Appendix A
Glossary

This appendix is based on the terms and definitions chapter in the standard. References are to the standard.

Actual argument entity (R1524) that appears in a procedure reference
Allocatable having the ALLOCATABLE attribute (8.5.3)
Array set of scalar data, all of the same type and type parameters, whose individual elements are arranged in a rectangular pattern (8.5.8, 9.5)

- **Array element** scalar individual element of an array
- **Array pointer** array with the POINTER attribute (8.5.14)
- **Array section** array subobject designated by array-section, and which is itself an array (9.5.3.3)
- **Assumed-shape array** nonallocatable nonpointer dummy argument array that takes its shape from its effective argument (8.5.8.3)
- **Assumed-size array** dummy argument array whose size is assumed from that of its effective argument (8.5.8.5)
- **Deferred-shape array** allocatable array or array pointer, declared with a deferred-shape-spec-list (8.5.8.4)
- **Explicit-shape array** array declared with an explicit-shape-spec-list, which specifies explicit values for the bounds in each dimension of the array (8.5.8.2)

ASCII character character whose representation method corresponds to ISO/IEC 646:1991 (International Reference Version)
Associate name name of construct entity associated with a selector of an ASSOCIATE, CHANGE TEAM, SELECT RANK, or SELECT TYPE construct (11.1.3, 11.1.5, 11.1.10, 11.1.11)
Associating entity 'in a dynamically-established association' the entity that did not exist prior to the establishment of the association (19.5.5)
Association inheritance association, name association, pointer association, or storage association.

- **Argument association** association between an effective argument and a dummy argument

Construct association association between a selector and an associate name in an ASSOCIATE, CHANGE TEAM, SELECT RANK, or SELECT TYPE construct (11.1.3, 11.1.5, 11.1.10, 11.1.11, 19.5.1.6)

Host association name association, other than argument association, between entities in a submodule or contained scoping unit and entities in its host (19.5.1.4)

Inheritance association association between the inherited components of an extended type and the components of its parent component (19.5.4)

Linkage association association between a variable or common block with the BIND attribute and a C global variable (18.9, 19.5.1.5)

Name association argument association, construct association, host association, linkage association, or use association (19.5.1)

Pointer association association between a pointer and an entity with the TARGET attribute (19.5.2)

Storage association association between storage sequences (19.5.3)

Use association association between entities in a module and entities in a scoping unit or construct that references that module, as specified by a USE statement (14.2.2)

Assumed-rank dummy data object dummy data object that assumes the rank, shape, and size of its effective argument (8.5.8.7)
Assumed-type declared with a TYPE(*) type specifier (7.3.2)
Attribute property of an entity that determines its uses (8.1)
Automatic data object nondummy data object with a type parameter or array bound that depends on the value of a specification-expr that is not a constant expression (8.3)
Base object 'data-ref' object designated by the leftmost part-name (9.4.2)
Binding type-bound procedure or final subroutine (7.5.5)
Binding name name given to a specific or generic type-bound procedure in the type definition (7.5.5)
Binding label default character value specifying the name by which a global entity with the BIND attribute is known to the companion processor (18.10.2, 18.9.2)
Block sequence of executable constructs formed by the syntactic class block and which is treated as a unit by the executable constructs described in 11.1
Bound array bound limit of a dimension of an array (8.5.8)
Branch target statement action-stmt, associate-stmt, end-associate-stmt, if-then-stmt, end-if-stmt, select-case-stmt, end-select-stmt, selectrank-stmt, end-select-rank-stmt, select-type-stmt, end-select-type-stmt, do-stmt, end-do-stmt, block-stmt, endblock-stmt, critical-stmt, end-critical-stmt, forall-construct-stmt, where-construct-stmt, end-function-stmt, end-mp-subprogram-stmt, end-program-stmt, or end-subroutine-stmt.
C address value identifying the location of a data object or procedure either defined by the companion processor or which might be accessible to the companion processor NOTE 3.1 This is the concept that ISO/IEC 9899:2011 calls the address.

C descriptor C structure of type CFI_cdesc_t defined in the source file ISO_Fortran_binding.h (18.4, 18.5)
Character context within a character literal constant (7.4.4) or within a character string edit descriptor (13.3.2)
Characteristics 'dummy argument' being a dummy data object, dummy procedure, or an asterisk (alternate return indicator)
Characteristics 'dummy data object' properties listed in 15.3.2.2
Characteristics 'dummy procedure or dummy procedure pointer' properties listed in 15.3.2.3
Characteristics 'function result' properties listed in 15.3.3
Characteristics 'procedure' properties listed in 15.3.1
Coarray data entity that has nonzero corank (5.4.7)

Established coarray coarray that is accessible using an image-selector (5.4.8)

Cobound bound (limit) of a codimension (8.5.6)
Codimension dimension of the pattern formed by a set of corresponding coarrays (8.5.6)
Coindexed object data object whose designator includes an image-selector (R924, 9.6)
Collating sequence one-to-one mapping from a character set into the nonnegative integers (7.4.4.4)
Common block block of physical storage specified by a COMMON statement (8.10.2)

Blank common unnamed common block

Companion processor processor-dependent mechanism by which global data and procedures may be referenced or defined (5.5.7)
Component part of a derived type, or of an object of derived type, defined by a component-def-stmt (7.5.4)

Direct component one of the components, or one of the direct components of a nonpointer nonallocatable component (7.5.1)
Parent component component of an extended type whose type is that of the parent type and whose components are inheritance associated with the inherited components of the parent type (7.5.7.2)
Potential subobject component nonpointer component, or potential subobject component of a nonpointer component (7.5.1)
Subcomponent 'structure' direct component that is a subobject of the structure (9.4.2)
Ultimate component component that is of intrinsic type, a pointer, or allocatable; or an ultimate component of a nonpointer nonallocatable component of derived type

Component order ordering of the nonparent components of a derived type that is used for intrinsic formatted input/output and structure constructors (where component keywords are not used) (7.5.4.7)

Conformable 'of two data entities' having the same shape, or one being an array and the other being scalar
Connected relationship between a unit and a file: each is connected if and only if the unit refers to the file (12.5.4)
Constant data object that has a value and which cannot be defined, redefined, or become undefined during execution of a program (6.2.3, 9.3)

Literal constant	constant that does not have a name (R605, 7.4)
Named constant	named data object with the PARAMETER attribute (8.5.13)

Construct entity entity whose identifier has the scope of a construct (19.1, 19.4)
Constant expression expression satisfying the requirements specified in 10.1.12, thus ensuring that its value is constant
Contiguous 'array' having array elements in order that are not separated by other data objects, as specified in 8.5.7
Contiguous 'multi-part data object' that the parts in order are not separated by other data objects
Corank number of codimensions of a coarray (zero for objects that are not coarrays) (8.5.6)
Cosubscript (R925) scalar integer expression in an image-selector (R924)
Data entity data object, result of the evaluation of an expression, or the result of the execution of a function reference
Data object object constant, variable, or subobject of a constant
Decimal symbol character that separates the whole and fractional parts in the decimal representation of a real number in a file (13.6)
Declaration specification of attributes for various program entities NOTE 3.2 Often this involves specifying the type of a named data object or specifying the shape of a named array object.
Default initialization mechanism for automatically initializing pointer components to have a defined pointer association status, and nonpointer components to have a particular value (7.5.4.6)
Default-initialized 'subcomponent' subject to a default initialization specified in the type definition for that component (7.5.4.6)
Definable capable of definition and permitted to become defined
Defined 'data object' has a valid value
Defined 'pointer' has a pointer association status of associated or disassociated (19.5.2.2)
Defined assignment assignment defined by a procedure (10.2.1.4, 15.4.3.4.3)
Defined input/output input/output defined by a procedure and accessed via a defined-io-generic-spec (R1509, 12.6.4.8)
Defined operation operation defined by a procedure (10.1.6.1, 15.4.3.4.2)
Definition 'data object' process by which the data object becomes defined (19.6.5)
Definition 'derived type (7.5.2), enumeration (7.6), or procedure (15.6)' specification of the type, enumeration, or procedure
Descendant 'module or submodule' submodule that extends that module or submodule or that extends another descendant thereof (14.2.3)

Designator name followed by zero or more component selectors, complex part selectors, array section selectors, array element selectors, image selectors, and substring selectors (9.1)

Complex part designator designator that designates the real or imaginary part of a complex data object, independently of the other part (9.4.4)
Object designator data object designator designator for a data object NOTE 3.3 An object name is a special case of an object designator.
Procedure designator designator for a procedure

Disassociated ‘pointer association’ pointer association status of not being associated with any target and not being undefined (19.5.2.2)
Disassociated ‘pointer’ has a pointer association status of disassociated
Dummy argument entity whose identifier appears in a dummy argument list in a FUNCTION, SUBROUTINE, ENTRY, or statement function statement, or whose name can be used as an argument keyword in a reference to an intrinsic procedure or a procedure in an intrinsic module

Dummy data object dummy argument that is a data object
Dummy function dummy procedure that is a function

Effective argument entity that is argument-associated with a dummy argument (15.5.2.3)
Effective item scalar object resulting from the application of the rules in 12.6.3 to an input/output list
Elemental independent scalar application of an action or operation to elements of an array or corresponding elements of a set of conformable arrays and scalars, or possessing the capability of elemental operation NOTE 3.4 Combination of scalar and array operands or arguments combine the scalar operand(s) with each element of the array operand(s).

Elemental assignment assignment that operates elementally
Elemental operation operation that operates elementally
Elemental operator operator in an elemental operation
Elemental procedure elemental intrinsic procedure or procedure defined by an elemental subprogram (15.8)
Elemental reference reference to an elemental procedure with at least one array actual argument
Elemental subprogram subprogram with the ELEMENTAL prefix (15.8.1)

END statement end-block-data-stmt, end-function-stmt, end-module-stmt, end-mp-subprogram-stmt, end-program-stmt, end-submodule-stmt, or end-subroutine-stmt
Explicit initialization initialization of a data object by a specification statement (8.4, 8.6.7)
Extent number of elements in a single dimension of an array
External file file that exists in a medium external to the program (12.3)
External unit external input/output unit entity that can be connected to an external file (12.5.3, 12.5.4)

File storage unit unit of storage in a stream file or an unformatted record file (12.3.5)
Final subroutine subroutine whose name appears in a FINAL statement (7.5.6) in a type definition, and which can be automatically invoked by the processor when an object of that type is finalized (7.5.6.2)
Finalizable ‘type’ has a final subroutine or a nonpointer nonallocatable component of finalizable type
Finalizable ‘nonpointer data entity’ of finalizable type
Finalization process of calling final subroutines when one of the events listed in 7.5.6.3 occurs
Function procedure that is invoked by an expression
Function result entity that returns the value of a function (15.6.2.2)
Generic identifier lexical token that identifies a generic set of procedures, intrinsic operations, and/or intrinsic assignments (15.4.3.4.1)
Host instance ‘internal procedure, or dummy procedure or procedure pointer associated with an internal procedure’ instance of the host procedure that supplies the host environment of the internal procedure (15.6.2.4)
Host scoping unit host scoping unit immediately surrounding another scoping unit, or the scoping unit extended by a submodule
IEEE infinity ISO/IEC/IEEE 60559:2011 conformant infinite floating-point value
IEEE NaN ISO/IEC/IEEE 60559:2011 conformant floating-point datum that does not represent a number
Image instance of a Fortran program (5.3.4)

- **Active image** image that has not failed or stopped (5.3.6)
- **Failed image** image that has not initiated termination but which has ceased to participate in program execution (5.3.6)
- **Stopped image** image that has initiated normal termination (5.3.6)

Image index integer value identifying an image within a team
Image control statement statement that affects the execution ordering between images (11.6)
Inclusive scope nonblock scoping unit plus every block scoping unit whose host is that scoping unit or that is nested within such a block scoping unit NOTE 3.5 That is, inclusive scope is the scope as if BLOCK constructs were not scoping units.
Inherit ‘extended type’ acquire entities (components, type-bound procedures, and type parameters) through type extension from the parent type (7.5.7.2)
Inquiry function intrinsic function, or function in an intrinsic module, whose result depends on the properties of one or more of its arguments instead of their values
Interface ‘procedure’ name, procedure characteristics, dummy argument names, binding label, and generic identifiers (15.4.1)

- **Abstract interface** set of procedure characteristics with dummy argument names (15.4.1)
- **Explicit interface** interface of a procedure that includes all the characteristics of the procedure and names for its dummy arguments except for asterisk dummy arguments (15.4.2)

Generic interface set of procedure interfaces identified by a generic identifier
Implicit interface interface of a procedure that is not an explicit interface (15.4.2, 15.4.3.8)
Specific interface interface identified by a nongeneric name

Interface block abstract interface block, generic interface block, or specific interface block (15.4.3.2)

Abstract interface block interface block with the ABSTRACT keyword; collection of interface bodies that specify named abstract interfaces
Generic interface block interface block with a generic-spec; collection of interface bodies and procedure statements that are to be given that generic identifier
Specific interface block interface block with no generic-spec or ABSTRACT keyword; collection of interface bodies that specify the interfaces of procedures

Interoperable 'Fortran entity' equivalent to an entity defined by or definable by the companion processor (18.3)
Intrinsic type, procedure, module, assignment, operator, or input/output operation defined in this document and accessible without further definition or specification, or a procedure or module provided by a processor but not defined in this document

Standard intrinsic 'procedure or module' defined in this document (16)
Nonstandard intrinsic 'procedure or module' provided by a processor but not defined in this document

Internal file character variable that is connected to an internal unit (12.4)
Internal unit input/output unit that is connected to an internal file (12.5.4)
ISO 10646 character character whose representation method corresponds to UCS-4 in ISO/IEC 10646
Keyword statement keyword, argument keyword, type parameter keyword, or component keyword

Argument keyword word that identifies the corresponding dummy argument in an actual argument list (15.5.2.1)
Component keyword word that identifies a component in a structure constructor (7.5.10)
Statement keyword word that is part of the syntax of a statement (5.5.2)
Type parameter keyword word that identifies a type parameter in a type parameter list

Lexical token keyword, name, literal constant other than a complex literal constant, operator, label, delimiter, comma, =, =>, :, ::, ;, or % (6.2)
Line sequence of zero or more characters
Main program program unit that is not a subprogram, module, submodule, or block data program unit (14.1)
Masked array assignment assignment statement in a WHERE statement or WHERE construct

Module program unit containing (or accessing from other modules) definitions that are to be made accessible to other program units (14.2)
Name identifier of a program constituent, formed according to the rules given in 6.2.2
NaN Not a Number, a symbolic floating-point datum (ISO/IEC/IEEE 60559:2011)
Operand data value that is the subject of an operator
Operator intrinsic-operator, defined-unary-op, or defined-binary-op (R608, R1003, R1023)
Passed-object dummy argument dummy argument of a type-bound procedure or procedure pointer component that becomes associated with the object through which the procedure is invoked (7.5.4.5)
Pointer data pointer or procedure pointer

- **Data pointer** data entity with the POINTER attribute (8.5.14)
- **Procedure pointer** procedure with the EXTERNAL and POINTER attributes (8.5.9, 8.5.14)
- **Local procedure pointer** procedure pointer that is part of a local variable, or a named procedure pointer that is not a dummy argument or accessed by use or host association

Pointer assignment association of a pointer with a target, by execution of a pointer assignment statement (10.2.2) or an intrinsic assignment statement (10.2.1.2) for a derived-type object that has the pointer as a subobject
Polymorphic 'data entity' able to be of differing dynamic types during program execution (7.3.2.3)
Preconnected 'file or unit' connected at the beginning of execution of the program (12.5.5)
Procedure entity encapsulating an arbitrary sequence of actions that can be invoked directly during program execution

- **Dummy procedure** procedure that is a dummy argument (15.2.2.3)
- **External procedure** procedure defined by an external subprogram (R503) or by means other than Fortran (15.6.3)
- **Internal procedure** procedure defined by an internal subprogram (R512)
- **Module procedure** procedure defined by a module subprogram, or a procedure provided by an intrinsic module (R1408)
- **Pure procedure** procedure declared or defined to be pure (15.7)
- **Type-bound procedure** procedure that is bound to a derived type and referenced via an object of that type (7.5.5)

Processor combination of a computing system and mechanism by which programs are transformed for use on that computing system
Processor dependent not completely specified in this document, having methods and semantics determined by the processor
Program set of Fortran program units and entities defined by means other than Fortran that includes exactly one main program

Program unit main program, external subprogram, module, submodule, or block data program unit (5.2.1)
Rank number of array dimensions of a data entity (zero for a scalar entity)
Record sequence of values or characters in a file (12.2)
Record file file composed of a sequence of records (12.1)
Reference data object reference, procedure reference, or module reference

- **Data object reference** appearance of a data object designator (9.1) in a context requiring its value at that point during execution
- **Function reference** appearance of the procedure designator for a function, or operator symbol for a defined operation, in a context requiring execution of the function during expression evaluation (15.5.3)
- **Module reference** appearance of a module name in a USE statement (14.2.2)
- **Procedure reference** appearance of a procedure designator, operator symbol, or assignment symbol in a context requiring execution of the procedure at that point during execution; or occurrence of defined input/output (13.7.6) or derived-type finalization (7.5.6.2)

Saved having the SAVE attribute (8.5.16)
Scalar data entity that can be represented by a single value of the type and that is not an array (9.5)
Scoping unit BLOCK construct, derived-type definition, interface body, program unit, or subprogram, excluding all nested scoping units in it

- **Block scoping unit** scoping unit of a BLOCK construct

Sequence set of elements ordered by a one-to-one correspondence with the numbers 1, 2, to n
Sequence structure scalar data object of a sequence type (7.5.2.3)
Sequence type derived type with the SEQUENCE attribute (7.5.2.3)

- **Character sequence type** sequence type with no allocatable or pointer components, and whose components are all default character or of another character sequence type
- **Numeric sequence type** sequence type with no allocatable or pointer components, and whose components are all default complex, default integer, default logical, default real, double precision real, or of another numeric sequence type

Shape array dimensionality of a data entity, represented as a rank-one array whose size is the rank of the data entity and whose elements are the extents of the data entity NOTE 3.6 Thus the shape of a scalar data entity is an array with rank one and size zero.
Simply contiguous ‘array designator or variable’ satisfying the conditions specified in 9.5.4 NOTE 3.7 These conditions are simple ones which make it clear that the designator or variable designates a contiguous array.
Size ‘array’ total number of elements in the array

Specification expression expression satisfying the requirements specified in 10.1.11, thus being suitable for use in specifications
Specific name name that is not a generic name
Standard-conforming program program that uses only those forms and relationships described in, and has an interpretation according to, this document
Statement sequence of one or more complete or partial lines satisfying a syntax rule that ends in -stmt (6.3)

Executable statement end-function-stmt, end-mp-subprogram-stmt, end-program-stmt, end-subroutine-stmt, or statement that is a member of the syntactic class executable-construct, excluding those in the block-specification-part of a BLOCK construct
Nonexecutable statement statement that is not an executable statement

“aStatement entity entity whose identifier has the scope of a statement or part of a statement (19.1, 19.4)
Statement label label unsigned positive number of up to five digits that refers to an individual statement (6.2.5)
Storage sequence contiguous sequence of storage units (19.5.3.2)
Storage unit character storage unit, numeric storage unit, file storage unit, or unspecified storage unit (19.5.3.2)

Character storage unit unit of storage that holds a default character value (19.5.3.2)
Numeric storage unit unit of storage that holds a default real, default integer, or default logical value (19.5.3.2)
Unspecified storage unit unit of storage that holds a value that is not default character, default real, double precision real, default logical, or default complex (19.5.3.2)

Stream file file composed of a sequence of file storage units (12.1)
Structure scalar data object of derived type (7.5)

Structure component component of a structure
Structure constructor syntax (structure-constructor, 7.5.10) that specifies a structure value or creates such a value

Submodule program unit that extends a module or another submodule (14.2.3)
Subobject portion of data object that can be referenced, and if it is a variable defined, independently of any other portion
Subprogram function-subprogram (R1529) or subroutine-subprogram (R1534)

External subprogram subprogram that is not contained in a main program, module, submodule, or another subprogram
Internal subprogram subprogram that is contained in a main program or another subprogram
Module subprogram subprogram that is contained in a module or submodule but is not an internal subprogram

Subroutine procedure invoked by a CALL statement, by defined assignment, or by some operations on derived-type entities

Atomic subroutine intrinsic subroutine that performs an action on its ATOM argument atomically
Collective subroutine intrinsic subroutine that performs a calculation on a team of images without requiring synchronization

Target entity that is pointer associated with a pointer (19.5.2.2), entity on the right-hand-side of a pointer assignment statement (R1033), or entity with the TARGET attribute (8.5.17)
Team ordered set of images created by execution of a FORM TEAM statement, or the initial ordered set of all images

Current team team specified by the most recently executed CHANGE TEAM statement of a CHANGE TEAM construct that has not completed execution (11.1.5), or initial team if no CHANGE TEAM construct is being executed
Initial team team existing at the beginning of program execution, consisting of all images
Parent team 'team except for initial team' current team at time of execution of the FORM TEAM statement that created the team (11.6.9)
Team number -1 which identifies the initial team, or positive integer that identifies a team within its parent team

Transformational function intrinsic function, or function in an intrinsic module, that is neither elemental nor an inquiry function
Type data type named category of data characterized by a set of values, a syntax for denoting these values, and a set of operations that interpret and manipulate the values (7.1)

Abstract type type with the ABSTRACT attribute (7.5.7.1)
Declared type type that a data entity is declared to have, either explicitly or implicitly (7.3.2, 10.1.9)
Derived type type defined by a type definition (7.5) or by an intrinsic module
Dynamic type type of a data entity at a particular point during execution of a program (7.3.2.3, 10.1.9)
Extended type type with the EXTENDS attribute (7.5.7.1)
Extensible type type that may be extended using the EXTENDS clause (7.5.7.1)
Extension type 'of one type with respect to another' is the same type or is an extended type whose parent type is an extension type of the other type
Intrinsic type type defined by this document that is always accessible (7.4)
Numeric type one of the types integer, real, and complex
Parent type 'extended type' type named in the EXTENDS clause
Type compatible compatibility of the type of one entity with respect to another for purposes such as argument association, pointer association, and allocation (7.3.2)

- **Type parameter** value used to parameterize a type (7.2)
- **Assumed type parameter** length type parameter that assumes the type parameter value from another entity NOTE 3.8 The other entity is the selector for an associate name, the constant-expr for a named constant of type character, or NOTE 3.8 (cont.) the effective argument for a dummy argument.
- **Deferred type parameter** length type parameter whose value can change during execution of a program and whose type-param-value is a colon
- **Kind type parameter** type parameter whose value is required to be defaulted or given by a constant expression
- **Length type parameter** type parameter whose value is permitted to be assumed, deferred, or given by a specification expression
- **Type parameter inquiry** syntax (type-param-inquiry) that is used to inquire the value of a type parameter of a data object (9.4.5)
- **Type parameter order** ordering of the type parameters of a type (7.5.3.2) used for derived-type specifiers (derived-type-spec, 7.5.9)

Ultimate argument nondummy entity with which a dummy argument is associated via a chain of argument associations (15.5.2.3)
Undefined 'data object' does not have a valid value
Undefined 'pointer' does not have a pointer association status of associated or disassociated (19.5.2.2)
Unit input/output unit means, specified by an io-unit, for referring to a file (12.5.1)
Unlimited polymorphic able to have any dynamic type during program execution (7.3.2.3)
Unsaved not having the SAVE attribute (8.5.16)
Variable data entity that can be defined and redefined during execution of a program

- **Event variable** scalar variable of type EVENT_TYPE (16.10.2.10) from the intrinsic module ISO_FORTRAN_ENV
- **Local variable** variable in a scoping unit that is not a dummy argument or part thereof, is not a global entity or part thereof, and is not an entity or part of an entity that is accessible outside that scoping unit
- **Lock variable** scalar variable of type LOCK_TYPE (16.10.2.19) from the intrinsic module ISO_FORTRAN_ENV
- **Team variable** scalar variable of type TEAM_TYPE (16.10.2.32) from the intrinsic module ISO_FORTRAN_ENV

Vector subscript section-subscript that is an array (9.5.3.3.2)
Whole array array component or array name without further qualification (9.5.2)

Appendix B
Attribute Declarations and Specifications

This appendix is based on Chap. 8 in the standard. References are to the standard.

Attributes of Procedures and Data Objects

Every data object has a type and rank and may have type parameters and other properties that determine the uses of the object. Collectively, these properties are the attributes of the object. The declared type of a named data object is either specified explicitly in a type declaration statement or determined implicitly by the first letter of its name (8.7). All of its attributes may be specified in a type declaration statement or individually in separate specification statements.

A function has a type and rank and may have type parameters and other attributes that determine the uses of the function. The type, rank, and type parameters are the same as those of the function result.

A subroutine does not have a type, rank, or type parameters, but may have other attributes that determine the uses of the subroutine.

Type Declaration Statement

A type declaration statement specifies the declared type of the entities in the entity declaration list.

Attribute Specification

An attribute specifier can be one or more of

- ALLOCATABLE
- ASYNCHRONOUS
- BIND C
- CODIMENSION
- CONTIGUOUS
- DIMENSION
- EXTERNAL
- INTENT
- INTRINSIC

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran*,
https://doi.org/10.1007/978-3-319-75502-1

- OPTIONAL
- PARAMETER
- POINTER
- PRIVATE
- PROTECTED
- PUBLIC
- SAVE
- TARGET
- VALUE
- VOLATILE

Attribute Specification Statements
These include

- ALLOCATABLE
- ASYNCHRONOUS
- BIND C
- CODIMENSION
- CONTIGUOUS
- DATA
- DIMENSION
- INTENT
- OPTIONAL
- PARAMETER
- POINTER
- PROTECTED
- SAVE
- TARGET
- VALUE
- VOLATILE

Appendix C
Compatibility

Previous Fortran Standards

Table C.1 lists the previous editions of the Fortran International Standard, along with their informal names.

Table C.1 Previous editions of the Fortran standard

Official name	Informal name
ISO R 1539-1972	Fortran 66
ISO 1539-1980	Fortran 77
ISO/IEC 1539:1991	Fortran 90
ISO/IEC 1539-1:1997	Fortran 95
ISO/IEC 1539-1:2004	Fortran 2003
ISO/IEC 1539-1:2010	Fortran 2008

New Intrinsic Procedures

Each Fortran International Standard since ISO 1539:1980 (Fortran 77), defines more intrinsic procedures than the previous one. Therefore, a Fortran program conforming to an older standard might have a different interpretation under a newer standard if it invokes an external procedure having the same name as one of the new standard intrinsic procedures, unless that procedure is specified to have the EXTERNAL attribute.

Fortran 2008 Compatibility

Except as identified in this subclause, and except for the deleted features noted in Annex B.2, the Fortran 2018 standard is an upward compatible extension to the preceding Fortran International Standard, ISO/IEC 1539-1:2010 (Fortran). Any standard-conforming Fortran 2008 program that does not use any deleted features,

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran*,
https://doi.org/10.1007/978-3-319-75502-1

and does not use any feature identified in this subclause as being no longer permitted, remains standard-conforming in the Fortran 2018 standard.

Fortran 2008 specifies that the IOSTAT = variable shall be set to a processor-dependent negative value if the flush operation is not supported for the unit specified. the Fortran 2018 standard specifies that the processor-dependent negative integer value shall be different from the named constants IOSTAT_EOR or IOSTAT_END from the intrinsic module ISO_FORTRAN_ENV.

Fortran 2008 permitted a noncontiguous array that was supplied as an actual argument corresponding to a contiguous INTENT (INOUT) dummy argument in one iteration of a DO CONCURRENT construct, without being previously defined in that iteration, to be defined in another iteration;

Fortran 2008 permitted a pure statement function to reference a volatile variable, and permitted a local variable of a pure subprogram or of a BLOCK construct within a pure subprogram to be volatile (provided it was not used); the Fortran 2018 standard does not permit this.

Fortran 2008 permitted a pure function to have a result that has a polymorphic allocatable ultimate component; the Fortran 2018 standard does not permit this.

Fortran 2008 permitted a PROTECTED TARGET variable accessed by use association to be used as an initial7 data-target; the Fortran 2018 standard does not permit this.

Fortran 2008 permitted a named constant to have declared type LOCK_TYPE, or have a noncoarray potential subobject component with declared type LOCK_TYPE; the Fortran 2018 standard does not permit this.

Fortran 2008 permitted a polymorphic object to be finalized within a DO CONCURRENT construct; the Fortran 2018 standard does not permit this.

Fortran 2003 Compatibility

Except as identified in this subclause, the Fortran 2018 standard is an upward compatible extension to ISO/IEC 1539-1:2004 (Fortran 2003). Except as identified in this subclause, any standard-conforming Fortran 2003 program remains standard-conforming in the Fortran 2018 standard.

Fortran 2003 permitted a sequence type to have type parameters; that is not permitted by the Fortran 2018 standard.

Fortran 2003 specified that array constructors and structure constructors of finalizable type are finalized. The Fortran 2018 standard specifies that these constructors are not finalized.

The form produced by the G edit descriptor for some values and some input/output rounding modes differs from that specified by Fortran 2003.

Fortran 2003 required an explicit interface only for a procedure that was actually referenced in the scope, not merely passed as an actual argument. the Fortran 2018 standard requires an explicit interface for a procedure under the conditions listed in 15.4.2.2, regardless of whether the procedure is referenced in the scope.

Fortran 2003 permitted the function result of a pure function to be a polymorphic allocatable variable, to have a polymorphic allocatable ultimate component, or to be finalizable by an impure final subroutine. These are not permitted by the Fortran 2018 standard.

Fortran 2003 permitted an INTENT (OUT) argument of a pure subroutine to be polymorphic; that is not permitted by the Fortran 2018 standard.

Fortran 2003 interpreted assignment to an allocatable variable from a nonconformable array as intrinsic assignment, even when an elemental defined assignment was in scope; the Fortran 2018 standard does not permit assignment from a nonconformable array in this context.

Fortran 2003 permitted a statement function to be of parameterized derived type; the Fortran 2018 standard does not permit this.

Fortran 2003 permitted a pure statement function to reference a volatile variable, and permitted a local variable of a pure subprogram to be volatile (provided it was not used); the Fortran 2018 standard does not permit this

Fortran 95 Compatibility

Except as identified in this subclause, the Fortran 2018 standard is an upward compatible extension to ISO/IEC 1539-1:1997 (Fortran 95). Except as identified in this subclause, any standard-conforming Fortran 95 program remains standard-conforming in the Fortran 2018 standard.

Fortran 95 permitted defined assignment between character strings of the same rank and different kinds. This document does not permit that if both of the different kinds are ASCII, ISO 10646, or default kind.

The following Fortran 95 features might have different interpretations in the Fortran 2018 standard.

Earlier Fortran standards had the concept of printing, meaning that column one of formatted output had special meaning for a processor-dependent (possibly empty) set of external files. This could be neither detected nor specified by a standard-specified means. The interpretation of the first column is not specified by the Fortran 2018 standard.

The Fortran 2018 standard specifies a different output format for real zero values in list-directed and namelist output.

If the processor distinguishes between positive and negative real zero, the Fortran 2018 standard requires different returned values for ATAN2(Y,X) when $X < 0$ and Y is negative real zero and for LOG(X) and SQRT(X) when X is complex with $X\%RE < 0$ and $X\%IM$ is negative real zero.

The Fortran 2018 standard has fewer restrictions on constant expressions than Fortran 95; this might affect whether a variable is considered to be an automatic data object.

The form produced by the G edit descriptor with d equal to zero differs from that specified by Fortran 95 for some values.

Fortran 90 Compatibility

Except for the deleted features noted in Annex B.1, and except as identified in this subclause, the Fortran 2018 standard is an upward compatible extension to ISO/IEC 1539:1991 (Fortran 90). Any standard-conforming Fortran 90 program that does not use one of the deleted features remains standard-conforming in the Fortran 2018 standard.

The PAD = specifier in the INQUIRE statement in the Fortran 2018 standard returns the value UNDEFINED if there is no connection or the connection is for unformatted input/output. Fortran 90 specified YES.

Fortran 90 specified that if the second argument to MOD or MODULO was zero, the result was processor dependent. The Fortran 2018 standard specifies that the second argument shall not be zero.

Fortran 90 permitted defined assignment between character strings of the same rank and different kinds. This document does not permit that if both of the different kinds are ASCII, ISO 10646, or default kind.

The following Fortran 90 features have different interpretations in the Fortran 2018 standard:

if the processor distinguishes between positive and negative real zero, the result value of the intrinsic function SIGN when the second argument is a negative real zero;

formatted output of negative real values (when the output value is zero);

whether an expression is a constant expression (thus whether a variable is considered to be an automatic data object);

the G edit descriptor with d equal to zero for some values.

FORTRAN 77 Compatibility

Except for the deleted features noted in Annex B.1, and except as identified in this subclause, the Fortran 2018 standard is an upward compatible extension to ISO 1539:1980 (Fortran 77). Any standard-conforming Fortran 77 program that does not use one of the deleted features noted in Annex B.1 and that does not depend on the differences specified here remains standard-conforming in the Fortran 2018 standard. the Fortran 2018 standard restricts the behaviour for some features that were processor dependent in Fortran 77. Therefore, a standard-conforming Fortran 77 program that uses one of these processor-dependent features might have a different interpretation in the Fortran 2018 standard, yet remain a standard-conforming program. The following Fortran 77 features might have different interpretations in the Fortran 2018 standard.

Fortran 77 permitted a processor to supply more precision derived from a default real constant than can be represented in a default real datum when the constant is used to initialize a double precision real data object in a DATA statement. the Fortran 2018 standard does not permit a processor this option.

If a named variable that was not in a common block was initialized in a DATA statement and did not have the SAVE attribute specified, Fortran 77 left its SAVE attribute processor dependent. the Fortran 2018 standard specifies (8.6.7) that this named variable has the SAVE attribute.

Fortran 77 specified that the number 1 of characters required by the input list was to be less than or equal to the number of characters in the record during formatted input. the Fortran 2018 standard specifies (12.6.4.5.3) that the input record is logically padded with blanks if there are not enough characters in the record, unless the PAD= specifier with the value 'NO' is specified in an appropriate OPEN or READ statement.

A value of 0 for a list item in a formatted output statement will be formatted in a different form for some G edit descriptors. In addition, the Fortran 2018 standard specifies how rounding of values will affect the output field form, but Fortran 77 did not address this issue. Therefore, the form produced for certain combinations of values and G edit descriptors might differ from that produced by some Fortran 77 processors.

Fortran 77 did not permit a processor to distinguish between positive and negative real zero; if the processor does so distinguish, the result will differ for the intrinsic function SIGN when the second argument is negative real zero, and formatted output of negative real zero will be different.

Appendix D
Intrinsic Functions and Procedures

This appendix has a brief coverage of some of the more commonly used intrinsic functions and procedures. Chapter 16 of the standard should be consulted for an exhaustive coverage.

The following abbreviations and typographic conventions are used in this appendix.

D.1 Argument Type and Return Type

These are documented in Table D.1.

Table D.1 Argument and return type abbreviations

Abbreviation	Meaning
a	Any
i	Integer
r	Real
c	Complex
n	Numeric (any of integer, real, complex)
l	Logical
p	Pointer
p*	Polymorphic
t	Target
dp	Double precision
char	Character, length = 1
s	Character
boz	Boz-literal-constant
co	Coarray or coindexed object
te	Team type

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran*,
https://doi.org/10.1007/978-3-319-75502-1

D.2 Classes of Function

There are several classes of function in Fortran and they are documented below (Table D.2).

Table D.2 Classes of function

Class	Description
a	Indicates that the procedure is an atomic subroutine
e	Indicates that the procedure is an elemental function
es	Indicates that the procedure is an elemental subroutine
i	Indicates that the procedure is an inquiry function
ps	Indicates that the procedure is a pure subroutine
s	Indicates that the procedure is an impure subroutine
t	Indicates that the procedure in a transformational function

D.3 Optional Arguments

Arguments in italics or [] brackets are optional arguments.

In the example `ALL(mask,`*`dim`*`)` `dim` may be omitted.

D.4 Common Optional Arguments

These are documented in Table D.3.

Table D.3 Common optional arguments

Argument	Description
Back	Controls the direction of string scan, forward or backward
Dim	A selected dimension of an array argument
Kind	Describes the kind type parameter of the result If the kind argument is absent the result is the same type as the first argument.
Mask size	A mask may be applied to the arguments f an array, the total number of elements

D.5 Double Precision

Before Fortran 90 if you required real variables to have greater precision than the default real then the only option available was to declare them as double precision. With the introduction of kind types with Fortran 90 the use of double precision declarations is not recommended, and instead real entities with a kind type offering more than the default precision should be used.

D.6 Result Type

When the result type is the same as the argument type then the result is not just the same type as the argument but also the same kind.

D.7 Miscellaneous Rules

All intrinsic procedures may be invoked with either positional arguments or argument keywords.

Many of the intrinsic functions have optional arguments.

Unless otherwise specified the intrinisc inquiry functions accept array arguments for which the shape need not be defined. The shape of array arguments to transformational and elemental intrinsic functions shall be defined.

Some array intrinsic functions are reduction functions - they reduce the rank of an array by collapsing one dimension (or all dimensions, usually producing a scalar result).

When the argument is `back` it is of logical type.

When the argument is `count_rate`, `count_max`, `dim`, `kind`, `len`, `order`, `n_copies`, `shape`, `shift`, `values` it is of integer type.

When the argument is `mask` it is of logical type.

When the argument is `target` it is of `pointer` or `target` type.

Fortran 2008 introduced several changes to Fortran 2003 that affected intrinsic procedures.

- The following functions can now have arguments of type complex: `acos`, `asin`, `atan`, `cosh`, `sinh`, `tan` and `tanh`.
- The intrinsic function `atan2` can be referenced by the name `atan`.
- The intrinsic functions `lge`, `lgt`, `lle` and `llt` can have arguments of ASCII kind.
- The intrinsic functions `maxloc` and `minloc` have an additional `back` argument.
- The intrinsic function `selected_real_kind` has an additional `radix` argument.

Fortran 2018 introduced the following intrinsic functions and procedures

- ATOMIC_ADD (ATOM, VALUE [, STAT])
- ATOMIC_AND (ATOM, VALUE [, STAT])
- ATOMIC_CAS (ATOM, OLD, COMPARE, NEW [, STAT])
- ATOMIC_DEFINE (ATOM, VALUE [, STAT])
- ATOMIC_FETCH_ADD (ATOM, VALUE, OLD [, STAT])
- ATOMIC_FETCH_AND (ATOM, VALUE, OLD [, STAT])
- ATOMIC_FETCH_OR (ATOM, VALUE, OLD [, STAT])
- ATOMIC_FETCH_XOR (ATOM, VALUE, OLD [, STAT])
- ATOMIC_OR (ATOM, VALUE [, STAT])
- ATOMIC_REF (VALUE, ATOM [, STAT])
- ATOMIC_XOR (ATOM, VALUE [, STAT])
- CO_BROADCAST(A, SOURCE_IMAGE [, STAT, ERRMSG])
- CO_MAX(A [, RESULT_IMAGE, STAT, ERRMSG])
- CO_MIN(A [, RESULT_IMAGE, STAT, ERRMSG])
- CO_REDUCE(A, OPERATION [, RESULT_IMAGE, STAT, ERRMSG])
- CO_SUM(A [, RESULT_IMAGE, STAT, ERRMSG])
- COSHAPE (COARRAY [, KIND])
- FAILED_IMAGES([TEAM, KIND])
- FINDLOC (ARRAY, VALUE, DIM [, MASK, KIND, BACK])
- FINDLOC (ARRAY, VALUE [, MASK, KIND, BACK])
- GET_TEAM([LEVEL])
- IMAGE_STATUS (IMAGE [, TEAM])
- LCOBOUND (COARRAY [, DIM, KIND])
- OUT_OF_RANGE (X, MOLD [, ROUND])
- RANDOM_INIT (REPEATABLE, IMAGE_DISTINCT)
- RANK (A)
- REDUCE (ARRAY, OPERATION [, MASK, IDENTITY, ORDERED])
- REDUCE (ARRAY, OPERATION, DIM [, MASK, IDENTITY,ORDERED])
- STOPPED_IMAGES([TEAM, KIND])
- TEAM_NUMBER([TEAM])
- THIS_IMAGE ([TEAM]) or THIS_IMAGE (COARRAY [, TEAM])
- THIS_IMAGE (COARRAY, DIM [, TEAM])
- UCOBOUND (COARRAY [, DIM, KIND])

D.8 Intrinsic functions list

These are documented in Table D.4, where some of the procedure names are split over multiple lines.

Table D.4 Standard generic intrinsic procedure summary

Procedure	Class	Description
ABS	E	Absolute value
ACHAR	E	Character from ASCII code value
ACOS	E	function
ACOSH	E	Inverse hyperbolic cosine function
ADJUSTL	E	Left-justified string value
ADJUSTR	E	Right-justified string value
AIMAG	E	Imaginary part of a complex number
AINT	E	Truncation toward 0 to a whole number
ALL	T	Array reduced by AND operator
ALLOCATED	I	Allocation status of allocatable variable
ANINT	E	Nearest whole number
ANY	T	Array reduced by OR operator
ASIN	E	function
ASINH	E	Inverse hyperbolic sine function
ASSOCIATED	I	Pointer association status inquiry
ATAN	E	function
ATAN2	E	function
ATANH	E	Inverse hyperbolic tangent function
ATOMIC_ADD	A	Atomic addition
ATOMIC_AND	A	Atomic bitwise AND
ATOMIC_CAS	A	Atomic compare and swap
ATOMIC_DEFINE	A	Define a variable atomically
ATOMIC_FETCH_ADD	A	Atomic fetch and add
ATOMIC_FETCH_AND	A	Atomic fetch and bitwise AND
ATOMIC_FETCH_OR	A	Atomic fetch and bitwise OR
ATOMIC_FETCH_XOR	A	Atomic fetch and bitwise exclusive OR
ATOMIC_OR	A	Atomic bitwise OR
ATOMIC_REF	A	Reference a variable atomically
ATOMIC_XOR	A	Atomic bitwise exclusive OR
BESSEL_J0	E	Bessel function of the 1st kind, order 0
BESSEL_J1	E	Bessel function of the 1st kind, order 1
BESSEL_JN	E	Bessel function of the 1st kind, order N
BESSEL_JN	T	Bessel functions of the 1st kind
BESSEL_Y0	E	Bessel function of the 2nd kind, order 0
BESSEL_Y1	E	Bessel function of the 2nd kind, order 1
BESSEL_YN	E	Bessel function of the 2nd kind, order N
BESSEL_YN	T	Bessel functions of the 2nd kind
BGE	E	Bitwise greater than or equal to
BGT	E	Bitwise greater than
BIT_SIZE	I	Number of bits in integer model

Table D.4 (continued)

BLE	E	Bitwise less than or equal to
BLT	E	Bitwise less than
BTEST	E	Test single bit in an integer
CEILING	E	Least integer greater than or equal to A
CHAR	E	Character from code value
CMPLX	E	Conversion to complex type
CO_BROADCAST	C	Broadcast value to images
CO_MAX	C	Compute maximum value across images
CO_MIN	C	Compute minimum value across images
CO_REDUCE	C	Generalized reduction across images
CO_SUM	C	Compute sum across images
COMMAND_ARGUMENT_COUNT	T	Number of command arguments
CONJG	E	Conjugate of a complex number
COS	E	Cosine function
COSH	E	Hyperbolic cosine function
COSHAPE	I	Sizes of codimensions of a coarray
COUNT	T	Logical array reduced by counting true values
CPU_TIME	S	Processor time used
CSHIFT	T	Circular shift of an array
DATE_AND_TIME	S	Date and time
DBLE	E	Conversion to double precision real
DIGITS	I	Significant digits in numeric model
DIM	E	Maximum of X - Y and zero
DOT_PRODUCT	T	Dot product of two vectors
DPROD	E	Double precision real product
DSHIFTL	E	Combined left shift
DSHIFTR	E	Combined right shift
EOSHIFT	T	End-off shift of the elements of an array
EPSILON	I	Model number that is small compared to 1
ERF	E	Error function
ERFC	E	Complementary error function
ERFC_SCALED	E	Scaled complementary error function
EVENT_QUERY	S	Query event count
EXECUTE_COMMAND_LINE	S	Execute a command line
EXP	E	Exponential function
EXPONENT	E	Exponent of floating-point number
EXTENDS_TYPE_OF	I	Dynamic type extension inquiry
FAILED_IMAGES	T	Indices of failed images
FINDLOC	T	Location(s) of a specified value
FLOOR	E	Greatest integer less than or equal to A

Table D.4 (continued)

FRACTION	E	Fractional part of number
GAMMA	E	Gamma function
GET_COMMAND	S	Get program invocation command
GET_COMMAND_ ARGUMENT	S	Get program invocation argument
GET_ENVIRONMENT_VARIABLE	S	Get environment variable
GET_TEAM	T	Team
HUGE	I	Largest model number
HYPOT	E	Euclidean distance function
IACHAR	E	ASCII code value for character
IALL	T	Array reduced by IAND function
IAND	E	Bitwise AND
IANY	T	Array reduced by IOR function
IBCLR	E	I with bit POS replaced by zero
IBITS	E	Specified sequence of bits
IBSET	E	I with bit POS replaced by one
ICHAR	E	Code value for character
IEOR	E	Bitwise exclusive OR
IMAGE_INDEX	I	Image index from cosubscripts
IMAGE_STATUS	T	Image execution state
INDEX	E	Character string search
INT	E	Conversion to integer type
IOR	E	Bitwise inclusive OR
IPARITY	T	Array reduced by IEOR function
ISHFT	E	Logical shift
ISHFTC	E	Circular shift of the rightmost bits
IS_CONTIGUOUS	I	Array contiguity test
IS_IOSTAT_END	E	IOSTAT value test for end of file
IS_IOSTAT_EOR	E	IOSTAT value test for end of record
KIND	I	Value of the kind type parameter of X
LBOUND	I	Lower bound(s)
LCOBOUND	I	Lower cobound(s) of a coarray
LEADZ	E	Number of leading zero bits
LEN	I	Length of a character entity
LEN_TRIM	E	Length without trailing blanks
LGE	E	ASCII greater than or equal
LGT	E	ASCII greater than
LLE	E	ASCII less than or equal
LLT	E	ASCII less than
LOG	E	Natural logarithm
LOG_GAMMA	E	Logarithm of the absolute value of the gamma function

Table D.4 (continued)

LOG10	E	Common logarithm
LOGICAL	E	Conversion between kinds of logical
MASKL	E	Left justified mask
MASKR	E	Right justified mask
MATMUL	T	Matrix multiplication
MAX	E	Maximum value
MAXEXPONENT	I	Maximum exponent of a real model
MAXLOC	T	Location(s) of maximum value
MAXVAL	T	Maximum value(s) of array
MERGE	E	Expression value selection
MERGE_BITS	E	Merge of bits under mask
MIN	E	Minimum value
MINEXPONENT	I	Minimum exponent of a real model
MINLOC	T	Location(s) of minimum value
MINVAL	T	Minimum value(s) of array
MOD	E	Remainder function
MODULO	E	Modulo function
MOVE_ALLOC	PS	Move an allocation
MVBITS	ES	Copy a sequence of bits
NEAREST	E	Adjacent machine number
NEW_LINE	I	Newline character
NINT	E	Nearest integer
NORM2	T	L2 norm of an array
NOT	E	Bitwise complement
NULL	T	Disassociated pointer or unallocated allocatable entity
NUM_IMAGES	T	Number of images
OUT_OF_RANGE	E	Whether a value cannot be converted safely
PACK	T	Array packed into a vector
PARITY	T	Array reduced by NEQV operator
POPCNT	E	Number of one bits
POPPAR	E	Parity expressed as 0 or 1
PRECISION	I	Decimal precision of a real model
PRESENT	I	Presence of optional argument
PRODUCT	T	Array reduced by multiplication
RADIX	I	Base of a numeric model
RANDOM_INIT	S	Initialise the pseudorandom number generator
RANDOM_NUMBER	S	Generate pseudorandom number(s)
RANDOM_SEED	S	Restart or query the pseudorandom number generator
RANGE	I	Decimal exponent range of a numeric model

Table D.4 (continued)

RANK	I	Rank of a data object
REAL	E	Conversion to real type
REDUCE	T	General reduction of array
REPEAT	T	Repetitive string concatenation
RESHAPE	T	Arbitrary shape array construction
RRSPACING	E	Reciprocal of relative spacing of model numbers
SAME_TYPE_AS	I	Dynamic type equality test
SCALE	E	Real number scaled by radix power
SCAN	E	Character set membership search
SELECTED_CHAR_KIND	T	Character kind selection
SELECTED_INT_KIND	T	Integer kind selection
SELECTED_REAL_KIND	T	Real kind selection
SET_EXPONENT	E	Real value with specified exponent
SHAPE	I	Shape of an array or a scalar
SHIFTA	E	Right shift with fill
SHIFTL	E	Left shift
SHIFTR	E	Right shift
SIGN	E	Magnitude of A with the sign of B
SIN	E	Sine function
SINH	E	Hyperbolic sine function
SIZE	I	Size of an array or one extent
SPACING	E	Spacing of model numbers
SPREAD	T	Value replicated in a new dimension
SQRT	E	Square root
STOPPED_IMAGES	T	Indices of stopped images
STORAGE_SIZE	I	Storage size in bits
SUM	T	Array reduced by addition
SYSTEM_CLOCK	S	Query system clock
TAN	E	Tangent function
TANH	E	Hyperbolic tangent function
TEAM_NUMBER	T	Team number
THIS_IMAGE	T	Index of the invoking image
THIS_IMAGE	T	Cosubscript(s) of this image
TINY	I	Smallest positive model number
TRAILZ	E	Number of trailing zero bits
TRANSFER	T	Transfer physical representation
TRANSPOSE	T	Transpose of an array of rank two
TRIM	T	String without trailing blanks
UBOUND	I	Upper bound(s)
UCOBOUND	I	Upper cobound(s) of a coarray
UNPACK	T	Vector unpacked into an array
VERIFY	E	Character manipulation

D.9 Intrinsic Function Examples

In this section we provide coverage of a large subset of the intrinsic functions and procedures.

ABS(a) : Absolute value.

argument: a	**type**: n
result: as argument	**class**: e

Note(s):
If `a` is `complex(x,y)` then the functions returns $\sqrt{x^2 + y^2}$
Example(s): `r1=abs(a)`

ACHAR(i, *kind*) : Returns character in the ASCII character set.

argument: i	**type**: i
result: char	**class**: e

Note(s):
Inverse of the `iachar` function.
Example(s): `c=achar(i)`

ACOS(x) : Arccosine, inverse cosine.

argument: x	**type**: r,c
result: as argument	**class**: e

Note(s):
$|x| <= 1$
Example(s): `y=acos(x)`

ACOSH(x) : Inverse hyperbolic cosine function.

argument: x	**type**: r,c
result: as argument	**class**: e

Example(s): `y = acosh(x)`

ADJUSTL(string) : Adjust `string` left, removing leading blanks and inserting trailing blanks.

argument: string	**type**: s
result: as argument	**class**: e

Example(s): `s=adjustl(s)`

ADJUSTR(string) : Adjust `string` right, removing trailing blanks and inserting leading blanks.

argument: string	**type**: s
result: as argument	**class**: e

Example(s): `s=adjustr(s)`

AIMAG(z) : Imaginary part of complex argument.

argument: z	**type**: c
result: as argument	**class**: e

Example(s): `y=aimag(z)`

AINT(a, *kind*) : Truncation toward zero to a whole number.

argument: a	**type**: r
result: as a	**class**: e

Example(s): `y=aint(z)`

```
  z       y
 0.3      0
 2.73     2.0
-2.73    -2.0
```

ALL(mask, *dim*) : Determines whether all values are true in `mask`.

argument: mask	**type**: l
result: l	**class**: t

Note(s):
`dim` is optional and must be a scalar in the range $1 <= dim <= n$ where n is the rank of mask. The result is scalar if `dim` is absent or mask has rank 1. Otherwise it works on the dimension `dim` of mask and the result is an array of rank $n - 1$

Example(s): `t=all(m)`

ALLOCATED(variable) : Returns true if and only if the allocatable variable is allocated.

argument: variable	**type**: any
result: l	**class**: i

Note(s):
`variable` must be declared with the allocatable attribute and can be an array or a scalar.
Example(s): `if (allocated(array) ) then ...`

ANINT(a, *kind*) : Nearest whole number.

argument: a	**type**: r
result: as a	**class**: e

Example(s): `z=anint(a)`

```
     a      z
    5.63   6
   -5.7   -6.0
```

ANY(mask, *dim*) : Determines whether any value is true in `mask` along dimension `dim`.

argument: mask	**type**: l
result: l	**class**: t

Note(s):
`mask` must be an array. The result is a scalar if `dim` is absent or if `mask` is of rank 1. Otherwise it works on the dimension `dim` of `mask` and the result is an array of rank $n - 1$
Example(s): `t=any(a)`

ASIN(x) : Arcsine.

argument: x	**type**: r,c
result: as argument	**class**: e

Example(s): `z=asin(x)`

ASINH(x) : Inverse hyperbolic sine function.

argument: x	**type**: r,c
result: as argument	**class**: e

Example(s): `y = asinh(x)`

ASSOCIATED(pointer, *target*) : Returns the association status of the pointer.

argument: pointer	**type**: p
result: l	**class**: i

Note(s):
1. If `target` is absent then the result is true if pointer is associated with a target, otherwise false.
2. If `target` is present and is a target, the result is true if pointer is currently associated with target and false if it is not.
3. If `target` is present and is a pointer, the result is true if both pointer and target are currently associated with the same target, and is false otherwise. If either `pointer` or `target` is disassociated the result is false.
Example(s): `t=associated(p)`

ATAN(x) or
ATAN(y,x) : Arctangent.

argument: x	**type**: r,c
argument: y	**type**: r
result: as argument	**class**: e

Note(s):
If `y` appears, `x` shall be of type real with the same kind type parameter as `y`.
If `y` has the value zero, `x` shall not have the value zero.
If `y` does not appear, `x` shall be of type real or complex.
Example(s): `z=atan(x)`

ATAN2(y,x) : Arctangent of y / x.

argument: y	**type**: r
result: as arguments	**class**: e

Example(s): `z=atan2(y,x)`

ATANH(x) : Inverse hyperbolic tangent function.

argument: x	**type**: r,c
result: as argument	**class**: e

Example(s): `y = atanh(x)`

BESSEL_J0(x) : Bessel function of the first kind, order 0.

argument: x	**type**: r
result: as argument	**class**: e

Example(s): `y = bessel_j0(1.0)` has the value 0.765 (approximately)

BESSEL_J1(x) : Bessel function of the first kind, order 1.

argument: x	**type**: r
result: as argument	**class**: e

Example(s): `y = bessel_j1(1.0)` has the value 0.440 (approximately).

BESSEL_JN(n, x) : Bessel functions of the first kind. Elemental
BESSEL_JN(n1,n2,x) : Bessel function of the first kind. Transformational.

arguments: n	**type**: i
arguments: n1	**type**: i
arguments: n2	**type**: i
arguments: x	**type**: r
result: as x	**class**: e or t

Note(s):
`n` shall be nonnegative.
`n1` shall be nonnegative.
`n2` shall be nonnegative.
`x` if the function is transformational, `x` shall be scalar.

Additional Note(s):
The result of `bessel_jn(n, x)` is processor dependent approximation to the Bessel function of the first kind and order `n` of `x`.

Element i of the result value of `bessel_jn(n1,n2,x)` is a processor dependent approximation to the bessel function of the first kind and order $n1 + i - 1$ of x.
Example(s): `y = bessel_jn(2, 1.0)` has the value 0.115 (approximately).

BESSEL_Y0(x) : Bessel function of the second kind, order 0.

argument: x	**type**: r
result: as argument	**class**: e

Example(s): `y = bessel_y0(1.0)` has the value 0.088 (approximately).

BESSEL_Y1(x) : Bessel function of the second kind, order 1.

argument: x	**type**: r
result: as argument	**class**: e

Example(s): `y = bessel_y1(1.0)` has the value −0.781 (approximately).

BESSEL_YN(n1,n2,x) Bessel functions of the second kind. Transformational.
BESSEL_YN(n, x) : Bessel functions of the second kind. Elemental.

arguments: n	**type**: i
arguments: n1	**type**: i
arguments: n2	**type**: i
arguments: x	**type**: r
result: as x	**class**: e or t

Note(s):
`n` nonnegative.
`n1` nonnegative.
`n2` nonnegative.
`x` if the function is transformational, `x` shall be scalar. Its value shall be greater than zero.
Example(s): `y = bessel_yn(2, 1.0)` has the value -1.651 (approximately).

BGE(i, j) : True if `i` is bitwise greater than or equal to `j`.

arguments: i,j	**type**: i or boz
result: l	**class**: e

Example(s): If `bit_size(j)` has the value 8, `bge(z'ff', j)` has the value true for any value of j. `bge(0, -1)` has the value false.

BGT(i, j) : True if `i` is bitwise greater than `j`

arguments: i,j	**type**: i or boz
result: l	**class**: e

The result is true if the sequence of bits represented by `i` is greater than the sequence of bits represented by `j`, according to the method of bit sequence comparison in 16.3.2 of the standard; otherwise the result is false.
Example(s): `bgt(z'ff', z'fc')` has the value true. `bgt(0, -1)` has the value false.

BLE(i, j) : True if `i` is bitwise less than or equal to `j`.

arguments: i,j	**type**: i or boz
result: l	**class**: e

The result is true if the sequence of bits represented by `i` is less than or equal to the sequence of bits represented by `j`, according to the method of bit sequence comparison in 16.3.2 of the standard; otherwise the result is false.
Example(s): `ble(0, j)` has the value true for any value of j. `ble(-1, 0)` has the value false.

BLT(i, j) : Bitwise less than.

arguments: i,j	**type**: i or boz
result: l	**class**: e

The result is true if the sequence of bits represented by `i` is less than the sequence of bits represented by `j`, according to the method of bit sequence comparison in 16.3.2 of the standard; otherwise the result is false.
Example(s): `blt(0, -1)` has the value true. `blt(z'ff', z'fc')` has the value false.

BIT_SIZE(i) : Returns the number of bits, as defined by the bit model of Sect. 16.3 of the standard.

argument: i	**type**: i
result: as argument	**class**: i

Example(s): `n_bits=bit_size(i)`

BTEST(i, pos) : True if and only if a specified bit of an integer value is one.

argument: i	**type**: i
result: l	**class**: e

Example(s): `t=btest(i,pos)`

CEILING(a, *kind*) : Least integer greater than or equal to `a`.

argument: a	**type**: r
result: i	**class**: e

Note(s):
If `kind` is present the result has the kind type parameter `kind`. otherwise the result is of type default integer.
Example(s): `i=ceiling(a)` If a = 12.21 then i = 13, if a = −3.16 then i = −3.

CHAR(i, *kind)* : Returns the character in a given position in the processor collating sequence associated with the specified `kind` type parameter. It is the inverse of the `ICHAR` function.

argument: i	**type**: i
result: char	**class**: e

Note(s):
ASCII is the default character set.
Example(s): `c=char(65)` and for the ASCII character set c='a'.

CMPLX(x,*kind*) or
CMPLX(x, *y, kind*) : Converts to complex type.
First form.

argument: x	**type**: c
result: c	**class**: e

Second form.

argument: x	**type**: i, r, boz
argument: y	**type**: i, r boz
result: c	**class**: e

Note(s):

1. The result is of type complex. If `kind` is present, the kind type parameter is that specified by the value of `kind`; otherwise, the kind type parameter is that of default real kind

2. If Y is absent and X is not complex, it is as if Y were present with the value zero. If `kind` is absent, it is as if `kind` were present with the value `kind` (0.0). If X is complex, the result is the same as that of `cmplx (real (x), aimag (x), kind)`. The result of `cmplx (x, y, kind)` has the complex value whose real part is `real (x, kind)` and whose imaginary part is `real (y, kind)`.

Example(s): `z=cmplx(x,y)`

COMMAND_ARGUMENT_COUNT( ) : Number of command arguments.

arguments: none	**result**: i
class: t	

The result value is equal to the number of command arguments available. If there are no command arguments available or if the processor does not support command arguments, then the result has the value zero. if the processor has a concept of a command name, the command name does not count as one of the command arguments.

Example(s): `i = command_argument_count( )`

CONJG(z) : Conjugate of a complex argument.

argument: z	**type**: c
result: as z	**class**: e

Example(s): `z1=conjg(z)`

COS(x) : Cosine.

argument: x	**type**: r, c
result: as argument	**class**: e

Note(s):

The arguments of all trigonometric functions should be in radians, not degrees.

Example(s): `a=cos(x)`

COSH(x) : Hyperbolic cosine.

argument: x	**type**: r,c
result: as argument	**class**: e

Example(s): `z=cosh(x)`

COUNT(mask, *dim, kind*) : Returns the number of true elements in mask along dimension `dim`.

argument: mask	**type**: l
result: i	**class**: t

Note(s):
`dim` must be a scalar in the range $1 <= dim <= n$, where n is the rank of mask. The result is scalar if `dim` is absent or mask has rank 1. Otherwise it works on the dimension `dim` of mask and the result is an array of rank $n - 1$
Example(s): `n=count(a)`

CPU_TIME(time) : Returns the processor time.

argument: time	**type**: r
result: n/a	**class**: s

Example(s): `call cpu_time(time)`

CSHIFT(array, shift, *dim)* : Circular shift on a rank 1 array or rank 1 sections of higher-rank arrays.

argument: array	**type**: any
result: as array	**class**: t

Note(s):
`array` must be an array
`shift` must be a scalar if array has rank 1, otherwise it is an array of rank $n - 1$, where n is the rank of `array`.
`dim` must be a scalar with a value in the range $1 < dim <= n$.
Example(s): `array=cshift(array,10)`

DATE_AND_TIME(*date, time, zone, values)* : Returns the current date and time (compatible with ISO 8601:1988).

Note(s):

1. `Date` is optional and must be scalar and 8 characters long in order to return the complete value of the form `ccyymmdd`, where `cc` is the century, `yy` is the year, `mm` is the month and `dd` is the day. It is `intent(out)`.
2. `Time` is optional and must be scalar and 10 characters long in order to return the complete value of the form `hhmmss.sss` where `hh` is the hour, `mm` is the minutes and `ss.sss` is the seconds and milliseconds. It is `intent(out)`.
3. `Zone` is optional and must be scalar and must be 5 characters long in order to return the complete value of the form hhmm where hh and mm are the time differences with respect to coordinated universal time in hours and minutes. It is `intent(out)`.
4. `Values` is optional and a rank 1 array of size 8. It is `intent(out)`. The values returned are as shown below.

```
values(1)   year
values(2)   month
values(3)   day
values(4)   time with respect to coordinated
            universal time in minutes.
values(5)   hour (24 hour clock)
values(6)   minutes
values(7)   seconds
values(8)   milliseconds in the range 0 - 999.
```

Example(s): `call date_time(d,t,z,v)`

DBLE(a) : Converts to double precision from integer, real, and complex

argument: a	**type**: n
result: dp	**class**: e

Example(s): `d=dble(a)`

DIGITS(x) : Returns the number of significant digits of the argument as defined in the numeric models for integer and reals in Chap. 5.

argument: x	**type**: i,r
result: i	**class**: i

Example(s): `i=digits(x)`

DIM(x,y) : Difference of two values if positive or zero otherwise.

argument: x	**type**: i
result: as arguments	**class**: e

Example(s): `z=dim(x,y)`

DOT_PRODUCT(vector_1,vector_2) : Performs the mathematical dot product of two rank 1 arrays.

argument: vector_ 1	**type**: n
argument: vector_ 2	**type**: n
result: as arguments	**class**: t

`vector_2` is as `vector_1.`
Note(s):
1. For integer and real `vector_1` result has the value `sum(vector_1*vector_2).`
2. For complex `vector_1` result has the value `sum(conjg(vector_1)*vector_2).`
3. For logical `vector_1` result has the value `any(vector_1 .and. vector_2).`
Example(s): `a=dot_product(x,y)`

DPROD(x,y) : Double precision product of two reals.

argument: x	**type**: r
result: dp	**class**: e

Example(s): `d=dprod(x,y)`

DSHIFTL(i, j, shift) : Combined left shift.

arguments: i,j	**type**: i or boz
argument: shift	**type**: i
result: See note below.	**class**: e

Note(s):
Result type: Same as i if i is of type integer; otherwise, same as j. If either i or j is a boz-literal-constant, it is first converted as if by the intrinsic function int to type integer with the kind type parameter of the other. The rightmost shift bits of the result value are the same as the leftmost bits of j, and the remaining bits of the result value are the same as the rightmost bits of i. This is equal to ior(shiftl(i, shift), shiftr(j, bit size(j)-shift)). The model for the interpretation of an integer value as a sequence of bits is in Sect. 16.3 of the standard.

Example(s): `dshiftl(1, 2**30, 2)` has the value 5 if default integer has 32 bits. `dshiftl(i, i, shift)` has the same result value as ishftc(i, shift).

DSHIFTR(i, j, shift) : Combined right shift.

arguments: i,j	**type**: i or boz
argument: shift	**type**: i
result: See note below.	**class**: e

Note(s):

Result: Same as i if i is of type integer; otherwise, same as j. If either i or j is a boz-literal-constant, it is first converted as if by the intrinsic function int to type integer with the kind type parameter of the other. The leftmost shift bits of the result value are the same as the rightmost bits of i, and the remaining bits of the result value are the same as the leftmost bits of j. This is equal to ior(shiftl(i, bit size(i)-shift), shiftr(j, shift)). The model for the interpretation of an integer value as a sequence of bits is in 16.3 of the standard.
Example(s): `dshiftr(1, 16, 3)` has the value 229 +2 if default integer has 32 bits. `dshiftr(i, i, shift)` has the same result value as ishftc(i,-shift).

EOSHIFT(array, shift, *boundary, dim*) : End of shift of a rank 1 array or rank 1 section of a higher-rank array.

argument: array	**type**: any
argument: shift	**type**: n
argument: boundary	**type**: n
result: as array	**class**: t

Note(s):

1. `boundary` is as array.
2. `array` must be an array, `shift` must be a scalar if array has rank 1, otherwise it is an array of rank $n - 1$, where n is the rank of array.
3. `boundary` shall be of the same type and type parameters as `array`. It must be scalar if array has rank 1, otherwise it must be either scalar or of rank $n - 1$. See section 16.9.67 of the standard for additional information.
4. `dim` must be a scalar with a value in the range $1 <= dim <= n$.

Example(s): `a=eoshift(a,shift)`

EPSILON(x) : Smallest difference between two reals of that kind. See Chap. 5 and real numeric model.

argument: x	**type**: r
result: as argument	**class**: i

Example(s): `tiny=epsilon(x)`

ERF(x) : Error function.

argument: x	**type**: r
result: as x	**class**: e

Example(s): `y = erf(1.0)` has the value 0.843 (approximately).

ERFC(x) : Complementary error function.

argument: x	**type**: r
result: as x	**class**: e

Example(s): `y = erfc(1.0)` has the value 0.157 (approximately).

ERFC_SCALED(x) : Scaled complementary error function.

argument: x	**type**: r
result: as x	**class**: e

Example(s): `y = erfc_scaled(20.0)` has the value 0.0282 (approximately).

EXECUTE_COMMAND_LINE(command, *wait, exitstat,cmdstat, cmdmsg*) : Execute a command line.
Note(s):

`command` shall be a default character scalar. It is an `intent(in)` argument. Its value is the command line to be executed. the interpretation is processor dependent.

`wait` shall be a default logical scalar. It is an `intent(in)` argument. If `wait` is present with the value false, and the processor supports asynchronous execution of the command, the command is executed asynchronously; otherwise it is executed synchronously.

`exitstat` shall be a default integer scalar. It is an intent(inout) argument. If the command is executed synchronously, it is assigned the value of the processor-dependent exit status. Otherwise, the value of `exitstat` is unchanged.

`cmdstat` shall be a default integer scalar. It is an `intent(out)` argument. It is assigned the value -1 if the processor does not support command line execution, a processor-dependent positive value if an error condition occurs, or the value -2 if no error condition occurs but `wait` is present with the value false and the processor does not support asynchronous execution. otherwise it is assigned the value 0.

`cmdmsg` shall be a default character scalar. It is an `intent(inout)` argument. If an error condition occurs, it is assigned a processor-dependent explanatory message. otherwise, it is unchanged.

Example(s): `call execute_command_line('pwd')` will print the full pathname of the current directory under unix and an error message from windows.

EXP(x) : Exponential. e^x

argument: x	**type**: r, c
result: as argument	**class**: e

Example(s): `y=exp(x)`

EXPONENT(x) : Returns the exponent component of the argument. See Chap. 5 and the real numeric model.

argument: x	**type**: r
result: i	**class**: e

Example(s): `i=exponent(x)`

EXTENDS_TYPE_OF(a, mold) : Query dynamic type for extension.

arguments: a, mold	**type**: p*
result: l	**class**: i

Note(s):
`a` shall be an object of extensible declared type or unlimited polymorphic. If it is a polymorphic pointer, it shall not have an undefined association status.
`mold` shall be an object of extensible declared type or unlimited polymorphic. If it is a polymorphic pointer, it shall not have an undefined association status.
If `mold` is unlimited polymorphic and is either a disassociated pointer or unallocated allocatable variable, the result is true; otherwise if `a` is unlimited polymorphic and is either a disassociated pointer or unallocated allocatable variable, the result is false; otherwise if the dynamic type of `a` or `mold` is extensible, the result is true if and only if the dynamic type of `a` is an extension type of the dynamic type of `mold`; otherwise the result is processor dependent.
Example(s):

```
if(extends_type_of(a, mold)) then
   print *,'dynamic type of a is an'
   print *,'extension of dynamic type of mold'
end if
```

FINDLOC(array, value, dim, *mask, kind, back*) or
FINDLOC(array, value, *mask, kind, back*) : Location(s) of a specified value.

argument: array	**type**: intrinsic type
argument: value	**type**: as array
argument: dim	**type**: i
argument: mask	**type**: l
argument: kind	**type**: i
argument: back	**type**: l
result: i	**class**: t

Note(s):

1. `dim` shall be an integer scalar with a value in the range $1 <= dim <= n$, where n is the rank of array.

2. `mask` shall be conformable with array.

3. **result characteristics**: If `kind` is present, the kind type parameter is that specified by the value of `kind`; otherwise the kind type parameter is that of default integer type. If `dim` does not appear, the result is an array of rank one and of size equal to the rank of `array`; otherwise, the result is of rank $n-1$ 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`.

Example(s):

```
1. The value of
findloc ([1, 3, 5, 3, 1], value=3)
is [2]
The value of
findloc ([1, 3, 5, 3, 1], value=3, back=.true.)
is [4]
The value of
findloc ([1, 3, 5, 3, 1], value=3, dim=1)
is [2]
```

2. If B has the value

$$\begin{bmatrix} 1 & 2 & -9 \\ 2 & 2 & 6 \end{bmatrix}$$

`findloc (b, value=2, dim=1)` has the value [2, 1, 0] and `findloc (b,value=2, dim=2)` has the value [2, 1]. This is independent of the declared lower bounds for `b`.

FLOOR(a, *kind*) : Returns the greatest integer less than or equal to the argument

argument: a	**type**: r
result: i	**class**: e

Note(s):

If `kind` is present the result has the kind type parameter kind, otherwise the result is of type default integer.

Example(s): `i=floor(a)` when a = 5.2 i has the value 5, when a = − 9.7 i has the value −10.

FRACTION(x) : Returns the fractional part of the real numeric model of the argument. See Chap. 5 and the real numeric model.

argument: x	**type**: r
result: as x	**class**: e

Example(s): `f=fraction(x)`

GAMMA(x) : Gamma function.

argument: x	**type**: r
result: as x	**class**: e

Example(s): `y = gamma(1.0)` has the value 1.000 (approximately).

GET_COMMAND(*command, length, status*) : Query program invocation command.

GET_COMMAND_ARGUMENT(number, *value, length, status*) : Query arguments from program invocation.

GET_ENVIRONMENT_VARIABLE(name, *value, length, status, trim name*) : Query environment variable.

HUGE(x) : Returns the largest number for the kind type of the argument. See Chap. 5 and the real and integer numeric models.

argument: x	**type**: i,r
result: as argument	**class**: i

Example(s): `h=huge(x)`

HYPOT(x, y) : Euclidean distance function.

arguments: x,y	**type**: r
result: r	**class**: e

Example(s): `h = hypot(3.0, 4.0)` has the value 5.0 (approximately).

IACHAR(c) : Returns the position of the character argument in the ASCII collating sequence.

argument: c	**type**: char
result: i	**class**: e

Example(s): `i=iachar('a')` returns the value 65.

IALL(array, dim, *mask*) or
IALL(array, *mask*) : Reduce array with bitwise and operation.
IAND(i,j) : Performs a logical and on the arguments.

argument: i	**type**: i
result: as arguments	**class**: e

Example(s): `k=iand(i,j)`

IANY(array, dim, *mask*) or
IANY(array, *mask*) : Reduce array with bitwise or operation.
IBCLR(i,pos) : Clears one bit of the argument to zero.

argument: i	**type**: i
result: as i	**class**: e

Example(s): `i=ibclr(i,pos)`

IBITS(i,pos,len) : Returns a sequence of bits.

argument: i	**type**: i
result: as i	**class**: e

Example(s): `slice=ibits(i,pos,len)`

IBSET(i,pos) : Sets one bit of the argument to one.

argument: i	**type**: i
result: as i	**class**: e

Note(s):
$0 <= pos <= bit\_size(i)$
Example(s): `i=ibset(i,pos)`

ICHAR(c) : Returns the position of a character in the processor collating sequence associated with the kind type parameter of the argument. Normally the position in the ASCII collating sequence.

argument: c	**type**: char
result: i	**class**: e

Example(s): `i=ichar('a')` would return the value 65 for the ASCII character set.

IEOR(i, j) : Performs an exclusive or on the arguments.

argument: i	**type**: i
result: Same as `i` if `i` is of type integer; otherwise, same as `j`.	**class**: e

Example(s): `i=ieor(i,j)`

IMAGE_INDEX(coarray, sub) or
IMAGE_INDEX(coarray, sub, team) or
IMAGE_INDEX(coarray, sub, team_number) : Convert cosubscripts to image index.

argument: coarray	**type**: co
argument: sub	**type**: rank-one integer array
argument: team	**type**: te
argument: team_ number	**type**: i
result: i	**class**: i

Note(s):
1. `coarray` is of any type.
2. `team` is scalar.
3. `team_number` is scalar.

Example(s):

```
integer, codimension[0:*]:: x
integer, dimension(10,15), &
    codimension[3,0:1,-1:*]:: z
  print*, image_index(x,(/0/));
  print*, image_index(z,(/2,0,-1/))
```

would print 1 and 2 respectively.

INDEX(string, substring, *back, kind*) : Locates one substring in another, i.e., returns position of substring in character expression string.

argument: string	**type**: s
argument: substring	**type**: s
argument: back	**type**: l
result: i	**class**: e

Note(s):

If $len(string) < len(substring)$ the result is zero.

Otherwise, if there is an integer i in the range

$1 <= i <= (len(string) - len(substring) + 1)$

such that `string(i : i + len(substring) - 1)` is equal to `substring`, the result has the value of the smallest such `i` if `back` is absent or present with the value false, and the greatest such `i` if `back` is present with the value true.

If the substring is not found the result is zero.

Example(s):

`where=index(' hello world hello','hello')`

the result 2 is returned.

`where=index(' hello world hello','hello',.true.)`

the result 14 is returned.

INT(a, *kind*) : Converts to integer from integer, real, and complex.

argument: a	**type**: n
result: i	**class**: e

Example(s): `i=int(f)`

IOR(i, j) : Performs an inclusive or on the arguments.

argument: i	**type**: i
result: as i	**class**: e

Example(s): `i=ior(i,j)`

IPARITY(array, dim, *mask*) or
IPARITY(array, *mask*) : Array reduced by `ieor` function. Transformational.

argument: array	**type**: i
argument: dim	**type**: i
argument: mask	**type**: l
result: as i	**class**: e

Note(s):
`dim` integer scalar with a value in the range $1 <= dim <= n$, where `n` is the rank of `array`.
`mask` shall be of type logical and shall be conformable with `array`.
Example(s):

`iparity ([14, 13, 8])` has the value 11.
`iparity ([14, 13, 8], mask=[.true., .false., .true])` has the value 6.

ISHFT(i, shift) : Performs a logical shift. The bits of i are shifted by shift positions.

argument: i	**type**: i
result: as i	**class**: e

Note(s):
$|shift| <= bit\_size(i)$
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.
Example(s): `i=ishft(i,shift)`.

ISHFTC(i, shift, *size*) : Performs a circular shift of the rightmost bits. The size rightmost bits of i are circularly shifted by shift positions.

argument: i	**type**: i
result: i	**class**: e

Note(s):
$|shift| <= size$

The result has the value obtained by shifting the `size` rightmost bits of `i` circularly 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. The unshifted bits are unaltered.

Example(s): `i=ishftc(i,shift,size)`

IS_CONTIGUOUS(array) : Test contiguity of an array.

argument: array	**type**: any
result: l	**class**: i

Example(s):

```
integer,target, dimension(10)::a
integer,pointer,dimension(:) :: p
   p= a(1:10:2); print*,is_contiguous(p)
```

would print ‘f’.

IS_IOSTAT_END(i) : Test iostat value for end-of-file.

argument: i	**type**: i
result: l	**class**: e

Example(s): `is_iostat_end(i)` returns value true if i is an i/o status value that corresponds to an end-of-file condition, and false otherwise.

```
read(unit=1,fmt=*, iostat=ist)y(i)
if (is_iostat_end(ist)) then
  print*,'end of file!'
endif
```

IS_IOSTAT_EOR(i) : Test iostat value for end-of-record.

argument: i	**type**: i
result: l	**class**: e

Example(s): `is_iostat_eor(i)` returns the value true if `i` is an i/o status value that corresponds to an end-of-record condition, and false otherwise.

KIND(x) : Returns the kind type parameter of the argument.

argument: x	**type**: any
result: i	**class**: i

Example(s): `i=kind(x)`

LBOUND(array, *dim, kind*) : Lower bound(s) of an array.

argument: array	**type**: any
result: i	**class**: i

Note(s):
1. `dim` optional. $1 <= dim <= n$ where n is the rank of array. The result is scalar if `dim` is present otherwise the result is an array of rank 1 and size n. The result is scalar if `dim` is present, otherwise a rank 1 array and size n.
2. If `array` is a whole array and either array is an assumed-size array of rank `dim` or dimension `dim` of array has nonzero extent, `lbound (array, dim)` has a value equal to the lower bound for subscript `dim` of `array`. Otherwise the result value is 1.

Example(s): `i=lbound(array)`

LCOBOUND(coarray, *dim, kind*]) : Lower cobound(s) of a coarray.

argument: coarray	**type**: co
argument: dim (optional)	**type**: i
argument: kind(optional)	**type**: i
result: i	**class**: i

Example(s):

```
integer, codimension[:,:], allocatable::a
allocate(a[2:3,7:*])
```

`lcbound(a)` is [2,7] and `lcobound(a,dim=2)` is 7

LEADZ(i) : Number of leading zero bits.

argument: i	**type**: i
result: i	**class**: e

Example(s): `leadz(1)` has the value 31 if bit size(1) has the value 32.

LEN(string) : Length of a character entity.

argument: string	**type**: s
result: i	**class**: i

Example(s): `i=len(string)`

LEN_TRIM(string) : Length of character argument less the number of trailing blanks.

argument: string	**type**: s
result: i	**class**: e

Example(s): `i=len_trim(string)`

LGE(string_1, string_2) :
Lexically greater than or equal to and this is default character or ASCII.

argument: string_ 1	**type**: s,ASCII
result: l	**class**: e

string_2 is of type s.
Example(s): `l=lge(s1,s2)`

LGT(string_1, string_2) : Lexically greater than and this is based on the ASCII collating sequence.

argument: string_ 1	**type**: s

Example(s): `l=lgt(s1,s2)`

LLE(string_1, string_2) : Lexically less than or equal to and this is based on the ASCII collating sequence.

argument: string_ 1	**type**: s
result: l	**class**: e

string_2 is of type s.
Example(s): `l=lle(s1,s2)`

LLT(string_1, string_2) : Lexically less than and this is based on the ASCII collating sequence.

argument: string_ 1	**type**: s
result: l	**class**: e

Example(s): `l=llt(s1,s2)`

LOG(x) : Natural logarithm.

argument: x	**type**: r, c
result: as argument	**class**: e

Example(s): `y=log(x)`

LOG_GAMMA(x) : Logarithm of the absolute value of the gamma function.

argument: x	**type**: r
result: r	**class**: e

Example(s): `log_gamma(3.0)` has the value 0.693 (approximately).

LOG10(x) : Common logarithm, log10

argument: x	**type**: r
result: as argument	**class**: e

Example(s): `y=log10(x)`

LOGICAL(l, *kind*) : Converts between different logical kind types, i.e., performs a type cast.

argument: l	**type**: l
result: l	**class**: e

Example(s): `l=logical(k,kind)`

MASKL(i, *kind*) : Left justified mask.

argument : i	**type** : i
result: i	**class**: e

Example(s): `maskl(4)` has the value `shiftl(15, bit_size(0) - 4)`

MASKR(i, *kind*) : Right justified mask.

argument: i	**type**: i
result: i	**class**: e

Example(s): `maskr(4)` has the value 15.

MATMUL(matrix_1, matrix_2) : Performs mathematical matrix multiplication of the array arguments.

argument: matrix_ 1	**type**: n,l
argument: matrix_ 2	**type**: n,l
result: as arguments	**class**: t

matrix_2 is as matrix_1.
Note(s):
`matrix_a` shall be a rank-one or rank-two array of numeric type or logical type.
`matrix_b` shall be of numeric type if `matrix_a` is of numeric type and of logical type if `matrix_a` is of logical type. It shall be an array of rank one or two.
`matrix_a` and `matrix_b` shall not both have rank one.
The size of the first (or only) dimension of `matrix_b` shall equal the size of the last (or only) dimension of `matrix_a`.
The shape of the result depends on the shapes of the arguments as follows:
If `matrix_a` has shape [n,m] and `matrix_b` has shape [m, k], the result has shape [n, k].
If `matrix_a` has shape [m] and `matrix_b` has shape [m, k], the result has shape [k].
If `matrix_a` has shape [n,m] and `matrix_b` has shape [m], the result has shape [n].
Example(s): `r=matmul(m_1,m_2)`

MAX(a1, a2, *a3,...*) : Returns the largest value.

argument: a1	**type**: i,r,s
result: as arguments	**class**: e

a2, a3,.. are as a1.
Example(s): `a=max(a1,a2,a3,a4)`

MAXEXPONENT(x) : Returns the maximum exponent. See Chap. 5 and numeric models.

argument: x	**type**: r
result: i	**class**: i

Example(s): `i=maxexponent(x)`

MAXLOC(array, *mask, kind, back*) or
MAXLOC(array, dim, *mask, kind, back*) : Location(s) of maximum value.

argument: array	**type**: i,r,s
argument: dim	**type**: i
argument: mask	**type**: l
argument: kind	**type**: i
argument: back	**type**: l
result: i	**class**: t

Note(s):
1. `dim` shall be an integer scalar with a value in the range $1 <= dim <= n$, where n is the rank of `array`. The corresponding actual argument shall not be an optional dummy argument.
2. `mask` shall be of type logical and shall be conformable with `array`.
3. `kind` shall be a scalar integer constant expression.
4. `back` shall be scalar and of type logical.

Example(s):

```
a=(/5,6,7,8/)
i=maxloc(a)
```

is (4), which is the subscript of the location of the first occurrence of the maximum value in the rank 1 array.
If

$$A = \begin{pmatrix} 1 & 8 & 5 \\ 9 & 3 & 6 \\ 4 & 2 & 7 \end{pmatrix}$$

`i = maxloc(a, dim=1)`
is (2,1,3) returning the position of the largest in each column.
`i = maxloc(a, dim=2)`
is (2,1,3) returning the position of the largest in each row.

MAXVAL(array, *mask*) or
MAXVAL(array, dim, *mask*) : Maximum value(s) of `array`.

argument: array	**type**: i,r,s
argument: mask	**type**: l
argument: dim	**type**: i
result: as argument	**class**: t

Note(s):
1. `dim` shall be an integer scalar with a value in the range $1 <= dim <= n$, where n is the rank of `array`.
2. `mask` (optional) shall be of type logical and shall be conformable with `array`.
Example(s): `maxval((/1,2,3/))` returns the value 3.

```
maxval( c , mask = c < 0.0)
```

returns the maximum of the negative elements of c.
For

$$B = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$$

`maxval(b, dim=1)` returns(2,4,6)
`maxval(b, dim=2)` returns(5,6)

MERGE(true, false, mask) : Chooses alternative values according to the value of a mask.

argument: true	**type**: any
result: as true	**class**: e

Example(s): For

$$true = \begin{pmatrix} 2 & 6 & 10 \\ 4 & 8 & 12 \end{pmatrix}, false = \begin{pmatrix} 1 & 5 & 9 \\ 3 & 7 & 11 \end{pmatrix}, and\ mask = \begin{pmatrix} T & F & T \\ F & T & F \end{pmatrix}$$

$$result = \begin{pmatrix} 2 & 5 & 10 \\ 3 & 8 & 11 \end{pmatrix}$$

MERGE_BITS(i, j, mask) : Merge of bits under mask.

argument: i	**type**: i or boz
argument: j i or boz	
argument: mask i or boz	
result: same as i if integer, otherwise same as j.	
class: e	

Example(s): `merge_bits(14,18,22)` has the value 6.

MIN(a1, a2, a3,...) : Chooses the smallest value.

argument: a1	**type**: i, r, s
result: as arguments	**class**: e

Example(s): `y=min(x1,x2,x3,x4,x5)`

MINEXPONENT(x) : Returns the minimum exponent. See Chap. 5 and numeric models.

argument: x	**type**: r
result: i	**class**: i

Example(s): `i=minexponent(x)`

MINLOC(array,*mask,kind,back*) or
MINLOC(array,dim,*mask,kind,back*) : Location of minimum value.

argument: array	**type**: i,r,s
argument: dim	**type**: i
argument: mask	**type**: l
argument: kind	**type**: i
argument: back	**type**: l
result: i	**class**: t

Note(s):
1. `dim` shall be an integer scalar with a value in the range $1 <= dim <= n$, where n is the rank of `array`. The corresponding actual argument shall not be an optional dummy argument.
2. `mask` shall be of type logical and shall be conformable with `array`.
3. `kind` shall be a scalar integer constant expression.
4. `back` shall be scalar and of type logical.

Example(s): `i=minloc(array)`

In the above example if array is a rank 2 array of shape(5,10) and the smallest value is in position(2,1) then the result is the rank 1 array i with shape(2) and i(1) = 2 and i(2) = 1.

MINVAL(array, *mask* or
MINVAL(array, dim, mask) : Minimum value(s) of `array`.

argument: array	**type**: i,r,s
argument: mask	**type**: l
argument: dim	**type**: i
result: as array	**class**: t

Note(s):

1. `dim` shall be an integer scalar with a value in the range $1 <= dim <= n$, where n is the rank of `array`. The corresponding actual argument shall not be an optional dummy argument.

2. `mask` shall be of type logical and shall be conformable with `array`.

Example(s):

`minval((/1,2,3/))` returns the value 1.

`minval(c,mask=c>0.0)` returns the minimum of the positive elements of c.

For

$B = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$

`minval(b,dim=1)` returns(1,3,5).

`minval(b,dim=2)` returns(1,2).

MOD(a, b) : Returns the remainder when first argument divided by second.

argument: a	**type**: i, r
argument: b	**type**: as a
result: as arguments	**class**: e

Note(s):

`b` shall not be zero.

The result is $a - int(a/b) * b$.

Example(s): `r=mod(a,b)`

```
 a   b   r
 8   5   3
-8   5  -3
 8  -5   3
-8  -5  -3
```

MODULO(a, b) : Returns the modulo of the arguments.

argument: a	**type**: i,r
argument: b	**type**: as a
result: as a	**class**: e

Note(s):

b shall not be zero.

If a is of typr integer, modulo (a, b) has the value r such that $a = qb + r$, where q is an integer, the inequalities $0 <= r < b$ hold if $b > 0$, and $b < r <= 0$ hold if $b < 0$.

If a is a type real modulo (a,b) has the result $a - floor(a/b) * b$.

Example(s): r=modulo(a,b)

a	b	r
8	5	3
-8	5	2
8	-5	-2
-8	-5	-3

MOVE_ALLOC (from, to [, stat, errmsg]) : Move an allocation.

Note(s):

1. Subroutine, pure if and only if from is not a coarray.

2. from may be of any type, rank, and corank. It shall be allocatable and shall not be a coindexed object. It is an intent (inout) argument.

3. to shall be type compatible with from and have the same rank and corank. It shall be allocatable and shall not be a coindexed object. It shall be polymorphic if from is polymorphic. It is an intent (out) argument.

4. stat shall be a noncoindexed integer scalar with a decimal exponent range of at least four. It is an intent (out) argument.

5. errmsg shall be a noncoindexed default character scalar. It is an intent (inout) argument.

6. It is expected that the implementation of allocatable objects will typically involve descriptors to locate the allocated storage; move_alloc could then be implemented by transferring the the contents of the descriptor for from to the descriptor for to and clearing the descriptor for from.

Example(s):

```
real,allocatable :: grid(:),tempgrid(:)
...
allocate(grid(-n:n))
! initial allocation of grid
...
allocate(tempgrid(-2*n:2*n))
! allocate bigger grid
tempgrid(::2)=grid
! distribute values to new locations
call move_alloc(to=grid,from=tempgrid)
```

The old grid is deallocated because to is intent (out), and grid then takes over the new grid allocation.

MVBITS(from, frompos, len, to, topos) : Copies a sequence of bits from one data object to another.

argument: from	**type**: i
argument: frompos	**type**: i
argument: len	**type**: i
argument: to	**type**: i
argument: topos	**type**: i
result: n/a	**class**: s

All arguments are of integer type.

Note(s):

`from` It is an `intent(in)` argument.

`frompos` shall be nonnegative. It is an `intent(in)` argument. $frompos + len <= bit\_size(from)$.

`len` shall be nonnegative. It is an `intent(in)` argument.

`to` shall be a variable of the same type and kind type parameter value as `from` and may be associated with `from`. It is an `intent(inout)` argument.

`to` is defined 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 the len bits of `from` starting at `frompos` had on entry.

`topos` shall be nonnegative. It is an `intent(in)` argument. $topos + len <= bit\_size(to)$.

Example(s): If `to` has the initial value 6, the value of `to` after the statement `call mvbits (7, 2, 2, to, 0)` is 5.

Example(s): `call mvbits(f,fp,l,t,tp)`

NEAREST(x,next) : Returns the nearest different number. See Chap. 5 and the real numeric model.

argument: x	**type**: r
argument: next	**type**: r
result: as x	**class**: e

Note(s):

`next` Not equal to zero.

The result has a value equal to the machine-representable number distinct from `x` and nearest to it in the direction of infinity with the same sign as `next`.

Unlike other floating point manipulation functions, `nearest` operates on machine representable numbers rather than model numbers. On many systems there are machine representable numbers that lie between adjacent model numbers.

Example(s): `n=nearest(x,next)`

NEW_LINE(a) : Returns newline character used for formatted stream output.

argument: a	**type**: char
result: char	**class**: i

Note(s):

If `a` is default character and the character in position 10 of the ASCII collating sequence is representable in the default character set, then the result is `achar (10)`.

If `a` is ASCII character or ISO 10646 character, then the result is `char (10, kind (a))`.

Otherwise, the result is a processor-dependent character that represents a newline in output to files connected for formatted stream output if there is such a character.

Otherwise, the result is the blank character.

Example(s):

```
open(2,file='nline.txt', access='stream', form='formatted')
write(2,'(a)')'hola'//new_line('a')//'mundo'
```

This will write 2 lines to the file nline.txt.

NINT(a, *kind*) : Yields nearest integer.

argument: a	**type**: r
result: i	**class**: e

Note(s):

1. $a > 0$, the result is int(a+0.5).
2. $a <= 0$, the result is int(a-0.5).

Example(s): `i=nint(x)`

NORM2(x) or
NORM2(x, *dim*) : Norm of an array.

argument: x	**type**:*r*
argument: dim	**type**: i
result: r	**class**: t

Note(s):

1. `dim` shall be an integer scalar with a value in the range $1 <= dim <= n$, where n is the rank of `x`. The corresponding actual argument shall not be an optional dummy argument.

2. The result of `norm2(x)` has a value equal to a processor-dependent approximation to the generalized l2 norm of `x`, which is the square root of the sum of the squares of the elements of `x`.

3. If `dim` is present the array is reduced as for `sum(x,dim)` except that `norm2` is applied to the reduced vectors.

Example(s): See below.

```
norm2([3.0, 4.0]) is 5.0.
If x has the value
  1.0 2.0
  3.0 4.0
norm2(x,dim=1) is [3.162, 4.472]
norm2(x,dim=2) is [2.236,5.0]
approximately.
```

NOT(i) : Returns the logical complement of the argument.

argument: i	**type**: i
result: as i	**class**: e

Example(s): `i=not(i)`

NULL(*mold*) : Returns a disassociated pointer.

argument: mold	**type**: p
result: as argument	**class**: t

Note(s):

If the argument mold is present the result is the same as mold. Otherwise it is determined by context.

Example(s): `real, pointer :: p=>null()`

NUM_IMAGES( ) or
NUM_IMAGES(team) or
NUM_IMAGES(team_number) : Number of images.

argument: none	
argument: team	**type**: te
argument: team_ number	**type**: i
result: i	**class**: t

Notes(s):

`team` shall be a scalar of type `team_type` from the intrinsic module `iso_fortran_env`, with a value that identifies the current or an ancestor team.

`team_number` shall be an integer scalar. It shall identify the initial team or a team whose parent is the same as that of the current team.

The result is the number of images in the specified team, or in the current team if no team is specified.

Example(s): `print*,' number of images = ',num_images( )`

The following code uses image 1 to read data and broadcast it to other images.

```
REAL :: P[*]
  IF (THIS_IMAGE()==1) THEN
    READ (6,*) P
    DO I = 2, NUM_IMAGES()
      P[I] = P
    END DO
  END IF
SYNC ALL
```

OUT_OF_RANGE (x, mold [, round]) Whether a value cannot be converted safely.

argument: x	**type**: i,r
argument: mold	**type**: i,r scalar
argument: round	**type**: l scalar
result: l	**class**: e

Note(s):

1. `mold` If it is a variable, it need not be defined.
2. `round` shall be present only if `x` is of type real and `mold` is of type integer.

Example(s): If INT8 is the kind value for an 8-bit binary integer type, OUT_OF_RANGE (-128.5, 0_INT8) will have the value false and OUT_OF_RANGE (-128.5, 0_INT8, .TRUE.) will have the value true.

PACK(array, mask, *vector)* : Packs an array into an array of rank 1, under the control of a mask.

argument: array	**type**: any
argument: mask	**type**: l
argument: vector	**type**: same type as array
result: as array	**class**: t

Note(s):

1. `array` must be an array.

2. `mask` be conformable with `array`.

3. `vector` must have rank 1 and have at least as many elements as there are true elements in mask.

4. If `mask` is scalar with the value `true` vector must have at least as many elements as there are in `array`.

5. The result is an array of rank 1.

6. If `vector` is present the result size is that of `vector`.

7. If `vector` is not present the result size is t, the number of true elements in `mask`, unless `mask` is scalar with a value true in which case the result size is the size of `array`.

Example(s): `r=pack(a,m)`

The nonzero elements of an array `m` with the value

```
0 0 0
9 0 0
0 0 7
```

can be *gathered* by the function `pack`. The result of
`pack (m, mask = m/= 0)` is `[9, 7]` and the result of
`pack (m, m /= 0, vector = [2, 4, 6, 8, 10, 12])`
is `[9, 7, 6, 8, 10, 12]`.

PARITY(mask, *dim*) : Reduce array with `.neqv.` operation.

argument:`mask`	**type**: l array
argument:`dim` shall be an integer scalar in the range $1 <= dim <= n$ where n is rank of `mask`.	

Example(s): If `t` has the value `true` and `f` has the value `false`
`parity([t,t,t,f])` is `true`.

POPCNT(i) : Number of one bits in the sequence of bits of `i`.

argument: i	**type**: i
result: i	**class**: e

Example(s): `popcnt([1, 2, 3, 4, 5, 6, 7])`
has the value [1, 1, 2, 1, 2, 2, 3].

POPPAR(i) : Returns the parity of the bit count of an integer expressed as 0 or 1.

argument: i	**type**: i
result: i	**type**: e

Example(s): `poppar([1, 2, 3, 4, 5, 6, 7])`
has the value [1, 1, 0, 1, 0, 0, 1]

PRECISION(x) : Returns the decimal precision of the argument. See Chap. 5 and numeric models.

argument: x	**type**: r, c
result: i	**class**: i

Example(s): `i=precision(x)`

PRESENT(a) : Returns whether an optional argument is present.

argument: a	**type**: any
result: l	**class**: i

Note(s):
`a` must be an optional argument of the procedure in which the `present` function reference appears.
Example(s): `if(present(a)) then`

PRODUCT(array, *mask*) or
PRODUCT(array, dim, *mask)*
The product of all of the elements of `array` along the dimension `dim` corresponding to the true elements of `mask`.

argument: array	**type**: n
argument: dim	**type**: i
argument: mask	**type**: l
result: as array	**class**: t

Note(s):
1. `array` must be an array.
2. 1 $<= dim <= n$ where n is the rank of array.
3. `mask` must be conformable with array.

Example(s):
`product((/1,2,3/))` the result is 6.
`product( c , mask = c > 0.0)` forms the product of the positive elements of c.

If

$$B = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$$

`product(b,dim=1)`
is (2,12,30)
`product(b,dim=2)`
is (15,48)

RADIX(x) : Returns the base of the numeric argument. See Chap. 5 and numeric models.

argument: x	**type**: i,r
result: i	**class**: i

Example(s): `base=radix(x)`

RANDOM_INIT (repeatable, image_distinct) Initialize the pseudorandom number generator.

argument: repeatable	**type**: l
argument: image_ distinct	**type**: l
result: n/a	**class**: e

Note(s):
1. `repeatable` shall be a logical scalar. It is an `intent (in)` argument. If it has the value `true`, the seed accessed by the pseudorandom number generator is set to a processor-dependent value that is the same each time `random_init` is called from the same image. If it has the value `false`, the seed is set to a processor-dependent, unpredictably different value on each call.
2. `image_distinct` shall be a logical scalar. It is an `intent (in)` argument. If it has the value `true`, the seed accessed by the pseudorandom number generator is set to a processor-dependent value that is distinct from the value that would be set by a call to `random_init` by another image. If it has the value `false`, the value to which the seed is set does not depend on which image calls `random_init`.
Example(s):

```
call random_init (repeatable=.false.,
  image_distinct=.false.)
```

Initializes the pseudorandom number generator so that the seed is different on each call and that the sequence generated will differ from that of another image:

RANDOM_NUMBER(harvest) : Returns one pseudorandom number or an array of pseudorandom numbers from the uniform distribution over the range $0 <= x < 1$

argument: harvest	**type**: r
result: n/a	**class**: s

Note(s):
`harvest` is `intent(out)`.
Example(s): `call random_number(harvest=x)`
`call random_number(y)`
`x` and `y` contain uniformly distributed random numbers.

RANDOM_SEED(*size,put,get*) : Restarts(seeds) or queries the pseudorandom generator used by `random_number`.

argument: size	**type**: i
result: n/a	**class**: s

All arguments are of integer type.
Note(s):
1. `size` is `intent(out)`. It is set to the number n of integers that the processor uses to hold the value of the seed.
2. `put` is `intent(in)`. It is an array of rank 1 and $size >= n$ It is used by the processor to set the seed value.
3. `get` is `intent(out)`. It is an array of rank 1 and $size >= n$ It is set by the processor to the current value of the seed.
Example(s): `call random_seed`

RANGE(x) : Returns the decimal exponent range of the real argument. See Chap. 5 and the numeric model representing the argument.

argument: x	**type**: n
result: i	**class**: i

Example(s): `i=range(n)`

RANK (a) : Rank of a data object.

argument: a	**type**: n
result: i	**class**: e

Note(s):

1. `a` shall be a data object of any type.
2. Result Characteristics. Default integer scalar.

Example(s): If X is an assumed-rank dummy argument and its associated effective argument is an array of rank 3, RANK(X) has the value 3.

REAL(a,*kind*) : Converts to real from integer, real or complex.

argument: a	**type**: n
result: r	**class**: e

Example(s): `x=real(a)`

REDUCE (array, operation [, mask, identity, ordered]) or
REDUCE (array, operation , dim [, mask, identity, ordered]) : General reduction of array.

argument: array	**type**: ANY
argument: operation	**type**: See notes
argument: mask	**type**: l
argument: identity	**type**: n
argument: ordered	**type**: n
argument: dim	**type**: n
result: ?	**class**: t

Notes(s):

`operation` shall be a pure function with exactly two arguments; each argument shall be a scalar, nonallocatable, nonpointer, nonpolymorphic, nonoptional dummy data object with the same type and type parameters as `array`. If one argument has the `asynchronous, target, value` attribute, the other shall have that attribute. Its result shall be a nonpolymorphic scalar and have the same type and type parameters as `array`. `operation` should implement a mathematically associative operation. It need not be commutative.

`dim` shall be an integer scalar with a value in the range $1 <= dim <= n$, where n is the rank of ARRAY.

`mask` shall be of type logical and shall be conformable with `array`.

`identity` shall be scalar with the same type and type parameters as `array`.

`ordered` shall be a logical scalar.

If `operation` is not computationally associative, `reduce` without `ordered=.true.` with the same argument values might not always produce the same result, as the processor can apply the associative law to the evaluation.

Example(s):

The following examples all use the function `my_mult`, which returns the product of its two integer arguments.

The value of

```
reduce ([1, 2, 3], my_mult)
```

is 6.

```
reduce (c, my_mult, mask= c > 0, identity=1)
```

forms the product of the positive elements of `c`.

If B is the array

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

```
reduce (b, my_mult, dim = 1)
```

is [2, 12, 30] and

```
reduce (b,my_mult, dim = 2)
```

is [15, 48].

REPEAT(string, n_copies) : Concatenate several copies of a string.

argument: string	**type**: s
result: s	**class**: t

Example(s): `new_s=repeat(s,10)`

RESHAPE(source,shape,`pad,order`) : Constructs an array of a specified shape from the elements of a given array.

argument: source	**type**: any
result: as source	**class**: t

Note(s):

1. `source` must be an array. If `pad` is absent or of size zero the size of `source` must be `product(shape)`.
2. `shape` must be a rank 1 array and $0 <= size < 16$
3. `pad` must be an array.

4. `order` must have the same shape as `shape` and its value must be a permutation of (1,2,... ,n) where n is the size of `shape`. If absent it is as if it were present with the value (1,2,...,n).

5. the result is an array of shape `shape`

Example(s):

```
reshape((/1,2,3,4,5,6/),(/2,3/))
```

has the value

$$\begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$$

```
reshape((/1,2,3,4,5,6/) ,(/2,4/) ,(/0,0/) ,(/2,1/) )
```

has the value

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 0 & 0 \end{pmatrix}$$

RRSPACING(x) : Returns the reciprocal of the relative spacing of model numbers near the argument value. See Chap. 5 and the real numeric model.

argument: x	**type**: r
result: as x	**class**: e

Example(s): `z=rrspacing(x)`

SAME_TYPE_AS(a, b) : Query dynamic types for equality. If the dynamic type of `a` or `b` is extensible, the result is true if and only if the dynamic type of `a` is the same as the dynamic type of `b`. If neither `a` nor `b` has extensible dynamic type, the result is processor dependent.

Note(s):

`a` an object of extensible declared type or unlimited polymorphic. If it is a pointer, it shall not have an undefined association status.

`b` an object of extensible declared type or unlimited polymorphic. If it is a pointer, it shall not have an undefined association status.

The dynamic type of a disassociated pointer or unallocated allocatable variable is its declared type. An unlimited polymorphic entity has no declared type.

result: l

type: i

SCALE(x, i) : Returns xb^i where b is the base in the model representation of x. See Chap. 5 and the real numeric model.

argument: x	**type**: r
argument: i	**type**: i
result: as x	**class**: e

Example(s): `z=scale(x,i)`

SCAN(string, set, *back*) : Scans a string for any one of the characters in a set of characters.

argument: string	**type**: s
result: i	**class**: e

Note(s):

1. The default is to scan from the left, and will only be from the right when back is present and has the value true.
2. Zero is returned if the scan fails.

Example(s): `w=scan(string,set)`

SELECTED_CHAR_KIND(name) : Returns the kind value for the character set whose name is given by the character string name or -1 if not supported.

argument: name	**type**: char
result: i	**class**: t

Note(s):

If `name` has the value `default`, then the result has a value equal to that of the kind type parameter of default character.

If `name` has the value `ASCII`, then the result has a value equal to that of the kind type parameter of ASCII character if the processor supports such a kind; otherwise the result has the value 1.

If `name` has the value `ISO_10646`, then the result has a value equal to that of the kind type parameter of the ISO 10646 character kind (corresponding to UCS-4 as specified in ISO/IEC 10646) if the processor supports such a kind; otherwise the result has the value 1.

If `name` is a processor-defined name of some other character kind supported by the processor, then the result has a value equal to that kind type parameter value.

If `name` is not the name of a supported character type, then the result has the value 1. The `name` is interpreted without respect to case or trailing blanks.

SELECTED_INT_KIND(r) : Returns a value of the kind type parameter of an integer data type that represents all integer values n with $-10^r < n < 10^r$

argument: r	**type**: i
result: i	**class**: t

Note(s):

`r` must be scalar.

If a kind type parameter is not available then the value -1 is returned.

Example(s): `i=selected_int_kind(2)`

SELECTED_REAL_KIND(*p,r,radix)* : Returns a value of the kind type parameter of a real data type with decimal precision of at least p digits and a decimal exponent range of at least r.

argument: p and r	**type**: i
result: i	**class**: t

Note(s):

0. at least one argument must be present.

1. `p`, `r` and `radix` must be integer scalars.

2. The result is -1 if the processor supports a real type with radix `radix` and exponent range of at least `r` but not with precision of at least `p`; -2 if the processor supports a real type with radix `radix` and precision of at least `p` but not with exponent range of at least `r`; -3 if the processor supports a real type with radix `radix` but with neither precision of at least `p` nor exponent range of at least `r`; -4 if the processor supports a real type with radix `radix` and either precision of at least `p` or exponent range of at least `r` but not both together; -5 if the processor supports no real type with radix `radix`.

Example(s): `i=selected_real_kind(p,r)`

SET_EXPONENT(x,i) : Returns the model number whose fractional part is the fractional part of the model representation of x and whose exponent part is i.

argument: x	**type**: r
argument: i	**type**: i
result: as x	**class**: e

Example(s): `exp_part=set_exponent(x,i)`

SHAPE(source, *kind*) : Returns the shape of the array argument or scalar.

argument: source	**type**: any
result: i	**class**: i

Note(s):

1. `source` may be array valued or scalar. It must not be a pointer that is disassociated or an allocatable array that is not allocated. It must not be an assumed-size array.

2. the result is an array of rank 1 whose size is equal to the rank of `source`.

Example(s): `s=shape(a(2:5,-1:1))` yields s=(4,3)

SHIFTA(i, shift) : Right shift with fill.

argument: i	**type**: i
argument: shift	**type**: i
result: same as i	**class**: e

Note(s):

1. `shift` shall be nonnegative and less than or equal to `bit_size(i)`

2. If `shift` is zero the result is `i`. Bits shifted out from the right are lost. The model for the interpretation of an integer value as a sequence of bits is in 16.3 of the standard.

Example(s): `shifta (ibset (0, bit_size (0)), 2)`
is equal to `shiftl (7, bit_size (0) 3).`

SHIFTL(i, shift) : Shift left.

argument: i	**type**: i
argument: shift	**type**: i
result: same as i	**class**: e

Note(s):

1. shift shall be nonnegative and less than or equal to `bit_size(i)`

Example(s): `shiftl(4, 1) is 8`

SHIFTR(i, shift) : Shift right.

argument: i	**type**: i
argument: shift	**type**: i
result: same as i	**class**: e
class: e	

Note(s):

1. shift shall be nonnegative and less than or equal to `bit_size(i)`

Example(s): `shiftr(4, 1)` is 2.

SIGN(a, b) : Absolute value of a times the sign of b.

argument: a	**type**: i, r
result: as a	**class**: e

Note(s):
1. If $b > 0$, the value of the result is $|a|$.
2. If $b < 0$, the value of the result is $-|a|$.
3. If b is of type integer and $b = 0$, the value of the result is $|a|$.
4. If b is of type real and is zero, then:

if the processor cannot distinguish between positive and negative real zero, or 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 $-|a|$.
Example(s): `a=sign(a,b)`

SIN(x) : Sine.

argument: x	**type**: r, c
result: as argument	**class**: e

Note(s):
The argument is in radians.
Example(s): `z=sin(x)`

SINH(x) : Hyperbolic sine.

argument: x	**type**: r,c
result: as argument	**class**: e

Example(s): `z=sinh(x)`

SIZE(array, *dim, kind*) : Extent of an array along a specified dimension or the total number of elements in the array.

argument: array	**type**: any
result: i	**class**: i

Note(s):
1. `array` shall be a scalar or array of any type. It shall not be an unallocated allocatable variable or a pointer that is not associated. If `array` is an assumed-size array, `dim` shall be present with a value less than the rank of `array`.
2. `dim` (optional) shall be an integer scalar with a value in the range $1 <= dim <= n$, where n is the rank of `array`.

3. `kind` shall be a scalar integer constant expression.
4. result is equal to the extent of dimension `dim` of array, or if `dim` is absent, the total number of elements of array.

Example(s): `a=size(array)`

SPACING(x) : Returns the absolute spacing of model numbers near the argument value. See Chap. 5 and the real numeric model.

argument: x	**type**: r
result: as x	**class**: e

Example(s): `s=spacing(x)`

SPREAD(source, dim, n_copies) : Creates an array with an additional dimension, replicating the values in the original array.

argument: source	**type**: any
result: as source	**class**: t

Note(s):
1. `source` may be array valued or scalar, with rank less than 15.
2. `dim` must be scalar and in the range $1 <= dim <= n + 1$ where n is the rank of source.
3. `n_copies` must be scalar.
4. the result is an array of rank $n + 1$.

Example(s):
If a is the array(2,3,4) then `spread(a,dim=1,ncopies=3)` then the result is the array

$$\begin{pmatrix} 2 & 3 & 4 \\ 2 & 3 & 4 \\ 2 & 3 & 4 \end{pmatrix}$$

SQRT(x) : Square root.

argument: x	**type**: r, c
result: as argument	**class**: e

Example(s): `a=sqrt(b)`

STORAGE_SIZE(a, *kind*) : Storage size in bits.

argument: a	**type**: any type.
argument: kind(optional)	**result**: i
class: i	

Note(s):

If `a` is polymorphic it shall not be an undefined pointer. If it is unlimited polymorphic or has any deferred type parameters, it shall not be an unallocated allocatable variable or a disassociated or undefined pointer.

If `kind` is present, the kind type parameter is that specified by the value of `kind`; otherwise, the kind type parameter is that of default integer type.

The result value is the size expressed in bits for an element of an array that has the dynamic type and type parameters of `a`. If the type and type parameters are such that storage association applies, the result is consistent with the named constants defined in the intrinsic module `ISO_FORTRAN_ENV`.

An array element might take more bits to store than an isolated scalar, since any hardware-imposed alignment requirements for array elements might not apply to a simple scalar variable.

This is intended to be the size in memory that an object takes when it is stored; this might differ from the size it takes during expression handling (which might be the native register size) or when stored in a file. If an object is never stored in memory but only in a register, this function nonetheless returns the size it would take if it were stored in memory.

Example(s): `storage_size(1.0)` has the same value as the named constant `numeric_storage_size` in the intrinsic module `iso_fortran_env`.

SUM(array, *dim, mask*) or
SUM(array, *mask*) : Returns the sum of all elements of array along the dimension `dim` corresponding to the true elements of `mask`.

argument: array	**type**: n
argument: dim	**type**: i
argument: mask	**type**: l
result: as array	**class**: t

Note(s):

1. `array` must be an array.

2. $1 <= dim <= n$ where n is the rank of `array`.

3. `mask` must be conformable with `array`.

4. result is scalar if `dim` is absent, or `array` has rank 1, otherwise the result is an array of rank $n - 1$.

Example(s):

```
sum((/1,2,3/))
```

the result is 6.

`sum(c,mask=c> 0.0)`

forms the arithmetic sum of the positive elements of c.

If

$$B = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$$

`sum(b,dim=1)`

is (3,7,11)

`sum(b,dim=2)`

is (9,12)

SYSTEM_CLOCK(*count,count_rate,count_max*) : Returns integer data from a real time clock.

argument: count	**type**: i
result: n/a	**class**: s

Note(s):

1. `count` is `intent(out)` and is set to a processor dependent value based on the current value of the processor clock or to `-huge(0)` if there is no clock. It lies in the range 0 to `count_max` if there is a clock.

2. `count_rate` is `intent(out)` and it is set to the number of processor clock counts per second, or zero if there is no clock.

3. `count_max` is `intent(out)` and is set to the maximum value that count can have or to zero if there is no clock.

`call system_clock(c,r,m)`

TAN(x) : Tangent.

argument: x	**type**: r,c
result: as argument	**class**: e

Note(s):

x must be in radians.

Example(s): `y=tan(x)`

TANH(x) : Hyperbolic tangent.

argument: x	**type**: r,c
result: as argument	**class**: e

```
y=tanh(x)
```

THIS_IMAGE(team) or
THIS_IMAGE(coarray [,team]) or
THIS_IMAGE(coarray, dim [,team]) : Index of the invoking image, a single cosubscript, or a list of cosubscripts.

argument: team	**type**: te
argument: coarray	**type**: a
argument: dim	**type**: i
result: as argument	**class**: e

Note(s):
1. `coarray` shall be a coarray of any type. If it is allocatable it shall be allocated. If its designator has more than one part-ref , the rightmost part-ref shall have nonzero corank. If it is of type `team_type` from the intrinsic module `ISO_FORTRAN_ENV`, the `team` argument shall appear.
2. `dim` shall be an integer scalar. Its value shall be in the range $1 <= dim <= n$, where n is the corank of `coarray`.
3. `team` shall be a scalar of type `team_type` from the intrinsic module `ISO_FORTRAN_ENV`, whose value identifies the current or an ancestor team. If `coarray` appears, it shall be established in that team.
Example(s):

```
integer, dimension(10,20), &
         codimension[10,0:9,0:*] :: a
```

then on image 5, `this_image()` has the value 5 and `this_image(a)` has the value [3,1,2].

TINY(x) : Returns the smallest positive number in the model representing numbers of the same type and kind type parameter as the argument.

argument: x	**type**: r
result: as x	**class**: i

Example(s): `t=tiny(x)`

TRAILZ(i) : Number of trailing zero bits. If all of the bits of i are zero, the result value is `bit_size(i)`. Otherwise, the result value is the position of the rightmost 1 bit in `i`.

argument: i	**type**: i
result: i	**class**: e

Example(s):

```
trailz(4)
has the value 2.
```

TRANSFER(source, mold, *size*) : Returns a result with a physical representation identical to that of `source`, but interpreted with the type and type parameters of `mold`.

argument: source	**type**: any
result: as mold	**class**: t

Warning: A thorough understanding of the implementation specific internal representation of the data types involved is necessary for successful use of this function. Consult the documentation that accompanies the compiler that you work with before using this function.

TRANSPOSE(matrix) : Transposes an array of rank 2.

argument: matrix	**type**: any
result: as argument	**class**: t

Note(s):
`matrix` must be of rank 2. If its shape is (n, m) then the resultant matrix has shape (m, n)
Example(s):
`transpose(a)`

$$a = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \ \textit{yields} \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}$$

TRIM(string) : Returns the argument with trailing blanks removed.

argument: string	**type**: s
result: as string	**class**: t

Note(s):
string must be a scalar.

Example(s): `t_s=trim(s)`

UBOUND(array, *dim, kind*) : Upper bound(s).

argument: array	**type**: any
result: i	**class**: i

Note(s):

1. `dim` optional. Shall be an integer scalar with a value in the range $1 <= dim <= n$, where n is the rank of array. The corresponding actual argument shall not be an optional dummy argument.

2. For an array section or for an array expression, other than a whole array, `ubound(array, dim)` has a value equal to the number of elements in the given dimension; otherwise, it has a value equal to the upper bound for subscript `dim` of `array` if dimension `dim` of `array` does not have size zero and has the value zero if dimension `dim` has size zero.

Example(s): `z=ubound(a)`

UCOBOUND(coarray, *dim, kind*) : Upper cobound(s) of a coarray.

argument: coarray	**type**: co
argument: dim (optional)	**type**: i
argument: kind(optional)	**type**: i
result: i	**class**: i

UNPACK(vector, mask, field) : Unpacks an array of rank 1 into an array under the control of a mask.

argument: vector	**type**: any
result: as vector	**class**: t

Note(s):

1. `vector` must have rank 1. Its size must be at least t, where t is the number of true elements in mask.

2. `mask` must be array valued.

3. `field` must be conformable with `mask`. Result is an array with the same shape as `mask`.

Example(s):

With `vector=(1,2,3)`

$$and\ mask = \begin{pmatrix} f & t & f \\ t & f & f \\ f & f & t \end{pmatrix}\ and\ field \begin{pmatrix} 1\,0\,0 \\ 0\,1\,0 \\ 0\,0\,1 \end{pmatrix}\ the\ result\ is \begin{pmatrix} 1\,2\,0 \\ 1\,1\,0 \\ 0\,0\,3 \end{pmatrix}$$

VERIFY(string,set,*back,kind*) : Verify that a set of characters contains all the characters in a string by identifying the position of the first character in a string of characters that does not appear in a given set of characters.

argument: string	**type**: s
argument: set	**type**: s
argument: back	**type**: l
result: kind	**class**: i
result: i	**class**: e

Note(s):

1. The default is to scan from the left, and will only be from the right when back is present and has the value true.

2. The value of the result is zero if each character in `string` is in set, or if `string` has zero length.

Example(s) `i=verify(string,set)`

D.10 Fortran Intrinsics by Standard

We use a + character in the table to indicate that the name of the intrinsic continues on the next line. The intrinsics by standard year are in Table D.5.

Table D.5 Intrinsic functions by standard - Fortran 90 to Fortran 2018

Fortran 90	Fortran 95	Fortran 2003	Fortran 2008	Fortran 2018
ABS	ABS	ABS	ABS	ABS
ACHAR	ACHAR	ACHAR	ACHAR	ACHAR
ACOS	ACOS	ACOS	ACOS	ACOS
			ACOSH	ACOSH
ADJUSTL	ADJUSTL	ADJUSTL	ADJUSTL	ADJUSTL
ADJUSTR	ADJUSTR	ADJUSTR	ADJUSTR	ADJUSTR
AIMAG	AIMAG	AIMAG	AIMAG	AIMAG
AINT	AINT	AINT	AINT	AINT
ALL	ALL	ALL	ALL	ALL
ALLOCATED	ALLOCATED	ALLOCATED	ALLOCATED	ALLOCATED
ANINT	ANINT	ANINT	ANINT	ANINT
ANY	ANY	ANY	ANY	ANY
ASIN	ASIN	ASIN	ASIN	ASIN
			ASINH	ASINH
ASSOCIATED	ASSOCIATED	ASSOCIATED	ASSOCIATED	ASSOCIATED
ATAN	ATAN	ATAN	ATAN	ATAN
ATAN2	ATAN2	ATAN2	ATAN2	ATAN2
			ATANH	ATANH
				ATOMIC_ADD
				ATOMIC_AND
				ATOMIC_CAS
				ATOMIC_DEFINE
				ATOMIC_FETCH+ _ADD
				ATOMIC_FETCH+ _AND
				ATOMIC_FETCH+ _OR
				ATOMIC_FETCH+ _XOR
				ATOMIC_OR
				ATOMIC_REF
				ATOMIC_XOR
			BESSEL_J0	BESSEL_J0
			BESSEL_J1	BESSEL_J1
			BESSEL_JN	BESSEL_JN
			BESSEL_Y0	BESSEL_Y0
			BESSEL_Y1	BESSEL_Y1
			BESSEL_YN	BESSEL_YN
			BGE	BGE
			BGT	BGT
BIT_SIZE	BIT_SIZE	BIT_SIZE	BIT_SIZE	BIT_SIZE
			BLE	BLE
			BLT	BLT
BTEST	BTEST	BTEST	BTEST	BTEST
CEILING	CEILING	CEILING	CEILING	CEILING
CHAR	CHAR	CHAR	CHAR	CHAR

Table D.5 (continued)

Fortran 90	Fortran 95	Fortran 2003	Fortran 2008	Fortran 2018
CMPLX	CMPLX	CMPLX	CMPLX	CMPLX
				CO_BROADCAST
			CO_LBOUND	CO_LBOUND
				CO_MAX
				CO_MIN
				CO_REDUCE
				CO_SUM
			CO_UBOUND	CO_UBOUND
		COMMAND+	COMMAND+	COMMAND+
		_ARGUMENT+	_ARGUMENT+	_ARGUMENT+
		_COUNT	_COUNT	_COUNT
CONJG	CONJG	CONJG	CONJG	CONJG
COS	COS	COS	COS	COS
COSH	COSH	COSH	COSH	COSH
				COSHAPE
COUNT	COUNT	COUNT	COUNT	COUNT
	CPU_TIME	CPU_TIME	CPU_TIME	CPU_TIME
CSHIFT	CSHIFT	CSHIFT	CSHIFT	CSHIFT
DATE_AND_TIME	DATE_AND_TIME	DATE_AND_TIME	DATE_AND_TIME	DATE_AND_TIME
DBLE	DBLE	DBLE	DBLE	DBLE
DIGITS	DIGITS	DIGITS	DIGITS	DIGITS
DIM	DIM	DIM	DIM	DIM
DOT_PRODUCT	DOT_PRODUCT	DOT_PRODUCT	DOT_PRODUCT	DOT_PRODUCT
DPROD	DPROD	DPROD	DPROD	DPROD
			DSHIFTL	DSHIFTL
			DSHIFTR	DSHIFTR
EOSHIFT	EOSHIFT	EOSHIFT	EOSHIFT	EOSHIFT
EPSILON	EPSILON	EPSILON	EPSILON	EPSILON
			ERF	ERF
			ERFC	ERFC
			ERFC_SCALED	ERFC_SCALED
				EVENT_QUERY
			EXECUTE+	EXECUTE+
			_COMMAND+	_COMMAND+
			_LINE	_LINE
EXP	EXP	EXP	EXP	EXP
EXPONENT	EXPONENT	EXPONENT	EXPONENT	EXPONENT
		EXTENDS+	EXTENDS+	EXTENDS+
		_TYPE_OF	_TYPE_OF	_TYPE_OF
				FAILED_IMAGES
				FINDLOC
FLOOR	FLOOR	FLOOR	FLOOR	FLOOR
FRACTION	FRACTION	FRACTION	FRACTION	FRACTION
			GAMMA	GAMMA
		GET_COMMAND	GET_COMMAND	GET_COMMAND
		GET_COMMAND+	GET_COMMAND+	GET_COMMAND+
		_ARGUMENT	_ARGUMENT	_ARGUMENT
		GET+	GET+	GET+
		_ENVIRONMENT+	_ENVIRONMENT+	_ENVIRONMENT+
		_VARIABLE	_VARIABLE	_VARIABLE
				GET_TEAM

Table D.5 (continued)

Fortran 90	Fortran 95	Fortran 2003	Fortran 2008	Fortran 2018
HUGE	HUGE	HUGE	HUGE	HUGE
			HYPOT	HYPOT
IACHAR	IACHAR	IACHAR	IACHAR	IACHAR
IAND	IAND	IAND	IAND	IALL
IBCLR	IBCLR	IBCLR	IBCLR	IAND
IBITS	IBITS	IBITS	IBITS	IANY
IBSET	IBSET	IBSET	IBSET	IBCLR
				IBITS
				IBSET
ICHAR	ICHAR	ICHAR	ICHAR	ICHAR
IEOR	IEOR	IEOR	IEOR	IEOR
			IMAGE_INDEX	IMAGE_INDEX
				IMAGE_STATUS
INDEX	INDEX	INDEX	INDEX	INDEX
INT	INT	INT	INT	INT
IOR	IOR	IOR	IOR	IOR
				IPARITY
			IS_CONTIGUOUS	IS_CONTIGUOUS
		IS_IOSTAT_END	IS_IOSTAT_END	IS_IOSTAT_END
		IS_IOSTAT_EOR	IS_IOSTAT_EOR	IS_IOSTAT_EOR
ISHFT	ISHFT	ISHFT	ISHFT	ISHFT
ISHFTC	ISHFTC	ISHFTC	ISHFTC	ISHFTC
KIND	KIND	KIND	KIND	KIND
LBOUND	LBOUND	LBOUND	LBOUND	LBOUND
				LCOBOUND
			LEADZ	LEADZ
LEN	LEN	LEN	LEN	LEN
LEN_TRIM	LEN_TRIM	LEN_TRIM	LEN_TRIM	LEN_TRIM
LGE	LGE	LGE	LGE	LGE
LGT	LGT	LGT	LGT	LGT
LLE	LLE	LLE	LLE	LLE
LLT	LLT	LLT	LLT	LLT
LOG	LOG	LOG	LOG	LOG
			LOG_GAMMA	LOG_GAMMA
LOG10	LOG10	LOG10	LOG10	LOG10
LOGICAL	LOGICAL	LOGICAL	LOGICAL	LOGICAL
			MASKL	MASKL
			MASKR	MASKR
MATMUL	MATMUL	MATMUL	MATMUL	MATMUL
MAX	MAX	MAX	MAX	MAX
MAXEXPONENT	MAXEXPONENT	MAXEXPONENT	MAXEXPONENT	MAXEXPONENT
MAXLOC	MAXLOC	MAXLOC	MAXLOC	MAXLOC
MAXVAL	MAXVAL	MAXVAL	MAXVAL	MAXVAL
MERGE	MERGE	MERGE	MERGE	MERGE
			MERGE_BITS	MERGE_BITS
MIN	MIN	MIN	MIN	MIN
MINEXPONENT	MINEXPONENT	MINEXPONENT	MINEXPONENT	MINEXPONENT
MINLOC	MINLOC	MINLOC	MINLOC	MINLOC
MINVAL	MINVAL	MINVAL	MINVAL	MINVAL
MOD	MOD	MOD	MOD	MOD
MODULO	MODULO	MODULO	MODULO	MODULO
		MOVE_ALLOC	MOVE_ALLOC	MOVE_ALLOC

Table D.5 (continued)

Fortran 90	Fortran 95	Fortran 2003	Fortran 2008	Fortran 2018
MVBITS	MVBITS	MVBITS	MVBITS	MVBITS
NEAREST	NEAREST	NEAREST	NEAREST	NEAREST
		NEW_LINE	NEW_LINE	NEW_LINE
NINT	NINT	NINT	NINT	NINT
			NORM2	NORM2
NOT	NOT	NOT	NOT	NOT
	NULL	NULL	NULL	NULL
			NUM_IMAGES	NUM_IMAGES
				OUT_OF_RANGE
PACK	PACK	PACK	PACK	PACK
			PARITY	PARITY
			POPCNT	POPCNT
			POPPAR	POPPAR
PRECISION	PRECISION	PRECISION	PRECISION	PRECISION
PRESENT	PRESENT	PRESENT	PRESENT	PRESENT
PRODUCT	PRODUCT	PRODUCT	PRODUCT	PRODUCT
RADIX	RADIX	RADIX	RADIX	RADIX
				RANDOM_INIT
RANDOM+ _NUMBER	RANDOM+ _NUMBER	RANDOM+ _NUMBER	RANDOM+ _NUMBER	RANDOM+ _NUMBER
RANDOM_SEED	RANDOM_SEED	RANDOM_SEED	RANDOM_SEED	RANDOM_SEED
RANGE	RANGE	RANGE	RANGE	RANGE
				RANK
REAL	REAL	REAL	REAL	REAL
				REDUCE
REPEAT	REPEAT	REPEAT	REPEAT	REPEAT
RESHAPE	RESHAPE	RESHAPE	RESHAPE	RESHAPE
RRSPACING	RRSPACING	RRSPACING	RRSPACING	RRSPACING
		SAME_TYPE_AS	SAME_TYPE_AS	SAME_TYPE_AS
SCALE	SCALE	SCALE	SCALE	SCALE
SCAN	SCAN	SCAN	SCAN	SCAN
		SELECTED+ _CHAR+ _KIND	SELECTED+ _CHAR+ _KIND	SELECTED+ _CHAR+ _KIND
SELECTED_INT+ _KIND	SELECTED_INT+ _KIND	SELECTED_INT+ _KIND	SELECTED_INT+ _KIND	SELECTED_INT+ _KIND
SELECTED+ _REAL+ _KIND	SELECTED+ _REAL+ _KIND	SELECTED+ _REAL+ _KIND	SELECTED+ _REAL+ _KIND	SELECTED+ _REAL+ _KIND
SET_EXPONENT	SET_EXPONENT	SET_EXPONENT	SET_EXPONENT	SET_EXPONENT
SHAPE	SHAPE	SHAPE	SHAPE	SHAPE
			SHIFTA	SHIFTA
			SHIFTL	SHIFTL
			SHIFTR	SHIFTR
SIGN	SIGN	SIGN	SIGN	SIGN
SIN	SIN	SIN	SIN	SIN
SINH	SINH	SINH	SINH	SINH
SIZE	SIZE	SIZE	SIZE	SIZE
SPACING	SPACING	SPACING	SPACING	SPACING
SPREAD	SPREAD	SPREAD	SPREAD	SPREAD
SQRT	SQRT	SQRT	SQRT	SQRT
				STOPPED_IMAGES
			STORAGE_SIZE	STORAGE_SIZE

Table D.5 (continued)

Fortran 90	Fortran 95	Fortran 2003	Fortran 2008	Fortran 2018
SUM	SUM	SUM	SUM	SUM
SYSTEM_CLOCK	SYSTEM_CLOCK	SYSTEM_CLOCK	SYSTEM_CLOCK	SYSTEM_CLOCK
TAN	TAN	TAN	TAN	TAN
TANH	TANH	TANH	TANH	TANH
				TEAM_NUMBER
				THIS_IMAGE
TINY	TINY	TINY	TINY	TINY
			TRAILZ	TRAILZ
TRANSFER	TRANSFER	TRANSFER	TRANSFER	TRANSFER
TRANSPOSE	TRANSPOSE	TRANSPOSE	TRANSPOSE	TRANSPOSE
TRIM	TRIM	TRIM	TRIM	TRIM
UBOUND	UBOUND	UBOUND	UBOUND	UBOUND
				UCOBOUND
UNPACK	UNPACK	UNPACK	UNPACK	UNPACK
VERIFY	VERIFY	VERIFY	VERIFY	VERIFY
N = 113	N = 115	N = 126	N = 166	N = 200

D.11 Standard Intrinsic Modules

The standard defines five standard intrinsic modules:

- a Fortran environment module
- a set of three modules to support floating-point exceptions and IEEE arithmetic
- a module to support interoperability with the C programming language

The intrinsic modules

- IEEE_EXCEPTIONS
- IEEE_ARITHMETIC
- IEEE_FEATURES are described in Clause 17 of the standard.

The intrinsic module `ISO_C_BINDING` is described in Clause 18 of the standard.

The intrinsic module `ISO_FORTRAN_ENV` provides public entities relating to the Fortran environment.

The processor shall provide the named constants, derived types, and procedures described in subclause 16.10.2. of the standard.

Here is a complete list of the public entities in this module.

- ATOMIC_INT_KIND
- ATOMIC_LOGICAL_KIND
- CHARACTER_KINDS
- CHARACTER_STORAGE_SIZE
- COMPILER_OPTIONS ( )
- COMPILER_VERSION ( )
- CURRENT_TEAM
- ERROR_UNIT
- EVENT_TYPE
- FILE_STORAGE_SIZE
- INITIAL_TEAM
- INPUT_UNIT
- INT8, INT16, INT32, and INT64
- INTEGER_KINDS
- IOSTAT_END
- IOSTAT_EOR
- IOSTAT_INQUIRE_INTERNAL_UNIT
- LOCK_TYPE
- LOGICAL_KINDS
- NUMERIC_STORAGE_SIZE
- OUTPUT_UNIT
- PARENT_TEAM
- REAL_KINDS
- REAL32, REAL64, and REAL128
- STAT_FAILED_IMAGE

- STAT_LOCKED
- STAT_LOCKED_OTHER_IMAGE
- STAT_STOPPED_IMAGE
- STAT_UNLOCKED
- STAT_UNLOCKED_FAILED_IMAGE
- TEAM_TYPE

Consult the standard for more information.

Appendix E
Text extracts, English, Latin and coded

English and Latin

```
YET IF HE SHOULD GIVE UP WHAT
HE HAS BEGUN, AND AGREE TO MAKE US OR
OUR KINGDOM SUBJECT TO THE KING OF ENGLAND
OR THE ENGLISH, WE SHOULD
EXERT OURSELVES AT ONCE TO DRIVE HIM OUT AS
OUR ENEMY AND A SUBVERTER
OF HIS OWN RIGHTS AND OURS, AND MAKE SOME
OTHER MAN WHO WAS ABLE TO
DEFEND US OUR KING; FOR, AS LONG AS BUT A
HUNDRED OF US REMAIN ALIVE,
NEVER WILL WE ON ANY CONDITIONS BE BROUGHT
UNDER ENGLISH RULE. IT
IS IN TRUTH NOT FOR GLORY, NOR RICHES, NOR
HONOURS THAT WE ARE FIGHTING,
BUT FOR FREEDOM - FOR THAT ALONE, WHICH NO
HONEST MAN GIVES UP BUT
WITH LIFE ITSELF.

QUEM SI AB INCEPTIS
DIESISTERET, REGI ANGLORUM AUT ANGLICIS NOS
AUT
REGNUM NOSTRUM VOLENS SUBICERE, TANQUAM
INIMICUM NOSTRUM ET SUI NOSTRIQUE
JURIS SUBUERSOREM STATIM EXPELLERE NITEREMUR
ET ALIUM REGEM NOSTRUM
QUI AD DEFENSIONEM NOSTRAM SUFFICERET
FACEREMUS. QUIA QUANDIU CENTUM
EX NOBIS VIUI REMANSERINT, NUCQUAM ANGLORUM
```

```
DOMINIO ALIQUATENUS VOLUMUS
SUBIUGARI. NON ENIM PROPTER GLORIAM,
DIUICIAS AUT HONORES PUGNAMUS
SET PROPTER LIBERATEM SOLUMMODO QUAM NEMO
BONUS NISI SIMUL CUM VITA
AMITTIT.

from'The Declaration of Arbroath'
c.1320. The English translation is by
Sir James Fergusson.
```

Coded

```
OH YABY NSFOUN, YAN DUBZY LZ DBUYLTUBFAJ
BYYBOHNX GPDA FNUZNDYOLH
YABY YAN SBF LZ B GOHTMN FULWOHDN DLWNUNX
YAN GFBDN LZ BH NHYOUN DOYJ,
BHX YAN SBF LZ YAN NSFOUN OYGNMZ BH NHYOUN
FULWOHDN. OH YAN DLPUGN
LZ YOSN, YANGN NKYNHGOWN SBFG VNUN ZLPHX
GLSNALV VBHYOHT, BHX GL YAN
DLMMNTN LZ DBUYLTUBFANUG NWLMWNX B SBF LZ
YAN NSFOUN YABY VBG YAN
GBSN GDBMN BG YAN NSFOUN BHX YABY DLOHDOXNX
VOYA OY FLOHY ZLU FLOHY.
MNGG BYYNHYOWN YL YAN GYPXJ LZ DBUYLTUBFAJ,
GPDDNNXOHT TNHNUBYOLHG
DBSN YL RPXTN B SBF LZ GPDA SBTHOYPXN
DPSENUGLSN, BHX, HLY VOYALPY
OUUNWNUNHDN, YANJ BEBHXLHNX OY YL YAN
UOTLPUG LZ GPH BHX UBOH. OH
YAN VNGYNUH XNGNUYG, YBYYNUNX ZUBTSNHYG LZ
YAN SBF BUN GYOMM YL EN
ZLPHX, GANMYNUOHT BH LDDBGOLHBM ENBGY LU
ENTTBU; OH YAN VALMN HBYOLH,
HL LYANU UNMOD OG MNZY LZ YAN XOGDOFMOHN LZ
TNLTUBFAJ.
```

Appendix F
Formal syntax

Statement Ordering
Format statements may appear anywhere between the use statement and the contains statement.

The following table summarises the usage of the various statements within individual scoping units.

Kind of scoping unit	Main program	Module	External sub program	Module sub program	Internal sub program	Interface body
use	Y	Y	Y	Y	Y	Y
format	Y	N	Y	Y	Y	N
misc dec.	Y	Y	Y	Y	Y	Y
derived type definition	Y	Y	Y	Y	Y	Y
interface block	Y	Y	Y	Y	Y	Y
executable statement	Y	N	Y	Y	Y	N
contains	Y	Y	Y	Y	N	N

misc dec. (miscellaneous declaration) are parameter statements, implicit statements, type declaration statements and specification statements.

Syntax Summary of Some Frequently Used Fortran Constructs
The following provides simple syntactical definitions of some of the more frequently used parts of Fortran.

Main Program

```
program [ program-name ]
[ specification-construct ] ...
[ executable-construct ] ...
[contains
[ internal procedure ] ... ]
end [ program [ program-name ] ]
```

Subprogram

procedure heading
[specification-construct] ...
[executable-construct] ...
[contains
[internal procedure] ...]
procedure ending

Module

module name
[specification-construct] ...
[contains
subprogram
[subprogram] ...]
end [module [module-name]

Internal Procedure

procedure heading
[specification construct] ...
[executable construct] ...
procedure ending

Procedure Heading

[recursive] [type specification] function function-name
([dummy argument list]) [result (result name)]
[recursive] subroutine subroutine name
[([dummy argument list])]

Procedure Ending

end [function [function name]]
end [subroutine [subroutine name]]

Specification Construct

derived type definition
interface block
specification statement

Derived Type Definition

type [[, access specification] ::] type name
[private]
[sequence]
[type specification [[, pointer] ::] component specification list]
...
end type [type name]

Interface Block

interface [generic specification]
[procedure heading
[specification construct] ...
procedure ending] ...
[module procedure module procedure name list] ...
end interface

Specification Statement

allocatable [::] allocatable array list
dimension array dimension list
external external name list
format ([format specification list])
implicit implicit specification
intent (intent specification) :: dummy argument name list
intrinsic intrinsic procedure name list
optional [::] optional object list
parameter (named constant definition list)
pointer [::] pointer name list
public [[::] module entity name list]
private[[::] module entity name list]
save[[::] saved object list]
target [::] target name list
use module name [, rename list]
use module name , only : [access list]
type specification [[, attribute specification] ... :: object declaration list

Type Specification

integer [([kind=] kind parameter)]
real[([kind=] kind parameter)]
complex[([kind=] kind parameter)]
character[([kind=] kind parameter)]
character[([kind=] kind parameter)]
[len=] length parameter)
logical[([kind=] kind parameter)]
type (type name)

Attribute Specification

allocatable
dimension (array specification)
external
intent (intent specification)
intrinsic
optional
parameter
pointer
private
public
save
target

Executable Construct

action statement
case construct
do construct
if construct
where construct

Action Statement

allocate (allocation list) [,stat= scalar integer variable])
call subroutinename [([actual argument specification list])]
close (close specification list)
cycle [do construct name]
deallocate(name list) [, stat= scalar integer variable])
endfile external file unit
exit [do construct name]
goto label
if (scalar logical expression) action statement
inquire (inquire specification list) [output item list]
nullify (pointer object list)
open [and close] (connect specification list)
print format [, output item list]
read (i/o control specification list) [input item list]
read format [, output item list]
return [scalar integer expression]
rewind (position specification list)
stop [access code]
where (array logical expression) array assignment expression
write (i/o control specification list) [output item list]
pointer variable => target expression
variable = expression

Appendix G
Compiler Options

In this appendix we look at some of compiler options we have used during the development of the programs in the book.

Simplistically there are two kinds of compile or build.

- A debug build - used when developing code
- A production build - used when executing or running code

We provide debug and production build options for each compiler.

There are also extracts from the help files on what the various options mean.

G.1 Cray

G.1.1 Debug

```
-G Debug level
-R run time checks
```

G.1.2 Production

We used the default compiler options.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran*,
https://doi.org/10.1007/978-3-319-75502-1

G.2 gfortran

G.2.1 Debug

```
gfortran
-fbacktrace
-fcheck=all
-ffpe-trap=zero,overflow,underflow
-g
-O
-pedantic-errors
-std=f2008
-Wall
-Wunderflow
```

G.2.2 Production

```
gfortran
-ffast-math
-funroll-loops
-O3
```

Here are some extracts from the help files.

```
debug

-fbacktrace
  trace back in the event of a run time
  error, i.e. the Fortran runtime library
  tries to output a backtrace of the error

-fcheck
  Enable the generation of run-time checks;
  the argument shall be a comma-delimited
  list of the following keywords.
  all Enable all run-time test of -fcheck

-ffpe-trap=list
  Specify a list of floating point exception
  traps to enable
```

```
-pedantic
  Issue warnings for uses of extensions to
  Fortran 95

-std
  standard conformance

-Wall
  Enables commonly used warning options
  pertaining to usage that we
  recommend avoiding and that we believe
  are easy to avoid.  This
  currently includes -Waliasing,
  -Wampersand, -Wconversion,
  -Wsurprising, -Wc-binding-type,
  -Wintrinsics-std, -Wtabs,
  -Wintrinsic-shadow, -Wline-truncation,
  -Wtarget-lifetime,
  -Wreal-q-constant and -Wunused.

-Wunderflow
  Produce a warning when numerical constant
  expressions are   encountered, which
  yield an UNDERFLOW during compilation.
  Enabled by default

-Wrealloc-lhs
  Warn when the compiler might insert code
  to for allocation or   reallocation of
  an allocatable array variable of intrinsic
  type in intrinsic assignments

production

-fcoarray
  none
    Disable coarray support; using coarray
    declarations and image-
    control statements will produce a
    compile-time error. (Default)
  single
    Single-image mode, i.e. "num_images()"
    is always one.
  lib Library-based coarray parallelization; a
    suitable GNU Fortran
```

```
      coarray library needs to be linked

  -fopenmp
    Enable the OpenMP extensions
```

G.3 Intel

G.3.1 Debug

```
ifort
/check:all
/debug:all
/fpe:0
/gen-interfaces
/standard-semantics
/traceback
/warn:all
```

You will also need

```
/Qcoarray
/Qopenmp
```

when compiling the coarray and openmp examples.

Here is an extract from the help files.

```
/check:all

enables the following

check arg_temp_created

   Enables run-time checking on whether actual arguments are
   copied into temporary storage before routine calls. If a
   copy is made at run-time, an informative message is
   displayed.

check assume

   Enables run-time checking on whether the
   scalar-Boolean-expression in the ASSUME directive is true
   and that the addresses in the ASSUME_ALIGNED directive are
```

aligned on the specified byte boundaries. If the test is .FALSE., a run-time error is reported and the execution terminates.

check bounds

Enables compile-time and run-time checking for array subscript and character substring expressions. An error is reported if the expression is outside the dimension of the array or the length of the string.
For array bounds, each individual dimension is checked. For arrays that are dummy arguments, only the lower bound is checked for a dimension whose upper bound is specified as * or where the upper and lower bounds are both 1.
For some intrinsics that specify a DIM= dimension argument, such as LBOUND, an error is reported if the specified dimension is outside the declared rank of the array being operated upon.
Once the program is debugged, omit this option to reduce executable program size and slightly improve run-time performance.
It is recommended that you do bounds checking on unoptimized code. If you use option check bounds on optimized code, it may produce misleading messages because registers (not memory locations) are used for bounds values.

check contiguous

Tells the compiler to check pointer contiguity at pointer-assignment time. This will help prevent programming errors such as assigning contiguous pointers to non-contiguous objects.

check format

Issues the run-time FORVARMIS fatal error when the data type of an item being formatted for output does not match the format descriptor being used (for example, a REAL*4 item formatted with an I edit descriptor).
With check noformat, the data item is formatted using the specified descriptor unless the length of the item cannot accommodate the descriptor (for example, it is still an error to pass an INTEGER*2 item to an E edit descriptor).

check output_conversion

Issues the run-time OUTCONERR continuable error message when a data item is too large to fit in a designated format descriptor field without loss of significant digits. Format truncation occurs, the field is filled with asterisks (*), and execution continues.

check pointers

Enables run-time checking for disassociated or uninitialized Fortran pointers, unallocated allocatable objects, and integer pointers that are uninitialized.

check stack

Enables checking on the stack frame. The stack is checked for buffer overruns and buffer underruns. This option also enforces local variables initialization and stack pointer verification.
This option disables optimization and overrides any optimization level set by option O.

check uninit

Enables run-time checking for uninitialized variables. If a variable is read before it is written, a run-time error routine will be called. Only local scalar variables of intrinsic type INTEGER, REAL, COMPLEX, and LOGICAL without the SAVE attribute are checked.
To detect uninitialized arrays or array elements, please see option [Q]init or see the article titled: Detection of Uninitialized Floating-point Variables in Intel Fortran, which is located in
https://software.intel.com/articles/detection-of-uninitialized-floating-point-variables-in-intel-fortran

/debug:all

Generates complete debugging information. It produces symbol table information needed for full symbolic debugging of unoptimized code and global symbol information needed for linking. It is the same as specifying /debug with no keyword. If you specify /debug:full for an application that makes calls to C library routines and you need to debug calls into the C library, you should also specify /dbglibs to request that the appropriate C debug library be linked against.

/fpe:0

Floating-point invalid, divide-by-zero, and overflow exceptions are enabled throughout the application when the main program is compiled with this value. If any such exceptions occur, execution is aborted. This option causes denormalized floating-point results to be set to zero.

/gen_interfaces

Tells the compiler to generate an interface block for each routine in a source file.

/standard_semantics

Determines whether the current Fortran Standard behaviour of the compiler is fully implemented.

/traceback

Tells the compiler to generate extra information in the object file to provide source file traceback information when a severe error occurs at run time.

/warn:all

alignments

Warnings are issued about data that is not naturally aligned.

general

All information-level and warning-level messages are enabled.

nodeclarations

No warnings are issued for undeclared names.

noerrors

```
     Warning-level messages are not changed to error-level
     messages.

  noignore_loc

     No warnings are issued when %LOC is stripped from an
     argument.

  nointerfaces

     The compiler does not check interfaces of SUBROUTINEs
     called and FUNCTIONs invoked in your compilation against an
     external set of interface blocks.

  nostderrors

     Warning-level messages about Fortran standards violations
     are not changed to error-level messages.

  notruncated_source

     No warnings are issued when source exceeds the maximum
     column width in fixed-format files.

  nouncalled

     No warnings are issued when a statement function is not
     called.

  nounused

     No warnings are issued for variables that are declared but
     never used.

  usage

     Warnings are issued for questionable programming practices.
```

G.3.2 Production

Intel (autoparallel)

```
ifort
/fast
  enables
  /QxHOST /O3 /Qipo /Qprec-div
/fp:fast=2
/heap-arrays
/Qopenmp
/Qparallel
```

Here are some extracts from the compiler documentation.

```
/QxHost       generate instructions for the
              highest instruction set and
              processor available on the
              compilation host machine
/O3           optimize for maximum speed and
              enable more aggressive
              optimizations that may not
              improve performance on
              some programs
/Qipo         Interprocedural Optimization
              (IPO) enable multi-file IP
              between files
/Qprec-div    improve precision of FP divides
              (some speed impact) /Qprec-div-
              goes for speed over precision
/fp:<name>    enable <name> floating point
              model variation
  except[-]   - enable/disable floating point
                exception semantics
  fast[=1|2]  - enables more aggressive floating
                point optimizations
  precise     - allows value-safe optimizations
  source      - enables intermediates in
                source precision sets

                /assume:protect_parens
                for Fortran
  strict      - enables /fp:precise /fp:except,
                disables contractions and
                enables pragma stdc fenv_access
  consistent  - enables consistent,
reproducible

                results for different
                optimization levels or between
```

```
                  different processors of the
                  same architecture
/heap-arrays  temporary arrays are allocated
              in heap memory rather than on the
              stack.
/Qparallel    enable the auto-parallelizer to
              generate multi-threaded code for
              loops that can be safely executed
              in parallel
```

G.4 Nag

G.4.1 Debug

```
nagfor
-C=all
-C=undefined
-f2008
-g
-gline
-ieee=stop
-info
-mtrace=verbose
-thread_safe
```

Here are extracts from the compiler documentation.

```
-C=check

  Compile checking code according to
  the value of check, which must be one of:

  all       (perform all checks except for
            -C=undefined),
  array     (check array bounds),
  bits      (check bit intrinsic arguments),
  calls     (check procedure references),
  dangling  (check for dangling pointers),
  do        (check DO loops for zero step values),
  intovf    (check for integer overflow),
  none      (do no checking: this is the default),
  present   (check OPTIONAL references),
  pointer   (check POINTER references),
  recursion (check for invalid recursion) or
  undefined (check for undefined variables).

-f2008

  Specify that the base language is Fortran 2008.
  This is the default.

-g

  Produce information for interactive debugging
  by the host system debugger.

-gline

  Compile code to produce a traceback when a

  runtime error message is generated.

-ieee=mode

  Set the mode of IEEE arithmetic operation
  according to mode, which must be one of
  full, nonstd or stop.
  full

    enables all IEEE arithmetic
    facilities including
    non-stop arithmetic.

  nonstd

    Disables non-stop arithmetic, terminating
    execution on floating overflow, division
    by zero or invalid operand. If the
    hardware supports it, this also disables
    IEEE gradual underflow, producing
    zero instead of a denormalised number;
    this can improve performance on some systems.

  stop

    enables all IEEE arithmetic facilities
    except for non-stop arithmetic;
    execution will be terminated on

    floating overflow, division by zero
    or invalid
operand.

The -ieee option must be specified when

compiling the main program unit, and its
effect is global.

The default mode is -ieee=stop. For more

details see the
IEEE 754 Arithmetic Support section.

-info

  Request output of information messages. The

  default is to suppress these messages.

-mtrace=trace_opt_list

  Trace memory allocation and deallocation
  according to the value of trace_opt_list,
  which must be a comma separated
  list of one or more of:

address  (display addresses),
all      (all options except for off),
line     (display file/line info if known),
off      (disable tracing output),
on       (enable tracing output),
paranoia (protect memory allocator data structures
         against the user program),
size     (display size in bytes) or
verbose  (all options except for off and paranoia ).

-thread_safe

  Compile code for safe execution in a
  multi-threaded environment.
```

```
    This must be specified when compiling
    and also during the link phase.
      It is incompatible with the -gc and -gline
    options.
```

G.4.2 Production

```
nagfor
-O4
-openmp
-thread_safe
```

Here are some extracts from the compiler documentation.

```
-ON

    Set the optimisation level to N. The
  optimisation
    levels are:

  -O0

      No optimisation. This is the default, and
      is recommended when debugging.

    -O1

      Minimal quick optimisation.

    -O2

      Normal optimisation.

    -O3

      Further optimisation.

    -O4

      Maximal optimisation.
```

G.4.3 Nag Polish

The Nag compiler has a *polish* option. Here are some of the options used in the reformatting of the examples in the book. The examples in the book were set with a line length of 48 to fit the printed page. The examples on the web site were set with a line length of 132.

```
nagfor =polish -alter_comments
-noblank_cmt_to_blank_line
-blank_line_after_decls -break_long_comment_word
-format_start=100  -format_step=10  -idcase=L
-indent=2 -indent_continuation=2  -indent_max=16
-keep_blank_lines -keep_comments  -kwcase=L
-leave_formats_in_place -margin=0
-noindent_comment_marker
-noseparate_format_numbering -relational=F90+
-renumber -renumber_start=100 -renumber_step=10
-separate_format_numbering
-terminate_do_with_enddo  -width=48
```

G.5 Oracle

G.5.1 Debug

```
sunf90
-ansi
-w4
-xcheck=all
-C
-ftrap=common,overflow,underflow
```

G.5.2 Production

```
sunf90 -fast ch2502.f90 -V -v
```

maps into

```
-xO5
-xtarget=native
```

```
-xchip=pentium
-xcache=generic
-xarch=sse3
-xdepend=yes
-aligncommon=dalign
-fma=fused
-fsimple=2
-fns=yes
-ftrap=division,invalid,overflow
-xlibmil
-xlibmopt
-nofstore
-xregs=frameptr
-y-fsimple=2
-a dalign
-m3
-ev
-xall
-xivdep=loop
-H
```

Here are some extracts from the help files.

```
-C
  Enable runtime subscript range checking
-O
  Use default optimization level (-xO3)
-O<n>
  Same as -xO<n>
-aligncommon[=<a>]
  Align common block elements to the
  specified boundary requirement;
  <a>={1|2|4|8|16|dalign}
-ansi
  Report non-ANSI extensions
-autopar
  Enable automatic loop parallelization
-dalign
  Expands to -aligncommon=dalign
-fma=<a>
  Enable floating-point multiply-add
  instruction; <a>={none|fused}
-fns[={yes|no}]
  Select non-standard floating point mode
-fopenmp
```

```
  Equivalent to -xopenmp=parallel
-fprecision=<a>
  Set FP rounding precision mode;
  <a>={single|double|extended}
-fstore
  Force floating pt. values to target
  precision on assignment
-ftrap=<t>
  Select floating-point trapping mode in
  effect at startup
-g
  Compile for debugging
-xO<n>
  Generate optimized code; <n>={1|2|3|4|5}
-xarch=<a>
  Specify target architecture instruction set
-xcache=<t>
  Define cache properties for use by optimizer
-xchip=<a>
  Specify the target processor for use by the
  optimizer
-xdepend[={yes|no}]
  Analyze loops for data dependencies
-xivdep[=<a>]
  Ignore loop-carried dependences on array
  references in a loop;
  <a>={loop|loop_any|back|back_any|none}
-xlibmil
  Inline selected libm math routines for
  optimization
-xlibmopt
  Link with optimized math library
-xregs=<a>[,<a>]
  Specify the usage of optional registers;
  <a>={frameptr}
-xtarget=<a>
  Specify target system for optimization
```

Index