-48
Dialog routines (W*32), E-20
DIGITS function, 9-48
DIM function, 9-49
DIM keyword
    in intrinsics, 9-3
DIMAG function, 9-22
Dimension
    bounds in, 3-36
    definition of, 3-36
DIMENSION attribute and statement, 5-27
    attributes compatible with, 5-5
    examples of, 5-28
DIMENSION statement
    using record structure names in, B-21
DINT function, 9-23
Direct access
    definition of, 10-2
    READ statements, 10-24
        forms of, 10-24
    records
        deleting, 12-4
    See also your user's guide
    specifying, 12-24
    WRITE statements, 10-35
        forms of, 10-35
DIRECT specifier
    in INQUIRE statements, 12-12
DIRECT value
    for INQUIRE (ACCESS), 12-9
    for OPEN (ACCESS), 12-25
Directive enhanced compilation, 14-1
    See also Directives
Directives, 14-1
    affecting DO loops, 14-4
    ALIAS, 14-5
    ATOMIC, 14-52
    ATTRIBUTE, 14-5
    BARRIER, 14-53
    CRITICAL, 14-54
    DECLARE, 14-16
    DEFINE, 14-17
    DISTRIBUTE POINT, 14-18
    DO, 14-55
    FIXEDFORMLINESIZE, 14-19
    FLUSH, 14-59
    FREEFORM, 14-20
    general, 14-2
    IDENT, 14-20
    IF, 14-20
    IF DEFINED, 14-20
    INTEGER, 14-22
    IVDEP, 14-23
    LOOP COUNT, 14-25
    MASTER, 14-60
    MESSAGE, 14-26
    NODECLARE, 14-16
    NOFREEFORM, 14-20
    NOPARALLEL loop, 14-31
    NOPREFETCH, 14-32
    NOSTRICT, 14-36
    NOSWP (i64), 14-37
    NOUNROLL, 14-39
    NOVECTOR (i32), 14-40
    OBJCOMMENT, 14-26
    OpenMP Fortran
    OPTIONS, 14-27
    ORDERED, 14-61
    PACK, 14-30

PARALLEL DO, 14-64
PARALLEL loop, 14-31
PARALLEL OpenMP Fortran, 14-62
PARALLEL SECTIONS, 14-66
PREFETCH, 14-32
prefixes for, 14-1
PSECT, 14-33
REAL, 14-35
SECTIONS, 14-67
SINGLE, 14-68
STRICT, 14-36
SUBTITLE, 14-38
SWP (i64), 14-37
syntax rules for, 14-1
THREADPRIVATE, 14-69
TITLE, 14-38
UNDEFINE, 14-17
UNROLL, 14-39
VECTOR ALIGNED (i32), 14-39
VECTOR ALWAYS (i32), 14-40
VECTOR UNALIGNED (i32), 14-39

Disassociation
of pointers, 6-8

Disconnecting files, 12-3

Disjunction
logical, 4-8

DISPOSE specifier
in OPEN statements, 12-31

DISTRIBUTE POINT directive, 14-18
example of, 14-18

Division operator (/), 4-2
precedence of, 4-11
See also Slash character

DLLEXPORT (W*32, W*64)
option for ATTRIBUTES directive, 14-13

DLLIMPORT (W*32, W*64)
option for ATTRIBUTES directive, 14-13

DLOG function, 9-89

DLOG10 function, 9-90

DMAX1 function, 9-93

DMIN1 function, 9-99

DMOD function, 9-105

DNINT function, 9-25

DNUM function, 9-49

DO constructs, 7-14
block form of, 7-15
examples of, 7-16
execution of, 7-17
extended range for, 7-21
forms of, 7-15
immediate termination of, 7-24
interrupting, 7-24
iteration control in, 7-17
nested, 7-19
control transfers in, 7-21
nonblock form of, 7-15
range of, 7-17
terminal statement for labeled, 7-14
WHILE, 7-23

DO directive, 14-55
example of, 14-53, 14-57, 14-62

DO loop iterations
OpenMP directive to execute in parallel, 14-55

DO loops, 7-14
directive assisting dependence analysis of, 14-23
directive enabling prefetching of arrays in, 14-32
directive enabling software pipelining for, 14-37
directive enabling streaming storage, 14-41
directive specifying alignment of data in, 14-39
directive specifying auto-parallelization for, 14-31
directive specifying distribution for, 14-18
directive specifying the count for, 14-25
directive specifying unroll count for, 14-39
directive specifying vectorization for, 14-40
skipping, 7-24
transferring control, 7-21

DO WHILE statement, 7-14, 7-17, 7-23
examples of, 7-23
terminating, 7-23

Dollar sign character ($)
as edit descriptor, 11-37
in names, 2-4

Dollar sign editing, 11-37

DOT_PRODUCT function, 9-50

Dot-product multiplication
function performing, 9-50

Double colon separator, 5-4

DOUBLE COMPLEX data type, 3-3
constants, 3-12
function converting to, 9-47
function converting to double-precision real, 9-51
in type declaration statements, 5-2, 5-6
See also COMPLEX(8)
storage, 15-14
DOUBLE PRECISION data type, 3-2, 3-6
constants, 3-7, 3-9
function converting to, 9-48, 9-49, 9-51
in type declaration statements, 5-2, 5-6
See also REAL(8)
See also your user's guide
storage, 15-14
Double-precision product
function producing, 9-51
DPROD function, 9-51
DREAL function, 9-51
DSHIFTL, 9-52
DSHIFTR, 9-52
DSIGN function, 9-134
DSIN function, 9-135
DSIND function, 9-135
DSINH function, 9-136
DSQRT function, 9-139
DTAN function, 9-142
DTAND function, 9-143
DTANH function, 9-143
Dummy argument arrays, 5-13
Dummy arguments, 8-30, 8-37
definition of, 8-30
intent of, 5-41
optional, 5-46, 8-32
present, 5-47
See also your user's guide
specifying intended use of, 5-41
using aggregate field references as, B-23
Dummy procedures, 8-37
definition of, 8-1
interfaces for, 8-38
scope of, 15-2
using as actual arguments, 5-38
Dynamic allocation
of allocatable arrays, 6-2, 6-3
of pointer targets, 6-2, 6-4
Dynamic deallocation
of allocatable arrays, 6-5, 6-6
of pointer targets, 6-5, 6-7
Dynamic disassociation of pointers, 6-8
Dynamic memory management, 6-1
Dynamic objects
automatic array as, 5-12
in character declarations, 5-8
pointers and allocatable arrays as, 6-1
DYNAMIC schedule type, 14-56

E

E edit descriptor, 11-16
input processing, 11-17
output processing, 11-17
Edit descriptors
character string, 11-38
control, 11-30
forms for, 11-30
data, 11-6
forms for, 11-6
rules for numeric, 11-8
summary of, 11-3
Editing
character, 11-26
general rules for numeric, 11-8
integer, 11-9
logical, 11-25
real and complex, 11-14
Elapsed time
function calculating in seconds, 9-130
Element array assignment statements (FORALL), 4-26
Elemental intrinsic procedures
definition of, 9-1
references to, 8-42
ELEMENTAL prefix, 8-17
in FUNCTION statements, 8-18
in SUBROUTINE statements, 8-25
Elemental user-defined procedures, 8-17
examples of, 8-17

functions as, 8-18
subroutines as, 8-25

Elements, 3-35, 3-38
See also Array elements

ELSE directive, 14-20

ELSE IF directive, 14-20

ELSE IF statement, 7-26
branching to, 7-27

ELSE statement, 7-26
branching to, 7-27

ELSEWHERE statement, 4-23

EN edit descriptor, 11-19
input processing, 11-19
output processing, 11-19

ENCODE statement, B-3

END branch specifier, 10-7

END DO statement, 7-16, 7-23

END IF directive, 14-20

END IF statement, 7-26
branching to, 7-27

END statement, 7-25
retaining data after execution of, 5-54

Endfile record
definition of, 10-2
writing to a file, 12-5

ENDFILE statement, 12-5
examples of, 12-6

End-off shift on arrays
function performing, 9-54

End-of-file condition, 10-8
function to check, 9-53
I/O specifier for, 10-7

End-of-file records, 10-2
writing to a file, 12-5

End-of-record condition, 10-8
I/O specifier for, 10-7

Engineering notation
descriptor for (EN), 11-19

Entry names
referencing, 8-54

Entry points
for function subprograms, 8-55
for subprograms, 8-53
for subroutine subprograms, 8-56

ENTRY statement, 8-53
examples of, 8-55, 8-56
in function subprograms, 8-55
in subroutine subprograms, 8-56
RESULT keyword in, 8-53
result variable in, 8-55
using with FUNCTION statement, 8-20
using with SUBROUTINE statement, 8-25

Environment variables
FORTn, 12-32
OMP_SCHEDULE, 14-56

EOF function, 9-53

EOR branch specifier, 10-7

EOSHIFT function, 9-54

EPSILON function, 9-56

Equivalence
association, 5-29
logical, 4-8
objects, 5-29
set, 5-29

EQUIVALENCE statement, 5-29
compared to union declaration, B-20
examples of, 5-30
interaction with COMMON, 5-35
See also your user's guide
using with arrays, 5-31
using with substrings, 5-33

ERF function, 9-57

ERFC function, 9-57

ERR branch specifier, 10-7

Error conditions, 10-7
I/O specifier for, 10-7
subroutine returning information on, 9-58

Error functions
functions returning, 9-57

Errors
See your user's guide

ERRSNS subroutine, 9-58

ES edit descriptor, 11-20
input processing, 11-21
output processing, 11-21

Escape sequences
C-style, 3-16
Exclamation point character (!)
as comment indicator, 2-10, 2-11
Exclusive OR, 4-8
function performing, 9-71
Executable constructs
named, 7-1
Executable statements, 2-2
disallowed in main programs, 8-3
Executing
DO loops, 7-17
programs, 8-1
See also Program execution
EXIST specifier
in INQUIRE statements, 12-12
EXIT statement, 7-14, 7-24
EXIT subroutine, 9-59
EXP function, 9-59
Explicit format, 11-1, 11-2
using character expressions, 11-4
Explicit interfaces, 8-45
defining, 8-46
of dummy procedures, 8-38
when required, 8-46
Explicit-shape arguments, 8-33
Explicit-shape arrays, 5-11
adjustable, 5-13
automatic, 5-12
EXPONENT function, 9-60
Exponential operator (**), 4-2
in initialization expressions, 4-13
precedence of, 4-11
Exponential values
function returning, 9-59
Exponents
function returning range of decimal, 9-124
Expressions, 4-1
character, 4-6
determining data type of, 4-6
element array, 4-26
initialization, 4-12
length
effect on character assignments, 4-19
logical, 4-8
masked array, 4-23
numeric, 4-2
effects of parentheses within, 4-4
operator precedence in, 4-3
order of evaluation in, 4-3
relational, 4-7
See also Character expressions
See also Logical expressions
See also Numeric expressions
See also Relational expressions
specification, 4-13
variable format, 11-41
with arrays as operands, 4-2
with pointers as operands, 4-2
EXTEND_SOURCE
OPTIONS statement option, 13-3
Extended intrinsic operators
properties of, 8-50
Extended ranges
for DO constructs, 7-21
Extending arrays, 9-127, 9-138
Extent
definition of, 3-36
function returning, 9-136
EXTERN
option for ATTRIBUTES directive, 14-13
EXTERNAL attribute and statement, 5-38
attributes compatible with, 5-5
examples of, 5-39
External field separators, 11-3, 11-9
comma as, 11-29
External fields
separating, 11-29
External files
connecting to units, 12-20
definition of, 10-2
External procedures, 8-1, 8-28
compared to internal procedures, 8-29
declaring, 5-38
definition of, 8-1
directive specifying alternate name for, 14-5

invoking with CALL, 7-7
scope of, 15-2
See also Functions
See also Subroutines
using as actual arguments, 5-38
with the same name as intrinsic procedures, 5-38

External records, 10-2
transferring
character strings to, 11-39
direct-access input, 10-24
direct-access output, 10-35, 10-39
sequential access input, 10-13, 10-28
sequential access output, 10-29, 10-38

EXTERNAL statement
block data program unit in, 8-11
FORTRAN-66 implementation of, B-6
names in, 5-38
using with intrinsic procedures, 8-39

External subprograms, 2-1
providing entry points within, 8-53

F

F edit descriptor, 11-15
input processing, 11-15
output processing, 11-16

F77 OPTIONS statement option, 13-3

FALSE value
for INQUIRE (EXIST), 12-12
for INQUIRE (IOFOCUS), 12-14
for INQUIRE (NAMED), 12-15
for INQUIRE (OPENED), 12-16
for OPEN (IOFOCUS), 12-33

FDX value
for INQUIRE (CONVERT), 12-11
for OPEN (CONVERT), 12-28

FGX value
for INQUIRE (CONVERT), 12-11
for OPEN (CONVERT), 12-28

Field
definition of external, 11-6
in fixed source form, 2-13
in tab source form, 2-13
See also your user's guide

Field descriptors, 11-6
See also Data edit descriptors

Field names
in record structures, B-14, B-18

Field width
definition of, 11-7

File connection statements
CLOSE, 12-3
OPEN, 12-20

File connections
creating, 12-20

File inquiry statement (INQUIRE), 12-7

File name
in INQUIRE statements, 12-7
in OPEN statements, 12-20, 12-32
specifying default pathname as, 12-7, 12-30

File position statements
BACKSPACE, 12-2
ENDFILE, 12-5
REWIND, 12-41

File properties
inquiring about, 12-7

FILE specifier
in OPEN statements, 12-20, 12-32

Files
access methods for, 10-2
accessing with INCLUDE, 13-1
combining at compilation, 13-1
directive specifying a subtitle for listing header, 14-38
directive specifying a title for listing header, 14-38
disconnecting, 12-3
external, 10-2
See also External files
inquiring about properties of, 12-7
internal, 10-2
See also Internal files
opening, 12-20
relative
See Relative files
sequential
See Sequential files
types of, 10-2

FIND statement, B-5
compared to READ statement, B-5

FIPS standard, 1-2
FIRSTPRIVATE clause, 14-46
in DO directive, 14-55
in PARALLEL directive, 14-62
in PARALLEL DO directive, 14-65
in PARALLEL SECTIONS directive, 14-66
in SECTIONS directive, 14-67
in SINGLE directive, 14-68
Fixed source form, 2-11
blank characters in, 2-11
comment indicator in, 2-11
continuation character in, 2-11
debugging statement indicator in, 2-12
directive setting line length for, 14-19
fields in, 2-13
labels in, 2-8
sequence number field in, 2-13
short source lines in, 2-11
statement field in, 2-13
statement separator in, 2-7
FIXED value
for INQUIRE (RECORDTYPE), 12-18
for OPEN (RECORDTYPE), 12-37
FIXEDFORMLINESIZE directive, 14-19
example of, 14-19
Flags
See Compiler options
FLOAT
OPTIONS statement option, 13-3
FLOAT function, 9-125
FLOATI function, 9-125
Floating-point data types, 3-6
See also your user's guide
FLOATJ function, 9-125
FLOATK function, 9-125
FLOOR function, 9-60
Flow of control
in CASE construct, 7-11
in IF construct, 7-27
FLUSH directive, 14-59
example of, 14-60
FMT specifier, 10-5
FORALL construct and statement, 4-26
evaluation of, 4-27
examples of, 4-28, 8-16
pure procedures in, 4-27
FORCEINLINE
option for ATTRIBUTES directive, 14-14
Foreground process
temporarily suspending, 7-32
FORM specifier
in INQUIRE statements, 12-13
in OPEN statements, 12-32
Format
control, 11-43
explicit, 11-1, 11-2
implicit, 11-1
list-directed input, 10-15
list-directed output, 10-31
namelist input, 10-18
namelist output, 10-33
reversion, 11-43
rules for numeric, 11-8
See also Format specifications
specifier, 10-5
using character string edit descriptors, 11-38
using control edit descriptors, 11-30
using data edit descriptors, 11-6
zero-length numeric, 11-9
Format specifications
blanks in, 11-2
character, 11-4
definition of, 11-2
examples of, 11-5
extensions to, F-3
group repeat specifications in, 11-40
nested specifications in, 11-40
omitting a comma in, 11-2
output characters in, 11-3
repeat specifications in, 11-2
See also Format
summary of edit descriptors, 11-3
FORMAT statements, 11-2
field width
output size for D descriptor, 11-17
output size for E descriptor, 11-17
output size for EN descriptor, 11-19

output size for ES descriptor, 11-21
output size for F descriptor, 11-16
output size for G descriptor, 11-23
format reversion with I/O lists, 11-43
I/O lists
interaction with, 11-43
interpretation of blanks in, 11-3
See also Format
See also Format specifications
See also your user's guide
variable format expressions in, 11-41

Formatted data
transfer
See also READ statements
See also WRITE statements
specifier to test for, 12-13
specifying, 12-32

Formatted I/O statements
ACCEPT, 10-28
establishing labels for, 7-4
PRINT and TYPE, 10-38
READ
direct access, 10-24, 10-25
internal, 10-26
sequential, 10-13, 10-15
REWRITE, 10-39
using aggregate field references in, B-23
WRITE
direct access, 10-35, 10-36
internal, 10-36
sequential, 10-29, 10-30

Formatted records
definition of, 10-1
printing, 11-42

FORMATTED specifier
in INQUIRE statements, 12-13

FORMATTED value
for INQUIRE (FORM), 12-13
for OPEN (FORM), 12-32

Forms
for source code, 2-6

FORTn environment variable, 12-32

Fortran 2000 features, 1-2

FORTRAN 77 standard, 1-2

Fortran 90 features, 1-5

Fortran 90 standard, 1-2
directive disabling features not found in, 14-36

Fortran 95 features, 1-3

Fortran 95 standard, 1-2
directive disabling features not found in, 14-36
language extensions to, F-1

Fortran 95/90 character set, 2-5
extensions to, F-1

FORTRAN value
for INQUIRE (CARRIAGECONTROL), 12-11
for OPEN (CARRIAGECONTROL), 12-28

FORTRAN-66 semantics
effect on EXTERNAL, B-6

FP_CLASS function, 9-61

FRACTION function, 9-61

Free source form, 2-9
blank characters in, 2-9
comment indicator in, 2-10
continuation character in, 2-10
directive indicating, 14-20
labels in, 2-8
number of characters in a line, 2-9
statement separator in, 2-7

FREE subroutine, 9-62

FREEFORM directive, 14-20

Function references, 8-12, 8-23
elemental intrinsic, 8-42
example of, 8-24
to external names, 5-39

FUNCTION statement, 8-18
examples of, 8-20
prefixes in, 8-18
RESULT keyword in, 8-23
using with ENTRY statement, 8-20

Function subprograms, 8-12
See also Functions
See also Subprograms

Functions, 8-18
allocatable, 8-19
example of, 8-21
applying to arrays, 9-1
applying to scalar and array arguments, 9-1

character type, 8-19
containing OPERATOR specifier, 4-10, 8-50
declaring as external, 5-38
declaring as intrinsic, 5-43
definition of, 8-2
depending on the properties of an argument, 9-1
elemental intrinsic, 9-1
elemental user-defined, 8-17
ENTRY statements in, 8-55
example of allocatable, 8-21
examples of, 8-20
general rules for, 8-13
generic, 9-1
global intrinsic, 8-40
inquiry, 9-1
invoking, 8-23
invoking in a CALL statement, 8-20
local intrinsic, 8-40
not allowed as actual arguments, 9-2
prefixes in, 8-18
pure, 8-14
recursion in, 8-13
referencing, 8-23
result variable in, 8-23
specific, 9-1
statement defining, 8-18
transformational, 9-1

G

G edit descriptor, 11-22
input processing, 11-15
output processing, 11-16
General compiler directives, 14-2
affecting DO loops, 14-4
ALIAS, 14-5
ATTRIBUTES, 14-5
DECLARE, 14-16
DEFINE, 14-17
DISTRIBUTE POINT, 14-18
ELSE, 14-20
ELSEIF, 14-20
ENDIF, 14-20
FIXEDFORMLINESIZE, 14-19
FREEFORM, 14-20
IDENT, 14-20
IF, 14-20
IF DEFINED, 14-20
INTEGER, 14-22
IVDEP, 14-23
LOOP COUNT, 14-25
MESSAGE, 14-26
NODECLARE, 14-16
NOFREEFORM, 14-20
NOPARALLEL, 14-31
NOPREFETCH, 14-32
NOSTRICT, 14-36
NOSWP (i64), 14-37
NOVECTOR (i32), 14-40
OBJCOMMENT, 14-26
OPTIONS, 14-27
PACK, 14-30
PARALLEL, 14-31
PREFETCH, 14-32
PSECT, 14-33
REAL, 14-35
STRICT, 14-36
SUBTITLE, 14-38
SWP (i64), 14-37
syntax of prefix, 14-1
TITLE, 14-38
UNDEFINE, 14-17
UNROLL, 14-39
VECTOR ALIGNED (i32), 14-39
VECTOR ALWAYS (i32), 14-40
VECTOR NONTEMPORAL (i32), 14-41
VECTOR UNALIGNED (i32), 14-39
Generalized editing (G), 11-22
Generic assignment, 4-21
for procedures, 8-51
Generic identifier, 8-45, 8-47
scope of, 15-2
Generic interfaces, 8-46
in scoping units, 8-9
Generic names
for procedures, 8-49
of intrinsics, 9-1
Generic operators
for procedures, 8-50
Generic procedures
example of, 15-6

references to, 8-38
references to intrinsic, 8-39

Generic references
example of, 8-40

GETARG subroutine, 9-62

Global properties
of intrinsic functions, 8-40

Global scope
names having, 15-1

GO TO statements
assigned, 7-5
computed, 7-3
establishing labels for assigned, 7-4
unconditional, 7-2

Graphics routines (W*32, W*64), E-16

Group repeat format specifications, 11-40

GUIDED schedule type, 14-56

H

H edit descriptor
alternative for, A-3

H editing, 11-39

HABS function, 9-18

HBCLR function, 9-67

HBITS function, 9-68

HBSET function, 9-69

Hexadecimal constants, 3-29
alternative syntax for, B-10
assigning with DATA statement, 5-25
data type assignments of, 3-31
examples of, 3-30

Hexadecimal values
transferring, 11-13

HFIX function, 9-73

HIAND function, 9-65

HIEOR function, 9-71

HIOR function, 9-76

HIXOR function, 9-71

HMOD function, 9-105

HMVBITS subroutine, 9-107

HNOT function, 9-111

Hollerith constants, 3-30
as arguments, 8-36
assigned with DATA statements, 5-26
continuation in fixed and tab source, 2-12
data type assignments of, 3-31
examples of, 3-30
See also your user's guide
uppercase and lowercase letters in, 2-6

Hollerith values
transferring, 11-26

Host, 2-2
association, 15-11
IMPLICIT NONE for, 15-11
in modules, 8-5
scope of, 15-11

HSHFT function, 9-79

HSHFTC function, 9-80

HSIGN function, 9-134

HTEST function, 9-33

HUGE function, 9-64

Hyperbolic arccosine
function returning, 9-21

Hyperbolic arcsine
function returning, 9-27

Hyperbolic arctangent
function returning, 9-32

Hyperbolic cosine
function returning, 9-39

Hyperbolic sine
function returning, 9-136

Hyperbolic tangent
function returning, 9-143

I

I edit descriptor, 11-9
input processing, 11-10
output processing, 11-10

I/O
advancing and nonadvancing, 10-8

I/O control list, 10-3
advance specifier in, 10-8
branch specifiers in, 10-7

character count specifier in, 10-9
format specifier in, 10-5
namelist specifier in, 10-6
record specifier in, 10-6
specifiers in, 10-3
status specifier in, 10-6
unit specifier in, 10-4

I/O data transfer
formatted direct access
READ, 10-25
REWRITE, 10-39
WRITE, 10-36
formatted sequential
ACCEPT, 10-28
PRINT and TYPE, 10-38
READ, 10-15
WRITE, 10-30
internal
READ, 10-26
WRITE, 10-36
unformatted direct access
READ, 10-26
REWRITE, 10-39
WRITE, 10-36
unformatted sequential
READ, 10-23
WRITE, 10-34

I/O lists, 10-9
arrays in, 10-9
derived-type items in, 10-10
general rules for, 10-9
implied-DO lists in, 10-12
interaction with FORMAT statements, 11-43
items in, 10-10
pointers in, 10-9

I/O statements
ACCEPT, 10-28
auxiliary, 12-1
BACKSPACE, 12-2
CLOSE, 12-3
DELETE, 12-4
ENDFILE, 12-5
extensions for file operations, F-4
extensions to, F-3
for data transfer, 10-1
for operations on files, 12-1
formatting, 11-1
INQUIRE, 12-7
list-directed
input, 10-15
output, 10-31
namelist, 5-45
input, 10-18
output, 10-33
OPEN, 12-20
PRINT and TYPE, 10-38
READ, 10-13
REWIND, 12-41
REWRITE, 10-39
UNLOCK, 12-42
WRITE, 10-29

I/O status specifier, 10-6

I/O units
associating with files, B-1
definition of, 10-4
inquiring about properties of, 12-7
scope of, 15-2

I4
OPTIONS statement option, 13-3

IABS function, 9-18

IACHAR function, 9-64

IADDR function, 9-32

IAND function, 9-65

IARG function, 9-66

IARGC function, 9-66

IARGPTR function, 9-66

IBCHNG function, 9-67

IBCLR function, 9-67

IBITS function, 9-68

IBM value
for INQUIRE (CONVERT), 12-11
for OPEN (CONVERT), 12-28

IBSET function, 9-69

ICHAR function, 9-69, 9-70

IDATE subroutine, 9-71

IDENT directive, 14-20

IDIM function, 9-49

IDINT function, 9-73
IDNINT function, 9-110
IEEE* values
    function testing for NaN, 9-82
    See also your user's guide
IEOR function, 9-71
IF clause
    in PARALLEL directive, 14-62
    in PARALLEL DO directive, 14-65
    in PARALLEL SECTIONS directive, 14-66
IF constructs, 7-26
    branching in, 7-27
    examples of, 7-29
    flow of control in, 7-27
    nested, 7-29
IF DEFINED directive, 14-20
IF directive, 14-20
    example of, 14-22
IF loops
    flow of control in, 7-27
IF statements, 7-31
    arithmetic, 7-6
    examples of, 7-32
IF THEN statement, 7-26
IFIX function, 9-73
ifort command
    overriding, 13-3
    See also your user's guide
    statements affecting, 13-1
IGNORE_LOC
    option for ATTRIBUTES directive, 14-13
IIABS function, 9-18
IIAND function, 9-65
IIBCLR function, 9-67
IIBITS function, 9-68
IIBSET function, 9-69
IIDIM function, 9-49
IIDINT function, 9-73
IIDNNT function, 9-110
IIEOR function, 9-71
IIFIX function, 9-73
IINT function, 9-73
IIOR function, 9-76
IIQINT function, 9-73
IIQNNT function, 9-110
IISHFT function, 9-79
IISHFTC function, 9-80
IISIGN function, 9-134
IIXOR function, 9-71
IJINT function, 9-73
ILEN function, 9-72
IMAG function, 9-22
IMAX0 function, 9-93
IMAX1 function, 9-93
IMIN0 function, 9-99
IMIN1 function, 9-99
IMOD function, 9-105
Implicit data typing
    overriding default, 5-39
Implicit format, 11-1
    list-directed input, 10-15
    list-directed output, 10-31
    namelist input, 10-18
    namelist output, 10-33
Implicit interfaces, 8-45
    of dummy procedures, 8-38
IMPLICIT NONE statement, 5-39
    directive producing similar warnings, 14-16
    for host association, 15-11
    rules for, 5-40
IMPLICIT statement, 5-39
    and host association, 15-11
    examples of, 5-40
    restriction with intrinsic procedures, 5-40
    using to type variables, 3-34
    using with intrinsic procedures, 8-40
Implied-DO lists
    in DATA statements, 5-24
    in data transfer statements, 10-12
    in I/O lists, 10-12
    scope of variables in, 15-2
Implied-DO loops
    in array constructors, 3-44
    in I/O lists, 10-12

iteration control for, 7-17
Implied-DO variables
initializing with DATA statement, 5-24
scope of, 15-2
IMVBITS subroutine, 9-107
INCLUDE statements, 13-1
example of, 13-2
Including files during compilation, 13-1
Inclusive OR, 4-8
function performing, 9-76
Indefinite DO statement, 7-23
Index
case, 7-10
INDEX function, 9-73
Indexed DO statement, 7-15
See also Block DO construct
Inequivalence
logical, 4-8
ININT function, 9-110
Initialization expressions, 4-11, 4-12
examples of, 4-13
exponential operators in, 4-13
for derived-type components, 3-21, 3-22
in type declaration statements, 5-4
inquiry functions allowed in, 4-12
invoking inquiry functions in, 4-13
simplest form of, 4-12
transformational functions allowed in, 4-12
Initialization of data
default, 3-22
explicit, 5-2
INLINE
option for ATTRIBUTES directive, 14-14
INOT function, 9-111
Input data
terminating short fields of, 11-29
Input statements, 10-13
ACCEPT, 10-28
READ, 10-13
INQUIRE statements, 12-7
ACCESS specifier in, 12-8
ACTION specifier in, 12-9
BINARY specifier in (W*32, W*64), 12-9
BLANK specifier in, 12-9
BLOCKSIZE specifier in, 12-10
BUFFERED specifier in, 12-10
CARRIAGECONTROL specifier in, 12-10
CONVERT specifier in, 12-11
DELIM specifier in, 12-12
DIRECT specifier in, 12-12
examples of, 12-8
EXIST specifier in, 12-12
FORM specifier in, 12-13
FORMATTED specifier in, 12-13
general description of, 12-7
IOFOCUS specifier in (W*32, W*64), 12-13
MODE specifier in, 12-14
NAME specifier in, 12-14
NAMED specifier in, 12-14
NEXTREC specifier in, 12-15
NUMBER specifier in, 12-15
OPENED specifier in, 12-15
ORGANIZATION specifier in, 12-16
PAD specifier in, 12-16
POSITION specifier in, 12-16
READ specifier in, 12-17
READWRITE specifier in, 12-17
RECL specifier in, 12-17
RECORDTYPE specifier in, 12-18
See also your user's guide
SEQUENTIAL specifier in, 12-18
SHARE specifier in (W*32, W*64), 12-19
UNFORMATTED specifier in, 12-19
WRITE specifier in, 12-19
Inquiry
bit function, 9-4
functions for numeric, 9-4
Inquiry functions
ALLOCATED, 9-24
allowed in initialization expressions, 4-12
allowed in specification expressions, 4-14
ASSOCIATED, 9-28
BIT_SIZE, 9-33
CACHESIZE, 9-34
definition of, 9-1
DIGITS, 9-48
EOF, 9-53
EPSILON, 9-56
for argument presence, 9-115

for arrays, 9-24, 9-83, 9-132, 9-136, 9-147
for bit size, 9-33
for character length, 9-85
for numeric models, 9-48, 9-56, 9-64, 9-95, 9-100, 9-114, 9-120, 9-124, 9-144
for pointers, 9-28
HUGE, 9-64
IARGC, 9-66
IARGPTR, 9-66
INT_PTR_KIND, 9-75
invoking in initialization expressions, 4-13
KIND, 9-82
LBOUND, 9-83
LEN, 9-85
LOC, 9-89
MAXEXPONENT, 9-95
MINEXPONENT, 9-100
NARGS, 9-108
PRECISION, 9-114
PRESENT, 9-115
RADIX, 9-120
RANGE, 9-124
SHAPE, 9-132
SIZE, 9-136
SIZEOF, 9-137
TINY, 9-144
UBOUND, 9-147

INT function, 9-73

INT_PTR_KIND function, 9-75

INT1 function, 9-73

INT2 function, 9-73

INT4 function, 9-73

INT8 function, 9-73

INTEGER data type, 3-2, 3-4
constants, 3-4
default kind, 3-4
function converting to, 9-73
function converting to double-precision real, 9-46, 9-48
function converting to quad-precision real, 9-117, 9-118
in type declaration statements, 5-2, 5-6
See also INTEGER(4)
See also your user's guide
storage, 15-14

INTEGER directive, 14-22
example of, 14-23

Integer editing (I,B,O,Z), 11-9
B, 11-11
I, 11-9
O, 11-12
Z, 11-13

INTEGER KIND for address
function returning, 9-75

Integer model, D-2
function returning largest number in, 9-64

Integer pointers, B-12

Integer values
transferring, 11-9

Integer variables
assigning labels to, 7-4

INTEGER(1)
constants, 3-4
storage requirements, 15-14

INTEGER(2)
constants, 3-4
storage requirements, 15-14

INTEGER(4)
constants, 3-4
function converting to INTEGER(2) type, 9-73
storage requirements, 15-14

INTEGER(8)
constants, 3-4
storage requirements, 15-14

INTEGER*1 constants
See INTEGER(1)

INTEGER*2 constants
See INTEGER(2)

INTEGER*4 constants
See INTEGER(4)

INTEGER*8 constants
See INTEGER(8)

Integers
bit representation of, 9-5
constants, 3-4
in COMPLEX constants, 3-11
using to assign values, 3-5

default
See "default integer" in the Glossary
directive specifying default kind, 14-22
function converting to double-precision type, 9-48
function multiplying two 64-bit unsigned, 9-106
function returning difference between, 9-49
function returning leading zero bits in, 9-84
function returning number of 1 bits in, 9-114
function returning parity of, 9-114
function returning trailing zero bits in, 9-145
function returning two's complement length of, 9-72
model for data, D-2

Intent
of arguments, 5-41

INTENT attribute and statement, 5-41
attributes compatible with, 5-5
examples of, 5-42
for pure procedures, 8-15

INTERFACE
ASSIGNMENT
examples of, 8-52
generic
examples of, 8-49
OPERATOR
examples of, 8-50

Interface blocks, 8-2, 8-46
examples of, 8-48, 8-49, 8-50, 8-52
for generic assignment, 8-51
for generic names, 8-49
for generic operators, 8-50

INTERFACE statement, 8-46
defining generic assignment, 8-51
defining generic names, 8-49
defining generic operators, 8-50
generic identifier in, 8-46

INTERFACE TO statement, B-5

Interfaces, 8-45
defining explicit, 8-46
for dummy procedures, 8-38
generic, 8-46
in scoping units, 8-9
of external procedures, 8-29
of internal procedures, 8-30
procedures requiring explicit, 8-46

Internal address
function returning, 9-89

Internal files
definition of, 10-2
position of, 10-5
storage of, 10-4

Internal I/O statements
ENCODE and DECODE, B-3
READ, 10-26
WRITE, 10-36

Internal procedures, 8-29
compared to external procedures, 8-29
definition of, 8-1
example of, 8-30
scope of, 15-2

Internal READ statement, 10-26

Internal subprograms, 2-2, 8-29
introducing in program unit, 8-53

Internal WRITE statement, 10-36

Interrupting
DO constructs, 7-24

Intrinsic assignments, 4-16
array, 4-20
character, 4-18
derived-type, 4-19
logical, 4-18
numeric, 4-17
scope of, 15-2
See also Assignment statements
types of, 4-16

INTRINSIC attribute and statement, 5-43
attributes compatible with, 5-5
examples of, 5-44
names in, 5-44

Intrinsic data types
character, 3-14
complex, 3-10
integer, 3-4
logical, 3-14
numeric nondecimal constants, 3-28
real, 3-6
storage requirements for, 15-14

Intrinsic functions
alphabetical descriptions of all, 9-18 thru 9-152

categories of array, 9-5
categories of bit, 9-5
categories of character, 9-5
categories of numeric, 9-4
example of using as arguments, 8-40
for data representation models, D-1
kind type, 9-5
mathematical, 9-5
miscellaneous, 9-6
non-generic, 9-14
not allowed as actual arguments, 9-2
references to generic, 8-39
See also Functions
See also Intrinsic procedures
specified as INTRINSIC, 5-44
summary of generic, 9-6
using external procedures of same name as, 5-38

Intrinsic operators
properties of extended, 8-50
scope of, 15-2

Intrinsic procedures, 9-1
alphabetical descriptions of all, 9-18 thru 9-152
bit functions, 9-16
categories of array, 9-5
categories of bit, 9-5
categories of character, 9-5
categories of numeric, 9-4
classes of, 9-1
definition of, 8-1
extensions of KIND argument, F-9
extensions to, F-7
functions
See Intrinsic functions
inquiry functions
See Inquiry functions
keywords for, 9-3
kind type, 9-5
mathematical, 9-5
miscellaneous, 9-6
names of, 9-1
references to elemental, 8-42
references to generic, 8-39
scope of, 15-2
scope of name, 8-39
See also Elemental intrinsic procedures
subroutines
See Intrinsic subroutines
transformational functions
See Transformational functions
using as actual arguments, 5-43
using with EXTERNAL statement, 8-39
using with IMPLICIT statement, 8-40

Intrinsic subroutines, 9-1, 9-15
alphabetical descriptions of all, 9-18 thru 9-152
See also Intrinsic procedures
See also Subroutines

INUM function, 9-76

Inverse cosine
function returning in degrees, 9-20
function returning in radians, 9-20

Inverse sine
function returning in degrees, 9-27
function returning in radians, 9-26

Inverse tangent
function returning in degrees, 9-31
function returning in degrees (complex), 9-31
function returning in radians, 9-29
function returning in radians (complex), 9-30

IOFOCUS specifier (W*32, W*64)
in INQUIRE statements, 12-13
in OPEN statements, 12-33

IOR function, 9-76

IOSTAT specifier, 10-6

IQINT function, 9-73

IQNINT function, 9-110

ISHA function, 9-77

ISHC function, 9-78

ISHFT function, 9-79

ISHFTC function, 9-80

ISHL function, 9-81

ISIGN function, 9-134

ISNAN function, 9-82

ISO standards, 1-2

Iteration count, 7-17
and loop control, 7-17

Iterative DO loop, 7-17

IVDEP directive, 14-23
    example of, 14-24
IXOR function, 9-71
IZEXT function, 9-150

J

JFIX function, 9-73
JIABS function, 9-18
JIAND function, 9-65
JIBCLR function, 9-67
JIBITS function, 9-68
JIBSET function, 9-69
JIDIM function, 9-49
JIDINT function, 9-73
JIDNNT function, 9-110
JIEOR function, 9-71
JIFIX function, 9-73
JINT function, 9-73
JIOR function, 9-76
JIQINT function, 9-73
JIQNNT function, 9-110
JISHFT function, 9-79
JISHFTC function, 9-80
JISIGN function, 9-134
JIXOR function, 9-71
JMAX0 function, 9-93
JMAX1 function, 9-93
JMIN0 function, 9-99
JMIN1 function, 9-99
JMOD function, 9-105
JMVBITS subroutine, 9-107
JNINT function, 9-110
JNOT function, 9-111
JNUM function, 9-82
JZEXT function, 9-150

K

KEEP value
    for CLOSE statements, 12-3
    for OPEN (DISPOSE), 12-31
Keywords
    argument, 9-3
    directive
        See Compiler directives
    for control-list specifiers, 10-3
    statement
        See Statements
KIABS function, 9-18
KIAND function, 9-65
KIBCLR function, 9-67
KIBITS function, 9-68
KIBSET function, 9-69
KIDIM function, 9-49
KIDINT function, 9-73
KIDNNT function, 9-110
KIEOR function, 9-71
KIFIX function, 9-73
KIND
    directive specifying default for reals, 14-35
    function, 9-82
    keyword
        in intrinsics, 9-3
Kind functions
    definition of, 9-4
Kind parameters
    See Kind type parameter
Kind selector, 5-3
Kind type parameter
    for character constants, 3-3, 3-14
    for complex constants, 3-10
    for integer constants, 3-4
    for logical constants, 3-14
    for real constants, 3-6
    function returning value of, 9-82
    function selecting logical, 9-91
    functions to determine, 9-4
    of integer data
        function returning, 9-130

of real data
function returning, 9-131
restriction for real constants, 3-8
selector for, 5-3
KINT function, 9-73
KIOR function, 9-76
KIQINT function, 9-73
KIQNNT function, 9-110
KISHFT function, 9-79
KISHFTC function, 9-80
KISIGN function, 9-134
KMAX0 function, 9-93
KMAX1 function, 9-93
KMIN0 function, 9-99
KMIN1 function, 9-99
KMOD function, 9-105
KMVBITS subroutine, 9-107
KNINT function, 9-110
KNOT function, 9-111
KZEXT function, 9-150

L

L edit descriptor, 11-25
input processing, 11-25
output processing, 11-25
Labels
assigning, 7-4
in block DO constructs, 7-16
in formatted I/O statements, 7-4
in nonblock DO constructs, 7-16
in source form, 2-8
platform, xxxi
scope of, 15-2
See also your user's guide
Language compatibility
extensions for, F-9
features for, B-1
See also your user's guide
summary of, 1-2
Language extensions, F-1
directive disabling, 14-36
summary of, F-1
Language features
deleted in Fortran 95, A-1
for compatibility with older versions, B-1
obsolescent in Fortran 90, A-3
obsolescent in Fortran 95, A-2
LASTPRIVATE clause, 14-46
in DO directive, 14-55
in PARALLEL DO directive, 14-65
in PARALLEL SECTIONS directive, 14-66
in SECTIONS directive, 14-67
LBOUND function, 9-83
in pointer assignment, 5-16
LEADZ function, 9-84
Left shift
function performing arithmetic, 9-52, 9-77, 9-133
function performing circular, 9-78
function performing logical, 9-81
LEN
in character type declaration statements, 5-8
LEN function, 9-85
LEN_TRIM function, 9-85
Length specifier
in character type declaration statements, 5-8
Lexical string comparisons, 9-4
function determining .GE., 9-86
function determining .GT., 9-87
function determining .LE., 9-87
function determining .LT., 9-88
LGE function, 9-86
LGT function, 9-87
Library module routines, E-1
Library routines
DFAUTO (W*32), E-24
DFCOM (W*32), E-23
dialog (W*32), E-20
graphics (W*32, W*64), E-16
miscellaneous, E-22
NLS, E-8
parallel, E-25
portability, E-2
POSIX, E-10
QuickWin (W*32, W*64), E-14

List items
I/O, 10-10
LIST option
in INCLUDE statement, 13-1
LIST value
for INQUIRE (CARRIAGECONTROL), 12-11
for OPEN (CARRIAGECONTROL), 12-28
List-directed formatting
defaults for output, 10-31
for READ statements
sequential, 10-15
for WRITE statements
sequential, 10-31
indicator for, 10-5
input, 10-15
output, 10-31
List-directed statements
ACCEPT, 10-28
PRINT and TYPE, 10-38
READ, 10-13, 10-15
WRITE, 10-29, 10-31
Listing header
directive specifying subtitle for, 14-38
directive specifying title for, 14-38
Lists
I/O control, 10-3
implied-DO
in DATA statements, 5-24
items in I/O, 10-9
Literal constant, 3-1
LITTLE_ENDIAN value
for INQUIRE (CONVERT), 12-11
for OPEN (CONVERT), 12-28
LLE function, 9-87
LLT function, 9-88
LOC function, 9-89
using with integer pointers, B-12
Local properties
of intrinsic functions, 8-40
Local scope
names having, 15-2
Locked records
freeing, 12-42
LOG function, 9-89
LOG10 function, 9-90
Logarithm
function returning common, 9-90
function returning natural, 9-89
Logical AND
function performing, 9-65
Logical assignment statements, 4-18
Logical complement
function returning, 9-111
Logical conversion
function performing, 9-91
LOGICAL data type, 3-3, 3-14
constants, 3-14
default kind, 3-14
in type declaration statements, 5-2, 5-6
See also LOGICAL(4)
storage, 15-14
Logical editing, 11-25
Logical expressions, 4-8
evaluation of subexpressions in, 4-10
extensions to, F-2
order of evaluation in, 4-9
LOGICAL function, 9-91
Logical I/O units, 10-4
See also I/O units
Logical IF statement, 7-31
See also IF statement
Logical operands, 4-8
See also Logical expressions
Logical operations, 4-9
data types resulting from, 4-9
functions performing, 9-16
Logical operators, 4-8
Logical shift
function performing, 9-79
Logical values
transferring, 11-25
LOGICAL(1)
constants, 3-14
storage requirements, 15-14
LOGICAL(2)
constants, 3-14

storage requirements, 15-14
LOGICAL(4)
constants, 3-14
storage requirements, 15-14
LOGICAL(8)
constants, 3-14
storage requirements, 15-14
Loop control, 7-14
DO WHILE, 7-17
iteration count, 7-17
simple, 7-17
Loop control iteration, 7-17
LOOP COUNT directive, 14-25
Loop directives
DISTRIBUTE POINT, 14-18
IVDEP, 14-23
LOOP COUNT, 14-25
PARALLEL and NOPARALLEL, 14-31
PREFETCH and NOPREFETCH, 14-32
SWP and NOSWP (i64), 14-37
UNROLL and NOUNROLL, 14-39
VECTOR ALIGNED and VECTOR UNALIGNED (i32), 14-39
VECTOR ALWAYS and NOVECTOR (i32), 14-40
VECTOR NONTEMPORAL (i32), 14-41
Loop iteration, 7-14, 7-17
Loops, 7-14
DO
nested, 7-19
skipping, 7-24
terminating, 7-24
IF
flow of control in, 7-27
Lower bounds
function returning, 9-83
Lowercase letters
in character set, 2-5
treatment on compiler, 2-6
LSHFT function, 9-79
LSHIFT function, 9-79

M

Main program, 8-1, 8-2
as a program unit, 2-1
MALLOC function, 9-91
using with integer pointers, B-12
Manipulation
functions for array, 9-4
functions for bit, 9-4
functions for numeric, 9-4
Mantissa in real model, D-3
Many-one array sections, 3-44, 4-20
Map declarations, B-19
MAP statement, B-19
using to make record fields equivalent, B-22
MASK
keyword in intrinsics, 9-3
See also Mask expressions
Mask expressions
function combining arrays using, 9-98
function counting true elements using, 9-40
function determining all true using, 9-23
function determining any true using, 9-25
function finding location of maximum value using, 9-95
function finding location of minimum value using, 9-100
function packing array using, 9-113
function returning maximum value of elements using, 9-97
function returning minimum value of elements using, 9-102
function returning product of elements using, 9-116
function returning sum of elements using, 9-140
function unpacking array using, 9-148
in ELSEWHERE, 4-23
in FORALL, 4-26
in intrinsics, 9-3
in WHERE, 4-23
Masked-array assignment statements (WHERE), 4-23
generalization of (FORALL), 4-26
MASTER directive, 14-60
example of, 14-60
Mathematical functions, 9-4
MATMUL function, 9-92
Matrix multiplication
function performing, 9-92

MAX function, 9-93
MAX0 function, 9-93
MAX1 function, 9-93
MAXEXPONENT function, 9-95
Maximum exponent
    function returning, 9-95
Maximum value
    function returning, 9-93
    function returning location of, 9-95
    of array elements
        function returning, 9-97
MAXLOC function, 9-95
MAXREC specifier
    in OPEN statements, 12-33
MAXVAL function, 9-97
MBCS routines, E-8
MCLOCK function, 9-98
Memory
    function allocating, 9-91
    subroutine freeing allocated, 9-62
Memory cache
    function returning size of a level in, 9-34
Memory location
    OpenMP directive to dynamically update, 14-52
MERGE function, 9-98
MESSAGE directive, 14-26
Metacommands
    See General compiler directives
MIL standard, 1-2
MIN function, 9-99
MIN0 function, 9-99
MIN1 function, 9-99
MINEXPONENT function, 9-100
Minimum exponent
    function returning, 9-100
Minimum value
    function returning, 9-99
    function returning location of, 9-100
    of array elements
        function returning, 9-102
MINLOC function, 9-100
Minus operator, 4-2
    precedence of, 4-11
MINVAL function, 9-102
Miscellaneous run-time routines, E-22
Mixed-mode expressions, 4-5, 4-8
MM_PREFETCH subroutine, 9-103
MOD function, 9-105
MODE specifier
    in INQUIRE statements, 12-14
    in OPEN statements, 12-33
Models for data representation, D-1
    bit, D-4
    integer, D-2
    real, D-3
MODULE PROCEDURE statement, 8-47
    example of, 15-6
Module procedures, 8-4
    definition of, 8-1
    examples of, 8-5, 15-6
    in interface blocks, 8-47
    scope of, 15-2
Module references, 8-7
MODULE statement, 8-4
    example of, 8-5, 15-6
    See also Module procedures
Module subprograms
    introducing in program unit, 8-53
    providing entry points within, 8-53
Modules, 2-1, 8-1, 8-4
    accessibility of entities in, 5-51, 8-7
    containing interface blocks, 8-5
    examples of, 8-5, 8-9
    references to, 8-7
    See also your user's guide
    selecting entities in (USE), 8-8
    specification part of, 8-4
    terminating, 7-25
Modulo
    function returning, 9-106
MODULO function, 9-106
MULT_HIGH function (i64), 9-106
Multibyte Character Set routines, E-8

Multidimensional arrays
constructing, 3-46, 9-127
conversion between vectors and, 9-113, 9-148
declaring adjustable, 5-13
storage of, 3-39
Multiplication operator (*), 4-2
precedence of, 4-11
MVBITS subroutine, 9-107

N

NAME specifier
in INQUIRE statements, 12-14
in OPEN statements, 12-33
interaction with OPEN (FILE), 12-14
Named common blocks
agreement of data types in, 5-22
establishing and initializing values in, 8-10
OpenMP directive defining as local to a thread, 14-69
See also Common blocks
Named constants, 5-48
definition of, 3-1
scope of, 15-2
Named constructs
scope of, 15-2
Named control constructs, 7-1
CASE, 7-9
DO, 7-15
FORALL, 4-26
IF, 7-26
WHERE, 4-23
NAMED specifier
in INQUIRE statements, 12-14
Namelist external records, 10-18
alternative form for, B-11
Namelist formatting, 10-6
for READ statements, 10-18
for WRITE statements, 10-33
input, 10-18
output, 10-33
Namelist group, 5-45
accessibility of, 5-45
examples of, 5-46
prompting for information about, 10-20
variables in, 5-45
Namelist group names
scope of, 15-2
Namelist input
comments in, 10-20
Namelist specifier, 10-6
NAMELIST statement, 5-45
examples of, 5-46, 10-21
Namelist statements
ACCEPT, 10-28
PRINT and TYPE, 10-38
READ, 10-13, 10-18
WRITE, 10-29, 10-33
Names, 2-4
associating with constant value, 5-48
associating with group, 5-45
association of, 15-9
association of arguments, 15-10
association of use and host, 15-11
constants with, 3-1
containing dollar sign, 2-4
explicit typing of variable, 3-34
extension to characters in, F-1
extension to length of, F-1
implicit type if first character is $, 3-35
implicit typing of variable, 3-35
in FORTRAN-66 EXTERNAL statement, B-7
in PARAMETER statement, 5-48
length allowed, 2-4
of external and dummy procedures as actual arguments, 5-38
of intrinsic procedures as actual arguments, 5-43
of procedures
established as generic, 15-5
established as specific, 15-7
nonestablished, 15-8
overriding default data typing of, 5-39
rules for constants with, 5-48
scope of, 15-1
See also your user's guide
uniqueness within programs, 2-4
with global scope, 15-1
with local scope, 15-2
with statement scope, 15-2

NaN values
function testing for, 9-82
See also your user's guide
NARGS function, 9-108
National Language Support routines, E-8
NATIVE value
for INQUIRE (CONVERT), 12-11
for OPEN (CONVERT), 12-28
Nearest different number
function returning, 9-110
NEAREST function, 9-110
Nearest integer
function returning, 9-110
Negation
logical, 4-8
Nested constructs
DO, 7-19
IF, 7-29
Nested DO
construct, 7-21
control transfers in, 7-21
loops, 7-19
Nested format specifications, 11-40
Nested implied-DO lists
in I/O lists, 10-12
NEW value
for OPEN (STATUS), 12-39
NEXTREC specifier
in INQUIRE statements, 12-15
NINT function, 9-110
NLS routines, E-8
NML specifier, 10-6
NO_ARG_CHECK
option for ATTRIBUTES directive, 14-14
NOALIGN
OPTIONS directive option, 14-27
NOCHECK
OPTIONS statement option, 13-3
NODECLARE directive, 14-16
NOEXTEND_SOURCE
OPTIONS statement option, 13-3
NOF77
OPTIONS statement option, 13-3
NOFREEFORM directive, 14-20
NOI4
OPTIONS statement option, 13-3
NOINLINE
option for ATTRIBUTES directive, 14-14
NOLIST option
in INCLUDE statement, 13-1
NOMIXED_STR_LEN_ARG
option for ATTRIBUTES directive, 14-15
Nonadvancing I/O, 10-8, 10-9
See also your user's guide
Nonblock DO construct, 7-15
examples of, 7-16
Nondecimal numeric constants, 3-28
binary, 3-28
data type of, 3-31
hexadecimal, 3-29
Hollerith, 3-30
octal, 3-29
Nonelemental intrinsic procedures, 9-1
Nonexecutable statements, 2-2
Non-Fortran procedures
argument list functions for, 8-43
in argument list
defaults for, 8-43
references to, 8-43
See also your user's guide
referencing with %LOC function, 8-44
referencing with %REF function, 8-43
referencing with %VAL function, 8-43
Nonnative floating-point formats, 12-11, 12-28
NOPARALLEL loop directive, 14-31
NOPREFETCH loop directive, 14-32
NOSTRICT directive, 14-36
NOSWP directive (i64), 14-37
NOT function, 9-111
NOUNROLL directive, 14-39
NOVECTOR directive (i32), 14-40
example of, 14-40

NOWAIT clause
effect on implied FLUSH directive, 14-59
effect with REDUCTION clause, 14-48
in END DO directive, 14-57
in END SECTIONS directive, 14-67
in END SINGLE directive, 14-69

NULL function, 9-112

NULL value
for INQUIRE (BLANK), 12-10
for OPEN (BLANK), 12-26

Null values
in list-directed records, 10-16
in namelist records, 10-19

NULLIFY statement, 6-8
examples of, 6-8

NUM_THREADS clause
in PARALLEL directive, 14-63
in PARALLEL DO directive, 14-65

NUMARG function, 9-66

NUMBER specifier
in INQUIRE statements, 12-15

Numerals
in character set, 2-5

Numeric and logical type declaration statements, 5-6

Numeric assignment statements, 4-17

Numeric constants
alternative ways to represent, 3-28, 3-29, 3-30
complex, 3-11
integer, 3-4
nondecimal, 3-28
real, 3-7

Numeric editing
general rules for, 11-8

Numeric expressions, 4-2
effects of parentheses within, 4-4
examples of, 4-3
in relational expressions, 4-7
operator precedence in, 4-3
order of evaluation in, 4-3
ranking of data types in, 4-5
rules for typing of, 4-5
using in FORMAT statements, 11-41

Numeric functions
categories of, 9-4
models defining, D-1

Numeric models
integer, D-2
querying parameters in, 9-64, 9-114, 9-144
real, D-3

Numeric nondecimal constants, 3-28
data type of, 3-31
See also Nondecimal numeric constants

Numeric operators
in expressions, 4-2

Numeric storage unit, 15-13

Numeric values
size limits for A editing, 11-28

O

O edit descriptor, 11-12
input processing, 11-12
output processing, 11-12

OBJCOMMENT directive, 14-26
example of, 14-27

Object file
directive specifying library search path for, 14-26
See also Object module
See also your user's guide

Object libraries
searching for block data program units in, 8-11

Object module
identifying with compiler directives, 14-20
See also Object file

Obsolescent features
in Fortran 90, A-3
in Fortran 95, A-2

Octal constants, 3-29
alternative syntax for, B-10
assigning with DATA statement, 5-25
data type assignments of, 3-31
examples of, 3-29

Octal values
transferring, 11-12

OLD value
for OPEN (STATUS), 12-39
OMP_SCHEDULE environment variable, 14-56
ONLY keyword
in USE statement, 8-8
OPEN statements, 12-20
ACCESS specifier in, 12-24
ACTION specifier in, 12-25
ASSOCIATEVARIABLE specifier in, 12-25
BLANK specifier in, 12-26
BLOCKSIZE specifier in, 12-26
BUFFERCOUNT specifier in, 12-26
BUFFERED specifier in, 12-27
CARRIAGECONTROL specifier in, 12-28
CONVERT specifier in, 12-28
DEFAULTFILE specifier in, 12-30
defaults for specifiers, 12-20
DELIM specifier in, 12-30
DISPOSE specifier in, 12-31
examples of, 12-24
FILE specifier in, 12-20, 12-32
FORM specifier in, 12-32
general description of, 12-20
general rules for, 12-23
IOFOCUS specifier in (W*32, W*64), 12-33
MAXREC specifier in, 12-33
MODE specifier in, 12-33
NAME specifier in, 12-33
ORGANIZATION specifier in, 12-34
PAD specifier in, 12-34
POSITION specifier in, 12-35
READONLY specifier in, 12-35
RECL specifier in, 12-36
RECORDSIZE specifier in, 12-37
RECORDTYPE specifier in, 12-37
See also your user's guide
SHARE specifier in (W*32, W*64), 12-38
SHARED specifier in, 12-39
STATUS specifier in, 12-39
TITLE specifier in (W*32, W*64), 12-40
TYPE specifier in, 12-40
USEROPEN specifier in, 12-40
OPENED specifier
in INQUIRE statements, 12-15
OpenMP* Fortran clauses and options
COPYIN, 14-44
COPYPRIVATE, 14-44
DEFAULT, 14-45
DEFAULT NONE, 14-45
DEFAULT PRIVATE, 14-45
DEFAULT SHARED, 14-45
FIRSTPRIVATE, 14-46
IF, 14-62
LASTPRIVATE, 14-46
NOWAIT, 14-57, 14-67, 14-68
NUM_THREADS, 14-63
ORDERED, 14-56
PRIVATE, 14-46
REDUCTION, 14-47
SCHEDULE, 14-56
SHARED, 14-49
OpenMP* Fortran compiler directives, 14-42
ATOMIC, 14-52
BARRIER, 14-53
categories of, 14-43
conditional compilation of, 14-49
CRITICAL, 14-54
data scope attribute clauses for, 14-44
DO, 14-55
examples of, 14-43
FLUSH, 14-59
MASTER, 14-60
nesting and binding rules for, 14-50
ORDERED, 14-61
PARALLEL, 14-62
PARALLEL DO, 14-64
PARALLEL SECTIONS, 14-66
SECTION, 14-67
SECTIONS, 14-67
SINGLE, 14-68
syntax of prefix, 14-1
THREADPRIVATE, 14-69
OpenMP* Fortran library routines, E-25
Operands, 4-1
and binary operators, 4-3
and unary operators, 4-3
operating on pair of operands, 4-3
operating on single operand, 4-3

Operations
character, 4-6
complex, 4-6
conversion to higher precision, 4-6
defined, 4-10, 8-50
integer
conventions for determining, 4-6
numeric, 4-2
real
conventions for determining, 4-6
relational, 4-7
OPERATOR interface specifier
for functions, 4-10, 8-47, 8-50
Operators, 4-1
character, 4-6
defined, 4-10
exponential
in initialization expressions, 4-13
extended intrinsic
properties of, 8-50
logical, 4-8
numeric, 4-2
operating on pair of operands, 4-3
operating on single operand, 4-3
precedence in
logical expressions, 4-11
numeric expressions, 4-3
relational expressions, 4-7
scope of, 15-2
Optimization
preventing with VOLATILE statement, 5-57
See also your user's guide
specified by ATOMIC directive, 14-52
specified by SWP and NOSWP directives (i64), 14-37
specified by UNROLL and NOUNROLL directives, 14-39
Optional arguments, 8-32
examples of, 8-33
function returning presence of, 9-115
interface for, 8-33
OPTIONAL attribute and statement, 5-46, 8-32
attributes compatible with, 5-5
examples of, 5-47
OPTIONS directive, 14-27
example of, 14-29
OPTIONS statement, 13-3
examples of, 13-4
position in program unit, 13-4
OR function, 9-76
Order of
elements in an array, 3-39
statements, 2-2
subscript progression, 3-39
ORDERED clause, 14-56
in DO directive, 14-55
in PARALLEL DO directive, 14-65
ORDERED directive, 14-61
example of, 14-62
ORGANIZATION specifier
in INQUIRE statements, 12-16
in OPEN statements, 12-34
Output statements, 10-29
PRINT and TYPE, 10-38
REWRITE, 10-39
WRITE, 10-29

P

P edit descriptor, 11-34
input processing, 11-34
output processing, 11-35
PACK directive, 14-30
example of, 14-30
PACK function, 9-113
Packed array
function creating, 9-113
PAD specifier
in INQUIRE statements, 12-16
in OPEN statements, 12-34
Padding
blank, 11-29, 12-34
Parallel compiler directives, 14-42
categories of, 14-43
data scope attribute clauses for, 14-44
examples of, 14-43
See also OpenMP* Fortran compiler directives
PARALLEL directive
general for loops, 14-31

OpenMP* Fortran, 14-62
examples of, 14-51, 14-53, 14-60, 14-64

PARALLEL DO directive, 14-64
examples of, 14-53, 14-65

Parallel execution library routines, E-25

Parallel region
copying data from the master thread in, 14-44
OpenMP directive defining, 14-62, 14-64, 14-66
specifying scope for variables in, 14-45

PARALLEL SECTIONS directive, 14-66
example of, 14-66

PARAMETER attribute and statement, 5-48
attributes compatible with, 5-5
examples of, 5-49

PARAMETER statement
alternate form of, B-8
using in record structure declarations, B-15

Parentheses
effect in
character expressions, 4-7
logical expression, 4-9
numeric expressions, 4-4
using to specify evaluation order, 4-4

Partial storage association, 15-15

Pathname
See also File name
See also your user's guide
specifying default for OPEN statement, 12-30
specifying for INQUIRE statement, 12-7
specifying for OPEN statement, 12-32

PAUSE statement, 7-32
alternative for, A-3
default message for, 7-32
effect on Linux systems, 7-33
effect on Windows systems, 7-33
examples of, 7-33

Performance
directives affecting, 14-14, 14-27, 14-39, 14-40, 14-41
See also your user's guide

Platform
description of, xxxi
labels, xxxi

Plus operator, 4-2
precedence of, 4-11

Pointer arguments, 8-34
requiring explicit interface, 8-46

Pointer assignment statements, 4-22
examples of, 4-22

Pointer association, 15-12
function returning, 9-28

POINTER attribute and statement, 5-50
attributes compatible with, 5-5
examples of, 5-51

POINTER statement
integer form of, B-12

Pointer targets, 5-55
dynamically allocating and deallocating, 6-1
See also Targets

Pointers
allocation of targets, 6-2, 6-4
array, 5-16
as arguments, 8-34
as automatic or static variables, 5-20
as variables, 4-16
assigning values to targets of, 4-16
assignment of, 4-22
associating with targets, 5-55
association of, 4-22, 8-34
association status of, 15-12
association with targets, 4-16
CRAY-style, B-12
deallocation of targets, 6-5, 6-7
disassociating from targets, 4-22, 6-1
examples of, 5-51
Fortran 95/90, 5-50
function returning association status of, 9-28
function returning disassociated, 9-112
in I/O lists, 10-9
initializing, 9-112
integer, B-12
nullifying, 4-22, 6-8
referencing, 5-50
volatile, 5-57

POPCNT function, 9-114

POPPAR function, 9-114

Portability
features for older Fortran programs, B-1
routines for, E-2
See also your Libraries Reference
Portability routines, E-2
POSITION specifier
in INQUIRE statements, 12-16
in OPEN statements, 12-35
Positional editing (T,TL,TR,X), 11-31
T, 11-31
TL, 11-32
TR, 11-32
X, 11-32
POSIX library routines, E-10
Precedence of operators
effect of parentheses on, 4-4
in logical expressions, 4-9, 4-11
in numeric expressions, 4-3
in relational expressions, 4-7
Precision
function converting to higher, 9-51, 9-119
function returning, 9-114
PRECISION function, 9-114
Predefined typing rules
for variables, 3-34
PREFETCH loop directive, 14-32
Prefetches of data
directive enabling, 14-32
subroutine performing, 9-103
PRESENT function, 5-47, 8-33, 9-115
Pretested DO statement, 7-23
Primary, 4-1
for array constructors
in initialization expressions, 4-12
in specification expressions, 4-14
PRINT statement, 10-38
example of, 10-38
PRINT value
for CLOSE statements, 12-3
for OPEN (DISPOSE), 12-31
PRINT/DELETE value
for CLOSE statements, 12-3
for OPEN (DISPOSE), 12-31
Printable characters, 2-6
PRIVATE attribute and statement, 5-51
attributes compatible with, 5-5
examples of, 5-52
in derived-type definition, 3-20
PRIVATE clause, 14-46
in DO directive, 14-55
in PARALLEL directive, 14-62
in PARALLEL DO directive, 14-65
in PARALLEL SECTIONS directive, 14-66
in SECTIONS directive, 14-67
in SINGLE directive, 14-68
Procedure arguments, 8-30
defaults for %VAL and %REF functions, 8-43
See also Arguments
Procedure interface, 8-45
blocks, 8-46
defining generic assignment, 8-51
defining generic names, 8-49
defining generic operators, 8-50
definition of, 8-2
modules containing, 8-5
See also your user's guide
when explicit is required, 8-46
Procedure references
requiring procedures with explicit interface, 8-46
resolving, 15-5
resolving established
generic, 15-5
specific, 15-7
resolving nonestablished, 15-8
unambiguous, 15-4
Procedures
arguments in, 8-30
declaring external, 5-38
declaring intrinsic, 5-43
defining generic assignment for, 8-51
defining generic names for, 8-49
defining generic operators for, 8-50
directive specifying properties of, 14-5
dummy, 8-37
See also Dummy procedures
elemental user-defined, 8-17
established as generic, 15-5
established as specific, 15-7

external, 8-28
external and dummy
using as actual arguments, 5-38
interfaces in, 8-45
internal, 8-29
intrinsic
using as actual arguments, 5-43
kinds of, 8-1
module, 8-4
nonestablished, 15-8
non-Fortran
%LOC function for, 8-44
argument list functions for, 8-43
pure, 8-14
recursive, 8-13
references to non-Fortran, 8-43
requiring explicit interface, 8-46
See also Functions
See also Subroutines

Processor time
subroutine returning, 9-42

PRODUCT function, 9-116

Product of array elements
function returning, 9-116

Program execution, 8-1
stopping, 7-35
subroutine terminating, 9-59
temporary suspension of, 7-32

PROGRAM statement, 8-2

Program structure
overview of, 2-1

Program unit
block data, 2-1, 8-10
effect of using DATA statement in, 8-11
common blocks in, 5-21
definition of, 2-1
effect of EQUIVALENCE statement on, 5-29
external subprograms, 8-12, 8-28
kinds of, 8-1
main, 8-2
modules, 8-4
order of statements in, 2-2
scope of, 15-1, 15-2
terminating, 7-25

Program unit scoping, 15-1
See also Scoping unit

Prompting
for namelist group information, 10-20

PSECT directive, 14-33
ALIGN option, 14-33
NOWRT option, 14-33
WRT option, 14-33

Pseudorandom number generator
routines, 9-120, 9-121, 9-124

Pseudorandom numbers
function returning next in sequence of, 9-120
subroutine computing single-precision, 9-124
subroutine returning, 9-121
subroutine to change or query generator of, 9-122

PUBLIC attribute and statement, 5-51
attributes compatible with, 5-5
examples of, 5-52

PURE prefix, 8-14
in FUNCTION statements, 8-18
in SUBROUTINE statements, 8-25

Pure procedures, 8-14
example of, 8-16
functions as, 8-18
in FORALL statement, 4-27
in interface blocks, 8-48
INTENT for, 8-15
subroutines as, 8-25

Q

Q editing, 11-38

QABS function, 9-18

QACOS function, 9-20

QACOSD function, 9-20

QACOSH function, 9-21

QASIN function, 9-26

QASIND function, 9-27

QASINH function, 9-27

QATAN function, 9-29

QATAN2 function, 9-30

QATAN2D function, 9-31

QATAND function, 9-31

QATANH function, 9-32
QCMPLX function, 9-117
QCONJG function, 9-37
QCOS function, 9-38
QCOSD function, 9-38
QCOSH function, 9-39
QCOTAN function, 9-39
QCOTAND function, 9-40
QDIM function, 9-49
QERF function, 9-57
QERFC function, 9-57
QEXP function, 9-59
QEXT function, 9-117
QEXTD function, 9-117
QFLOAT function, 9-118
QIMAG function, 9-22
QINT function, 9-23
QLOG function, 9-89
QLOG10 function, 9-90
QMAX1 function, 9-93
QMIN1 function, 9-99
QMOD function, 9-105
QNINT function, 9-25
QNUM function, 9-119
QREAL function, 9-119
QSIGN function, 9-134
QSIN function, 9-135
QSIND function, 9-135
QSINH function, 9-136
QSQRT function, 9-139
QTAN function, 9-142
QTAND function, 9-143
QTANH function, 9-143
Quad-precision product
    function producing, 9-51
Qualification
    of variable names in record structures, B-23
Question mark character (?)
    as namelist prompt, 10-20
QuickWin routines (W*32, W*64), E-14
Quotation mark character (")
    as delimiter for character strings, 3-15
    See also Character constants
    See also Character strings
QUOTE value
    for INQUIRE (DELIM), 12-12
    for OPEN (DELIM), 12-31

R

Radix
    function returning, 9-120
    in integer model, D-2
    in real model, D-3
RADIX function, 9-120
RAN function, 9-120
Random number generator
    function, 9-120
    subroutine, 9-121
    subroutine querying the seed for, 9-122
Random numbers
    function returning next in sequence of, 9-120
    subroutine computing as single-precision, 9-124
    subroutine returning, 9-121
RANDOM_NUMBER subroutine, 9-121
    See also RANDOM_SEED subroutine
RANDOM_SEED subroutine, 9-122
    See also RANDOM_NUMBER subroutine
RANDU subroutine, 9-124
Range
    for case values, 7-10, 7-11
    for character length, 3-16
    for H editing, 11-39
    for parameter in control edit descriptors, 11-30
    for parameters in data edit descriptors, 11-7
    for repeat specifications in data edit descriptors, 11-7
    for scale factor, 11-34
RANGE function, 9-124
Rank
    definition of, 3-36
Ranking
    of data types, 4-5

READ specifier
in INQUIRE statements, 12-17
READ statements, 10-13
compared to DECODE statement, B-3
compared to FIND statement, B-5
direct access, 10-24
example of, 10-25, 10-26
formatted, 10-24, 10-25
forms of, 10-24
unformatted, 10-24, 10-26
internal, 10-26
example of, 10-27
form of, 10-26
list-directed, 10-13, 10-15
example of, 10-17
namelist, 10-13, 10-18
example of, 10-21
nondelimited character strings in, 10-23
sequential, 10-13
formatted, 10-13, 10-15
forms of, 10-13
unformatted, 10-14, 10-23
READ value
for INQUIRE (ACTION), 12-9
for OPEN (ACTION), 12-25
READONLY specifier
in OPEN statements, 12-35
READWRITE specifier
in INQUIRE statements, 12-17
READWRITE value
for INQUIRE (ACTION), 12-9
for OPEN (ACTION), 12-25
Real constants
REAL(16) type, 3-10
REAL(4) type, 3-8
REAL(8) type, 3-9
Real conversion
function performing, 9-125
REAL data type, 3-6
constants, 3-7
double-precision, 3-9
quad-precision, 3-10
single-precision, 3-8
conversion in expressions, 4-6
default kind, 3-6, 3-7
directive specifying default kind, 14-35
function converting to double-precision, 9-46
function converting to quad-precision, 9-117
function converting to single-precision, 9-125
general rules for, 3-7
in type declaration statements, 5-2, 5-6
kind parameters for, 3-2
model sets for, D-3
See also REAL(4)
See also your user's guide
storage, 15-14
Real data types, 3-6 thru 3-10
REAL directive, 14-35
example of, 14-35
Real DO control
alternative for, A-3
Real editing (F,E,EN,ES,D,G), 11-14
D, 11-16
E, 11-16
EN, 11-19
ES, 11-20
F, 11-15
G, 11-22
scale factor in, 11-34
REAL function, 9-125
Real model, D-3
function returning exponent part in, 9-60
function returning fractional part in, 9-61
function returning largest number in, 9-64
function returning number closest to unity in, 9-56
function returning smallest number in, 9-144
Real numbers
directive specifying default kind, 14-35
function returning absolute spacing of, 9-138
function returning ceiling of, 9-35
function returning class of IEEE, 9-61
function returning difference between, 9-49
function returning floor of, 9-60
function returning fractional part for model of, 9-132
function returning scale of model for, 9-128
function to determine nearest whole number, 9-25
function to truncate, 9-23
See also Real constants
See also REAL data type

Real values
transferring, 11-14, 11-15
REAL(16)
constants, 3-10
storage requirements, 15-14
REAL(4)
constants, 3-7, 3-8
storage requirements, 15-14
REAL(8)
constants, 3-7, 3-9
See also DOUBLE PRECISION data type
storage requirements, 15-14
REAL*16 constants
See REAL(16)
REAL*4 constants
See REAL(4)
REAL*8 constants
See REAL(8)
Real-time clock
subroutine returning data from, 9-141
subroutine returning data on, 9-44
REC specifier, 10-6
Reciprocal
function returning, 9-128
RECL specifier
default record lengths for, 12-37
in INQUIRE statements, 12-17
in OPEN statements, 12-36
maximum record lengths for, 12-36
Record number
identifying for data transfer, 10-6
Record specifier, 10-6
alternate form of, B-10
RECORD statement, B-21
Record structures, B-14
aggregate assignment in, B-24
example of, B-25
declarations in, B-14
directive modifying alignment of fields in, 14-27
directive specifying starting address of items in, 14-30
examples of, B-16, B-20, B-23
field names in, B-14, B-18
passing as arguments, B-15
qualifying variable names in, B-23
RECORD statement in, B-21
references to fields in, B-22
examples of, B-23
rules in using scalar field reference, B-22
statements that can use names of, B-21
substructure declarations in, B-18
type declarations in, B-18
union declarations in, B-19
using %FILL in, B-18
Record-oriented I/O, 10-9
Records
default length for OPEN(RECL), 12-37
default types upon file connection, 12-38
definition of, 10-1
deleting from relative files, 12-4
detecting deleted, 12-5
endfile, 10-2
external
See External records
formatted, 10-1
freeing locked (UNLOCK), 12-42
kinds of, 10-1
maximum length for OPEN(RECL), 12-36
unformatted, 10-2
RECORDSIZE specifier
in OPEN statements, 12-37
See also RECL specifier
RECORDTYPE specifier
defaults for, 12-38
in INQUIRE statements, 12-18
in OPEN statements, 12-37
Recursion, 8-2, 8-13
in functions, 8-18
in subroutines, 8-25
See also your user's guide
with automatic variables, 5-19
RECURSIVE keyword
in FUNCTION statements, 8-18
in OPTIONS statements, 13-3
in subprograms, 8-13
in SUBROUTINE statements, 8-24
Recursive procedures, 8-13
REDUCTION clause, 14-47
in DO directive, 14-55

in PARALLEL directive, 14-62
in PARALLEL DO directive, 14-65
in PARALLEL SECTIONS directive, 14-66
in SECTIONS directive, 14-67

REFERENCE
option for ATTRIBUTES directive, 14-15

References
module, 8-7
See also Function references
See also Subroutine references
to elemental intrinsic procedures, 8-42
to generic intrinsic procedures, 8-39
to generic procedures, 8-38
to non-Fortran procedures, 8-43

Relational expressions, 4-7

Relational operators, 4-7
avoiding use as field names, B-23

Relative files
associating with logical unit numbers, B-1
defining size and structure of, B-1
deleting records from, 12-4
detecting deleted records in, 12-5
freeing a record in, 12-42
See also your user's guide

Relative spacing
function returning reciprocal of, 9-128

RELATIVE value
for INQUIRE (ORGANIZATION), 12-16
for OPEN (ORGANIZATION), 12-34

Remainder
function returning, 9-105

REPEAT function, 9-126

Repeat specification
for control edit descriptors, 11-30
for data edit descriptors, 11-2, 11-7
for groups of descriptors, 11-40
for slash editing, 11-36
for string edit descriptors, 11-39
in DATA statements, 5-24

Repeated execution
See Loops

REPLACE value
for OPEN (STATUS), 12-39

Replicated arrays
function creating, 9-138

RESHAPE function, 3-46, 9-127

Resolving references
generic, 15-5
example of, 15-6
nonestablished, 15-8
specific, 15-7

Restricted expressions
definition of, 4-13
See also Specification expressions

RESULT keyword
in ENTRY statements, 8-53, 8-55
in FUNCTION statements, 8-18, 8-23

Result variables
in ENTRY statements, 8-53
in FUNCTION statements, 8-18, 8-23
value of, 8-19
requiring explicit interface, 8-46

RETURN statement, 7-33
effect in subprograms, 7-34
examples of, 7-34
retaining data after execution of, 5-54

Reversion
format, 11-43

REWIND statement, 12-41
examples of, 12-41

REWIND value
for INQUIRE (POSITION), 12-16
for OPEN (POSITION), 12-35

REWRITE statements, 10-39
example of, 10-40

Right shift
function performing arithmetic, 9-52, 9-77, 9-133
function performing circular, 9-78
function performing logical, 9-81

RNUM function, 9-128

RRSPACING function, 9-128

RSHFT function, 9-79

RSHIFT function, 9-79

Run-time formats, 11-4

Run-time library module routines, E-1

RUNTIME schedule type, 14-56

S

S edit descriptor, 11-33

SAVE attribute and statement, 5-54
- attributes compatible with, 5-5
- examples of, 5-55

SAVE statement
- effect of including common block in, 5-55

SAVE value
- for CLOSE statements, 12-3
- for OPEN (DISPOSE), 12-31

Scalar expressions
- assigning to array variables, 4-20

Scalar field references, B-22
- examples of, B-23

Scalars
- as array elements, 3-35, 3-38
- as results of expressions, 4-1
- as storage units, 15-13
- as variables, 3-33
- explicit typing of, 3-34
- implicit typing of, 3-35
- in array assignment statements, 4-20
- in elemental intrinsic procedures, 9-1
- in numeric expressions, 4-2
- in specification expressions, 4-13
- in structure constructors, 3-26

Scale factor editing, 11-34

SCALE function, 9-128

SCAN function, 9-129

SCHEDULE clause, 14-56
- in DO directive, 14-56
- in PARALLEL DO directive, 14-65
- types
  - DYNAMIC, 14-56
  - GUIDED, 14-56
  - RUNTIME, 14-56
  - STATIC, 14-56

Schedule types
- DYNAMIC, 14-56
- GUIDED, 14-56
- RUNTIME, 14-56
- STATIC, 14-56

Scientific notation
- descriptor for (ES), 11-20

Scope, 15-1 thru 15-9
- definition of, 15-1
- of argument keywords in procedures, 15-2
- of assignment symbol, 15-2
- of common blocks, 15-2
- of components of derived types, 15-2
- of defined assignments, 15-2
- of defined operators, 15-2
- of derived types, 15-2
- of dummy procedures, 15-2
- of external procedures, 15-2
- of generic identifiers, 15-2
- of I/O unit numbers, 15-2
- of internal procedures, 15-2
- of intrinsic assignments, 15-2
- of intrinsic operators, 15-2
- of intrinsic procedures, 8-39, 15-2
- of labels, 15-2
- of module procedures, 15-2
- of named constants, 15-2
- of named constructs, 15-2
- of namelist group names, 15-2
- of names, 15-1
- of operators, 15-2
- of program units, 15-2
- of statement functions, 15-2
- of unambiguous procedure references, 15-4

Scoping unit, 15-2
- definition of, 15-2
- rules for multiple USE statements in modules, 8-8
- statements not allowed in, 2-3

SCRATCH value
- for OPEN (STATUS), 12-39

SECNDS function, 9-130

SECTION directive, 14-67

Section subscript list, 3-35, 3-41

SECTIONS directive, 14-67
- example of, 14-68

Sections of arrays, 3-41

Segmented record
- definition of, 12-38

SEGMENTED value
for INQUIRE (RECORDTYPE), 12-18
for OPEN (RECORDTYPE), 12-37
SELECT CASE statement, 7-9
branching to, 7-13
SELECTED_INT_KIND function, 9-130
SELECTED_REAL_KIND function, 9-131
Semicolon character
as source form statement separator, 2-7
Separating
external fields, 11-29
statements in source form, 2-7
Sequence derived types, 3-20, 3-22
storage of, 15-15
Sequence number field, 2-13
restriction in tab-format source, 2-14
SEQUENCE statement, 3-20, 3-22
Sequential access
definition of, 10-2
specifying, 12-24
Sequential files
freeing a record in, 12-42
positioning
after an end-of-file record, 12-5
at beginning of preceding record, 12-2
at the beginning of the file, 12-41
See also your user's guide
Sequential I/O statements
READ, 10-13
forms of, 10-13
WRITE, 10-29
forms of, 10-29
SEQUENTIAL specifier
in INQUIRE statements, 12-18
SEQUENTIAL value
for INQUIRE (ACCESS), 12-9
for INQUIRE (ORGANIZATION), 12-16
for OPEN (ACCESS), 12-25
for OPEN (ORGANIZATION), 12-34
SET_EXPONENT function, 9-132
SHAPE function, 9-132
Shape of array, 3-36
declaring, 5-10
function constructing different, 9-127
function returning, 9-132
in assignment statements, 4-20
statement defining, 5-17, 5-27
when determined, 6-3
SHARE specifier (W*32, W*64)
in INQUIRE statements, 12-19
in OPEN statements, 12-38
SHARED clause, 14-49
in PARALLEL directive, 14-62
in PARALLEL DO directive, 14-65
in PARALLEL SECTIONS directive, 14-66
Shared DO termination
alternative for, A-4
SHARED specifier
in OPEN statements, 12-39
Shift
function performing left, 9-52
function performing right, 9-52
Shift operations
functions performing, 9-16
SHIFTL function, 9-133
SHIFTR function, 9-133
Short field termination, 11-29
Short source lines
in fixed and tab source form, 2-11
Sign editing (S,SP,SS), 11-33
S, 11-33
SP, 11-33
SS, 11-33
SIGN function, 9-134
Significant digits
function returning number of, 9-48, 9-114
Simple list items
in I/O lists, 10-10
SIN function, 9-135
SIND function, 9-135
Sine
function returning hyperbolic, 9-136
function with argument in degrees, 9-135
function with argument in radians, 9-135
SINGLE directive, 14-68
example of, 14-69

Single-bit processing
functions performing, 9-16
SINH function, 9-136
SIZE function, 9-136
Size of array, 3-36
function returning, 9-136
SIZE specifier
for nonadvancing READs, 10-9
SIZEOF function, 9-137
Slash character (/)
as division operator, 4-2
precedence of, 4-11
as edit descriptor, 11-36
denoting common block, 5-21
preceding OPTIONS statement option, 13-4
See also Division operator
Slash editing, 11-36
SNGL function, 9-125
SNGLQ function, 9-125
Source code
allowable characters in, 2-5
debugging statements in, 2-12
fixed form of, 2-11
forms of, 2-6
free form of, 2-9
labels in, 2-8
See also Fixed source form
See also Free source form
See also Source program
See also Tab source form
See also your user's guide
tab form of, 2-11
useable in all forms, 2-15
Source forms, 2-6 thru 2-15
coding that works in all, 2-15
differences between fixed and tab, 2-13
extensions to rules for, F-1
fixed, 2-11
free, 2-9
indicators in, 2-7
See also Fixed source form
See also Free source form
See also Tab source form
tab, 2-11
Source listing
directive specifying subtitle for header in, 14-38
directive specifying title for header in, 14-38
of included files, 13-1
See also your user's guide
Source program
names in, 2-4
program unit in, 2-1
See also Source code
statement order in, 2-2
using D in, 2-12
SP edit descriptor, 11-33
Space
allocating for arrays and pointer targets, 6-2
deallocating for arrays and pointer targets, 6-5
disassociating for pointers, 6-8
See also Storage
Space characters
See Blank characters
SPACING function, 9-138
Special characters
in character set, 2-5
Specific names of intrinsics, 9-1
Specification expressions, 4-11, 4-13
examples of, 4-15
inquiry functions allowed in, 4-14
simplest form of, 4-13
Specification statements, 5-1 thru 5-58
disallowed in main programs, 8-3
disallowed in modules, 8-4
extensions to, F-2
SPREAD function, 9-138
SQRT function, 9-139
Square root
function returning, 9-139
SS edit descriptor, 11-33
Standards
See ANSI standard
See FIPS standard
See Fortran 90 standard
See Fortran 95 standard
See ISO standard
See MIL standard

Statement functions, 8-27
alternative for, A-3
definition of, 8-1, 8-27
examples of, 8-28
scope of, 15-2
See also your user's guide
Statement labels
See Labels
Statement numbers
See Labels
Statement order
in program units, 2-2
of OPTIONS statement, 13-4
Statement scope
names having, 15-2
Statement separator
in source form, 2-7
Statements
ACCEPT, 10-28
ALLOCATABLE, 5-17
ALLOCATE, 6-2
array declaration, 5-10
ASSIGN, 7-4
assignment, 4-15
defined, 4-21
intrinsic, 4-16
pointer, 4-22
AUTOMATIC, 5-18
BACKSPACE, 12-2
BLOCK DATA, 8-10
branch, 7-2
CALL, 7-7
CASE, 7-9
CASE DEFAULT, 7-9
character type declaration, 5-8
CLOSE, 12-3
COMMON, 5-21
CONTAINS, 8-53
CONTINUE, 7-14
continuing in fixed source form, 2-11
continuing in free source form, 2-10
continuing in tab source form, 2-11
control, 7-1
CYCLE, 7-24
DATA, 5-24
DEALLOCATE, 6-5
DECODE, B-3
DEFINE FILE, B-1
DELETE, 12-4
derived-type, 3-20
derived-type declaration, 5-10
DIMENSION, 5-27
DO, 7-15
DO WHILE, 7-23
ELSE, 7-26
ELSE IF, 7-26
ELSEWHERE, 4-23
ENCODE, B-3
END, 7-25
ENDFILE, 12-5
ENTRY, 8-53
EQUIVALENCE, 5-29
executable and nonexecutable, 2-2
EXIT, 7-24
EXTERNAL, 5-38
FIND, B-5
for compatibility between language versions, B-1
FORALL, 4-26
FORMAT, 11-2
formatting, 11-1
FUNCTION, 8-18
GO TO
assigned, 7-5
computed, 7-3
unconditional, 7-2
I/O
for data transfer, 10-1
for file operations, 12-1
IF
arithmetic, 7-6
block, 7-26
logical, 7-31
IMPLICIT, 5-39
INCLUDE, 13-1
INQUIRE, 12-7
INTENT, 5-41
INTERFACE
ASSIGNMENT, 8-51
generic, 8-49
OPERATOR, 8-50
INTERFACE TO, B-5

INTRINSIC, 5-43
labels for, 2-8
MAP, B-19
MODULE, 8-4
NAMELIST, 5-45
NULLIFY, 6-8
numeric and logical type declaration, 5-6
OPEN, 12-20
OPTIONAL, 5-46
OPTIONS, 13-3
overview of, 2-2
PARAMETER, 5-48
PAUSE, 7-32
POINTER
    Fortran 95/90, 5-50
    Integer, B-12
PRINT, 10-38
PRIVATE, 5-51
PROGRAM, 8-2
PUBLIC, 5-51
READ, 10-13
RECORD, B-21
required order of, 2-2
restricted from scoping units, 2-3
RETURN, 7-33
REWIND, 12-41
REWRITE, 10-39
SAVE, 5-54
SELECT CASE, 7-9
separating in source form, 2-7
SEQUENCE, 3-22
specification, 5-1
statement function, 8-27
STATIC, 5-18
STOP, 7-35
STRUCTURE, B-14
SUBROUTINE, 8-24
TARGET, 5-55
terminal
    See Terminal statements
type declaration, 5-2
TYPE definition, 3-20
TYPE I/O, 10-38
UNION, B-19
UNLOCK, 12-42
USE, 8-8
VIRTUAL, B-9
VOLATILE, 5-57
WHERE, 4-23
WRITE, 10-29
STATIC attribute and statement, 5-18
    attributes compatible with, 5-5
    examples of, 5-20
STATIC schedule type, 14-56
Static variables, 5-18
STATUS specifier
    in CLOSE statements, 12-40
    in OPEN statements, 12-39
STDCALL
    option for ATTRIBUTES directive, 14-10
STOP statement, 7-35
    effect on Linux systems, 7-35
    effect on Windows systems, 7-35
    examples of, 7-36
Storage
    and integer pointers, B-12
    association, 5-29, 15-13
        full, 15-15
        partial, 15-15
    function allocating, 9-91
    function freeing, 9-62
    function returning byte size of, 9-137
    function returning internal address of, 8-44, 9-89
    of arrays, 3-39
    requirements for intrinsic types, 15-14
    sequence, 15-13
    statement creating for allocatable arrays and pointer targets, 6-2
    statement defining contiguous, 5-21
    statement freeing for allocatable arrays and pointer targets, 6-5
    statement preserving for derived-type components, 3-20
    statement specifying shared, 5-29
    statements controlling allocation of variable, 5-18
    units, 15-13
Storage units
    types of, 15-13
STREAM value
    for INQUIRE (RECORDTYPE), 12-18

for OPEN (RECORDTYPE), 12-37
STREAM_CR value
for INQUIRE (RECORDTYPE), 12-18
for OPEN (RECORDTYPE), 12-37
STREAM_LF value
for INQUIRE (RECORDTYPE), 12-18
for OPEN (RECORDTYPE), 12-37
STRICT directive, 14-36
example of, 14-36
Stride
in array constructors, 3-45
in FORALL triplets, 4-26
in subscript triplets, 3-42, 3-43
String edit descriptors, 11-38 thru 11-40
repeating, 11-40
See also Character string edit descriptors
String-handling character functions, 9-4
Structure components, 3-23, 5-26
arrays as, 3-24
examples of, 3-24
in pointer assignment, 4-22
Structure constructors, 3-19, 3-26
examples of, 3-27
Structure declarations
derived type, 3-19, 3-20, 5-10
record, B-14
nesting, B-15
type declarations for, B-18
using %FILL in, B-18
STRUCTURE statement, B-14
using to initialize record fields, B-22
Structures
array, 3-35
derived-type, 3-19
array as component of, 3-21
components of, 3-19, 3-20
referencing components of, 3-23
record, B-14
See also Record structures
Subexpressions
in logical expressions, 4-10
SUBMIT value
for CLOSE statements, 12-3
for OPEN (DISPOSE), 12-31
SUBMIT/DELETE value
for CLOSE statements, 12-3
for OPEN (DISPOSE), 12-31
Subobjects, 3-33
Subprogram arguments
associating arrays with, 15-15
using aggregate field references as, B-23
Subprograms
automatic and static variables in, 5-18
effect of RETURN statement in, 7-33
ENTRY statements in, 8-55, 8-56
external, 2-1
internal, 2-2
module, 2-1
See also Functions
See also Internal subprograms
See also Module subprograms
See also Subroutines
terminating, 7-25
using as actual arguments, 5-38, 5-43
using assumed-length character arguments in, 3-34
using AUTOMATIC or STATIC in called, 5-18
Subroutine arguments, 8-30
See also Procedure arguments
Subroutine references, 7-7, 8-25
to elemental intrinsics, 8-42
to external names, 5-39
SUBROUTINE statement, 8-24
examples of, 8-25
prefixes in, 8-25
using with ENTRY statement, 8-25
Subroutine subprograms, 8-12
See also Subprograms
See also Subroutines
Subroutines, 8-24
containing ASSIGNMENT specifier, 4-21, 8-51
definition of, 8-2
elemental user-defined, 8-17
examples of, 8-25
general rules for, 8-13
intrinsic, 9-1
invoking, 8-25
prefixes in, 8-25
pure, 8-14
recursion in, 8-13

statement declaring as external, 5-38
statement declaring as intrinsic, 5-43
statement defining, 8-24
statement referencing, 7-7
statement transferring control to, 7-7
statements excluded from, 8-25

Subscript list, 3-35, 3-38
in array sections, 3-41
referencing array elements in, 3-38

Subscript progression
order of, 3-39

Subscript triplets, 3-42
stride in, 3-42

Subscripts
order of progression, 3-39
vector, 3-43

Substrings, 3-17
function returning starting position of, 9-73
making equivalent, 5-33
See also your user's guide

Substructure declarations, B-18

SUBTITLE directive, 14-38

Subtraction operator, 4-2
See also Unary operators

SUM function, 9-140

Sum of array elements
function returning, 9-140

SWP directive (i64), 14-37
example of, 14-37

Synchronization points
OpenMP directive identifying, 14-59

System errors
subroutine returning information on, 9-58

System subprograms
CPU_TIME, 9-42
DATE, 9-44
DATE_AND_TIME, 9-44
EXIT, 9-59
IDATE, 9-71
SECNDS, 9-130
SYSTEM_CLOCK, 9-141
TIME, 9-144

System time
function calculating in seconds, 9-130
subroutine returning, 9-144

SYSTEM_CLOCK subroutine, 9-141

T

T edit descriptor, 11-31

Tab source form, 2-11
blank characters in, 2-11
comment indicator in, 2-11
continuation character in, 2-11
debugging statement indicator in, 2-12
fields in, 2-13
labels in, 2-8
short source lines in, 2-11
statement field in, 2-13
statement separator in, 2-7

TAN function, 9-142

TAND function, 9-143

Tangent
function returning hyperbolic, 9-143
function with argument in degrees, 9-143
function with argument in radians, 9-142

TANH function, 9-143

TARGET attribute and statement, 5-55
attributes compatible with, 5-5
examples of, 5-56

Target statements
branch
See also Branch target statements

Targets
allocation of, 6-2, 6-4
as variables, 4-22
assigning values to, 4-16
associating with pointers, 4-16, 4-22, 5-55
deallocation of, 6-5, 6-7
declaration of, 5-55
disassociating from pointers, 6-1
dynamically allocating and deallocating, 6-1
requiring explicit interface, 8-46

Temporary suspension
of program execution, 7-32

Terminal statement
for block DO constructs, 7-16
for CASE constructs, 7-10
for IF constructs, 7-27
for nested DO constructs, 7-19
for nonblock DO constructs, 7-16
Termination
immediate
of DO constructs, 7-24
of innermost (or named) DO, 7-24
of program execution before end, 7-35
of short fields, 11-29
THREADPRIVATE directive, 14-69
example of, 14-70
Threads in a team
OpenMP directive synchronizing, 14-53
Time
function returning current in seconds, 9-130
function returning for program, 9-98
subroutines returning current, 9-44, 9-144
TIME subroutine, 9-144
TINY function, 9-144
TITLE directive, 14-38
TITLE specifier (W*32, W*64)
in OPEN statements, 12-40
TL edit descriptor, 11-32
TR edit descriptor, 11-32
TRAILZ function, 9-145
TRANSFER function, 9-145
Transfer of data, 10-2
See also Data transfer statements
Transformational functions
allowed in initialization expressions, 4-12
array
ALL, 9-23
ANY, 9-25
COUNT, 9-40
CSHIFT, 9-42
EOSHIFT, 9-54
MAXLOC, 9-95
MAXVAL, 9-97
MINLOC, 9-100
MINVAL, 9-102
PACK, 9-113
PRODUCT, 9-116
RESHAPE, 9-127
SPREAD, 9-138
SUM, 9-140
TRANSPOSE, 9-146
UNPACK, 9-148
character
REPEAT, 9-126
TRIM, 9-147
data transfer, 9-145
definition of, 9-1
numeric, 9-4
DOT_PRODUCT, 9-50
MATMUL, 9-92
SELECTED_INT_KIND, 9-130
SELECTED_REAL_KIND, 9-131
pointer
NULL, 9-112
TRANSPOSE function, 9-146
Transposed arrays
function producing, 9-146
TRIM function, 9-147
TRUE value
for INQUIRE (EXIST), 12-12
for INQUIRE (IOFOCUS), 12-14
for INQUIRE (NAMED), 12-15
for INQUIRE (OPENED), 12-16
for OPEN (IOFOCUS), 12-33
Truncation of assigned values, 4-16
Two’s complement
function returning length in, 9-72
TYPE
I/O statement, 10-38
keyword in derived type statements, 3-20, 3-21
specifier in OPEN statements, 12-40
Type declaration statements, 3-34, 5-2
array, 5-10
arrays in, 5-3
attributes in, 5-3
See also Attributes
character, 3-34, 5-8
constants in, 5-3
derived-type, 5-10
double colon separator in, 5-4

examples of, 5-5
examples of character, 5-9
examples of noncharacter, 5-7
initialization expressions in, 5-4
kind parameters in, 5-6
kind selector in, 5-3
limits within block data program unit, 8-11
numeric and logical, 5-6
specifiers in, 5-2
using to explicitly type variables, 3-34

Types
data, 3-1 thru 3-33
See also Data types

U

UBOUND function, 9-147
in pointer assignment, 5-16

Unambiguous generic references, 15-4

Unary operations, 4-3

Unary operators, 4-2
definition of, 4-3
form of defined, 8-50
precedence of, 4-11

Unconditional
DO statement, 7-15
GO TO statement, 7-2

Undeclared names, 5-39
See also Names

UNDEFINE directive, 14-17
example of, 14-17

Undefined variables, 3-33

Underscore character (_)
in names, 2-4

Unformatted data
specifying nonnative numeric, 12-11, 12-28
transfer
See also READ statements
See also WRITE statements
specifier to test for, 12-13, 12-19
specifying, 12-32

Unformatted I/O statements
READ
direct access, 10-24, 10-26
sequential, 10-14, 10-23
REWRITE, 10-39
using aggregate field references in, B-23
WRITE
direct access, 10-35, 10-36
sequential, 10-30, 10-34

Unformatted records
definition of, 10-2

UNFORMATTED specifier
in INQUIRE statements, 12-19

UNFORMATTED value
for INQUIRE (FORM), 12-13
for OPEN (FORM), 12-32

Union declarations, B-19
compared to EQUIVALENCE statement, B-20
initializing data in, B-19
size of shared area, B-20

UNION statement, B-19
using to make record fields equivalent, B-22

Unit number
assignment of, 10-4

UNIT specifier, 10-4

UNKNOWN value
for OPEN (STATUS), 12-39

UNLOCK statement, 12-42
examples of, 12-42

Unlocking records, 12-42

UNPACK function, 9-148

Unpacked array
function creating, 9-148

UNROLL directive, 14-39

Unspecified storage unit, 15-14

Upper bounds
function returning, 9-147

Uppercase letters
in character set, 2-5
treatment on compiler, 2-6

Use association, 8-7, 15-11

USE statement, 8-7, 8-8
examples of, 8-9
ONLY keyword in, 8-8

User-defined
data types, 3-19
See also Derived data types
elemental procedures, 8-17
pure procedures, 8-14
USEROPEN specifier
in OPEN statements, 12-40
User-written subprograms
for opening files, 12-40
types of, 8-12

V

VALUE
option for ATTRIBUTES directive, 14-15
Variable format expressions, 11-41
See also your user's guide
VARIABLE value
for INQUIRE (RECORDTYPE), 12-18
for OPEN (RECORDTYPE), 12-37
Variables, 3-33
allocating to stack or static storage, 5-19
assigning labels to, 7-4
assigning values to, 4-16
associating with group name, 5-45
automatic and static, 5-18
controlling storage allocation and initial value of, 5-18
defining and undefining, 3-33, 4-15
directive creating symbolic, 14-17
directive generating warnings for undeclared, 14-16
DO, 7-18
explicit typing of scalar, 3-34
implicit typing of scalar, 3-35
initializing, 5-24
in DATA statement, 5-24
length
effect on character assignments, 4-19
of name, 2-4
pointers as, 4-16
public, 8-7
referencing, 4-2
saving values of, 5-54
See also your user's guide
targets as, 4-22
truncation of values assigned to, 4-16
warnings for undeclared, 5-40, 14-16
VARYING
option for ATTRIBUTES directive, 14-16
VAXD value
for INQUIRE (CONVERT), 12-11
for OPEN (CONVERT), 12-28
VAXG value
for INQUIRE (CONVERT), 12-11
for OPEN (CONVERT), 12-28
VECTOR ALIGNED directive (i32), 14-39
VECTOR ALWAYS directive (i32), 14-40
example of, 14-40
VECTOR NONTEMPORAL directive (i32), 14-41
Vector subscripts, 3-43, 4-20
VECTOR UNALIGNED directive (i32), 14-39
Vectors
function performing dot-product multiplication of, 9-50
VERIFY function, 9-149
Virtual memory
using allocatable arrays, 6-4
VIRTUAL statement, B-9
See also DIMENSION attribute and statement
VOLATILE attribute and statement, 5-57
attributes compatible with, 5-5
examples of, 5-58
See also your user's guide

W

WARN=[NO]ALIGNMENT
OPTIONS directive option, 14-27
Warnings
directive generating for undeclared variables, 14-16
directive modifying for data alignment, 14-27
See also your user's guide
WHERE construct and statement, 4-23
as branch target, 4-24
ELSEWHERE, 4-23
examples of, 4-24
execution of, 4-25
WHILE statement, 7-14, 7-17, 7-23
examples of, 7-23

terminating, 7-23
Whole arrays, 3-36, 3-38
Window
specifying active, 12-33
specifying title for, 12-40
WRITE specifier
in INQUIRE statements, 12-19
WRITE statements, 10-29
compared to ENCODE statement, B-3
direct access, 10-35
examples of, 10-36
formatted, 10-35, 10-36
forms of, 10-35
unformatted, 10-35, 10-36
internal, 10-36
example of, 10-37
form of, 10-36
list-directed, 10-29, 10-31
example of, 10-32
namelist, 10-29, 10-33
example of, 10-33
nondelimited character strings in, 10-23
sequential, 10-29
formatted, 10-29, 10-30
forms of, 10-29
unformatted, 10-30, 10-34
WRITE value
for INQUIRE (ACTION), 12-9
for OPEN (ACTION), 12-25

X

X edit descriptor, 11-32
XOR function, 9-71

Z

Z edit descriptor, 11-13
input processing, 11-13
output processing, 11-14
ZABS function, 9-18
ZCOS function, 9-38
Zero character
effect in statement label fields, 2-8
ZERO value
for INQUIRE (BLANK), 12-10
for OPEN (BLANK), 12-26
Zero-extend function, 9-150
Zero-length format, 11-9
Zero-size array, 3-36, 5-12
section, 3-41
ZEXP function, 9-59
ZEXT function, 9-150
ZLOG function, 9-89
ZSIN function, 9-135
ZSQRT function, 9-139
ZTAN function, 9-142