Fortran 2003 extensions and the NAG Fortran Compiler

March 15, 2019

1 Introduction

This document describes those parts of the Fortran 2003 language which are not in Fortran 95, and indicates which features are currently supported by the NAG Fortran Compiler.

The compiler release in which a feature was made available is indicated by square brackets; for example, a feature marked as '[5.3]' was first available in Release 5.3. Features marked '[n/a]' are not yet available.

2 Overview of Fortran 2003

Fortran 2003 is a major advance over Fortran 95: the new language features can be grouped as follows:

- object-oriented programming features,
- allocatable attribute extensions,
- other data-oriented enhancements,
- interoperability with C,
- IEEE arithmetic support,
- input/output enhancements and
- miscellaneous enhancements.

The basic object-oriented features are type extension, polymorphic variables, and type selection; these provide inheritance and the ability to program ad-hoc polymorphism in a type-safe manner. The advanced features are typed allocation, cloning, type-bound procedures, type-bound generics, and object-bound procedures. Type-bound procedures provide the mechanism for dynamic dispatch (methods).

The `ALLOCATABLE` attribute is extended to allow it to be used for dummy arguments, function results, structure components, and scalars (not just arrays). An intrinsic procedure has been added to transfer an allocation from one variable to another. Finally, in intrinsic assignment, allocatable variables or components are automatically reallocated with the correct size if they have a different shape or type parameter value from that of the expression. This last feature, together with deferred character length, provides the user with true varying-length character variables.

There are two other major data enhancements: the addition of type parameters to derived types, and finalisation (by final subroutines). Other significant data enhancements are the `PROTECTED` attribute, pointer bounds specification and rank remapping, procedure pointers, and individual accessibility control for structure components.

Interoperability with the C programming language consists of allowing C procedures to be called from Fortran, Fortran procedures to be called from C, and for the sharing of global variables between C and Fortran. This can only happen where C and Fortran facilities are equivalent: an intrinsic module provides derived types and named constants for mapping Fortran and C types, and the `BIND(C)` syntax is added for declaring Fortran entities that are to be shared with C. Additionally, C style enumerations have been added.

Support for IEEE arithmetic is provided by three intrinsic modules. Use of the `IEEE_FEATURES` module requests IEEE compliance for specific Fortran features, the `IEEE_EXCEPTIONS` module provides access to IEEE modes and exception handling, and the `IEEE_ARITHMETIC` module provides enquiry functions and utility functions for determining the extent of IEEE conformance and access to IEEE-conformant facilities.

The input/output facilities have had three major new features: asynchronous input/output, stream input/output, and user-defined procedures for derived-type input/output (referred to as "defined input/output"). Additionally, the input/output specifiers have been regularised so that where they make sense: all specifiers that can be used on an `OPEN` statement can also be used on a `READ` or `WRITE` statement, and vice versa. Access to input/output error messages

is provided by the new `IOMSG=` specifier, and processor-dependent constants for input/output (e.g. the unit number for the standard input file) are provided in a new intrinsic module.

Finally, there are a large number of miscellaneous improvements in almost every aspect of the language. Some of the more significant of these are the `IMPORT` statement (provides host association into interface blocks), the `VALUE` and `VOLATILE` attributes, the ability to use all intrinsic functions in constant expressions, and extensions to the syntax of array and structure constructors.

3 Object-oriented programming

3.1 Type Extension

Type extension provides the first phase of object orientation: inheritance and polymorphic objects.

3.1.1 Extending Types [5.0]

Any derived type can be extended using the EXTENDS keyword, except for SEQUENCE types and BIND(C) types. (The latter types are "non-extensible", as are intrinsic types, whereas all other derived types are "extensible".) The extended type inherits all the components of the parent type and may add extra components.

For example:

```
TYPE point
  REAL x,y
END TYPE
TYPE,EXTENDS(point) :: point_3d
  REAL z
END TYPE
```

The type `point_3d` has `x`, `y` and `z` components. Additionally, it has a `point` component which refers to the inherited part; this "parent component" is "inheritance-associated" with the inherited components, so that the `point%x` component is identical to the `x` component et cetera.

However, when extending a type it is not required to add any new components; for example,

```
TYPE,EXTENDS(point) :: newpoint
END TYPE
```

defines a new type `newpoint` which has exactly the same components as point (plus the associated parent component). Similarly, it is no longer necessary for a type to contain any components:

```
TYPE empty_type
END TYPE
```

declares the extensible (but not extended) type `empty_type` which has no components at all.

3.1.2 Polymorphic Variables [5.0]

A polymorphic variable is a pointer, allocatable array or dummy argument that is declared using the CLASS keyword instead of the TYPE keyword. A `CLASS(`*typename*`)` variable can assume any type in the class of types consisting of `TYPE(`*typename*`)` and all extensions of *typename*.

For example:

```
REAL FUNCTION bearing(a)
```

```
  CLASS(point) a
  bearing = atan2(a%y,a%x)
END
```

The function `bearing` may be applied to a `TYPE(point)` object or to a `TYPE(point_3d)` object, or indeed to an object of any type that is an extension of `TYPE(point)`.

3.1.3 Type Selection [5.0]

The `SELECT TYPE` construct provides both a means of testing the dynamic type of a polymorphic variable and access to the extended components of that variable.

For example:

```
CLASS(t) x
...
SELECT TYPE(p=>x)
TYPE IS (t1)
  !
  ! This section is executed only if X is exactly of TYPE(t1), not an
  ! extension thereof.  P is TYPE(t1).
  !
TYPE IS (t2)
  !
  ! This section is executed only if X is exactly of TYPE(t2), not an
  ! extension thereof.  P is TYPE(t2).
  !
CLASS IS (t3)
  !
  ! This section is executed if X is of TYPE(t3), or of some extension
  ! thereof, and if it is not caught by a more specific case.  P is CLASS(t3).
  !
END SELECT
```

Note that '`SELECT TYPE(x)`' is short for '`SELECT TYPE(x=>x)`'.

3.1.4 Unlimited polymorphism [5.2]

A variable that is '`CLASS(*)`' is an unlimited polymorphic variable. It has no type, but can assume any type including non-extensible types and intrinsic types (and kinds). Apart from allocation, deallocation and pointer assignment, to perform any operation on an unlimited polymorphic you first have to discover its type using `SELECT TYPE`. For example:

```
CLASS(*),POINTER :: x
CHARACTER(17),TARGET :: ch
x => ch
SELECT TYPE(x)
TYPE IS (COMPLEX(KIND=KIND(0d0)))
  PRINT *,x+1
TYPE IS (CHARACTER(LEN=*))
  PRINT *,LEN(x)
END SELECT
```

Note that in the case of `CHARACTER` the length must be specified as '`*`' and is automatically assumed from whatever the polymorphic is associated with.

In the case of a non-extensible (i.e. `BIND(C)` or `SEQUENCE`) type, `SELECT TYPE` cannot be used to discover the type; instead, an unsafe pointer assignment is allowed, for example:

```
TYPE t
  SEQUENCE
  REAL x
END TYPE
CLASS(*),POINTER :: x
TYPE(t),POINTER :: y
...
y => x ! Unsafe - the compiler cannot tell whether X is TYPE(t).
```

3.1.5 Ad hoc type comparison [5.3]

Two new intrinsic functions are provided for comparing the dynamic types of polymorphic objects. These are

```
EXTENDS_TYPE_OF(A,MOLD)
SAME_TYPE_AS(A,B)
```

The arguments must be objects of extensible types (though they need not be polymorphic). `SAME_TYPE_AS` returns `.TRUE.` if and only if both `A` and `B` have the same dynamic type. `EXTENDS_TYPE_OF` returns `.TRUE.` if and only if the dynamic type of `A` is the same as, or an extension of, the dynamic type of `MOLD`. Note that if `MOLD` is an unallocated unlimited polymorphic (`CLASS(*)`), the result will be true regardless of the state of `A`.

The arguments are permitted to be unallocated or disassociated, but they are not permitted to be pointers with an undefined association status.

It is recommended that where possible these intrinsic functions be avoided, and that `SELECT TYPE` be used for type checking instead.

3.2 Typed allocation [5.1]

The `ALLOCATE` statement now accepts a *type-spec*; this can be used to specify the dynamic type (and type parameters, if any) of an allocation. The *type-spec* appears before the allocation list, and is separated from it by a double colon.

For example, if `T` is an extensible type and `ET` is an extension of `T`,

```
CLASS(t),POINTER :: a(:)
ALLOCATE(et::a(100))
```

allocates `A` to have dynamic type `ET`. Note that the *type-spec* in an `ALLOCATE` statement omits the `TYPE` keyword for derived types, similarly to the `TYPE IS` and `CLASS IS` statements.

An unlimited polymorphic object can be allocated to be any type including intrinsic types: for example

```
CLASS(*),POINTER :: c,d
ALLOCATE(DOUBLE PRECISION::c)
READ *,n
ALLOCATE(CHARACTER(LEN=n)::d)
```

allocates `C` to be double precision real, and `D` to be of type `CHARACTER` with length `N`.

Typed allocation is only useful for allocating polymorphic variables and `CHARACTER` variables with deferred length (`LEN=:`). For a non-polymorphic variable, the *type-spec* must specify the declared type and, if it is type `CHARACTER` but not deferred-length, to have the same character length. The character length must not be specifed as an asterisk (`CHARACTER(LEN=*)`) unless the allocate-object is a dummy argument with an asterisk character length (and vice versa).

Finally, since there is only one *type-spec* it must be compatible with all the items in the allocation list.

3.3 Sourced allocation (cloning) [5.1]

The `ALLOCATE` statement now accepts the `SOURCE=` specifier. The dynamic type and value of the allocated entity is taken from the expression in the specifier. If the derived type has type parameters (q.v.), the value for any deferred type parameter is taken from the source expression, and the values for other type parameters must agree. This is not just applicable to derived types: if the entity being allocated is type `CHARACTER` with deferred length (`LEN=:`), the character length is taken from the source expression.

Only one entity can be allocated when the `SOURCE=` specifier is used. Note that when allocating an array the array shape is not taken from the source expression but must be specified in the usual way. If the source expression is an array, it must have the same shape as the array being allocated.

For example,

```
CLASS(*),POINTER :: a,b
...
ALLOCATE(a,SOURCE=b)
```

The allocated variable `A` will be a "clone" of `B`, whatever the current type of `B` happens to be.

3.4 Type-bound procedures [5.1]

Type-bound procedures provide a means of packaging operations on a type with the type itself, and also for dynamic dispatch to a procedure depending on the dynamic type of a polymorphic variable.

3.4.1 The type-bound procedure part

The type-bound procedure part of a type definition is separated from the components by the `CONTAINS` statement. The default accessibility of type-bound procedures is public even if the components are private; this may be changed by using the `PRIVATE` statement after the `CONTAINS`.

3.4.2 Specific type-bound procedures

The syntax of a specific, non-deferred, type-bound procedure declaration is:

`PROCEDURE` [[,*binding-attr-list*]::] *binding-name* [`=>`*procedure-name*]

The name of the type-bound procedure is *binding-name*, and the name of the actual procedure which implements it is *procedure-name*. If the optional `=>`*procedure-name* is omitted, the actual procedure has the same name as the binding.

A type-bound procedure is invoked via an object of the type, e.g.

```
CALL variable(i)%tbp(arguments)
```

Normally, the invoking variable is passed as an extra argument, the "passed-object dummy argument"; by default this is the first dummy argument of the actual procedure and so the first argument in the argument list becomes the second argument, etc. The passed-object dummy argument may be changed by declaring the type-bound procedure with the `PASS(`*argument-name*`)` attribute, in which case the variable is passed as the named argument. The `PASS` attribute may also be used to confirm the default (as the first argument), and the `NOPASS` attribute prevents passing the object as an argument at all. The passed-object dummy argument must be a polymorphic scalar variable of that type, e.g. `CLASS(t) self`.

When a type is extended, the new type either inherits or **overrides** each type-bound procedure of the old type. An overriding procedure must be compatible with the old procedure; in particular, each dummy argument must have the same type except for the passed-object dummy argument which must have the new type. A type-bound procedure that is declared to be `NON_OVERRIDABLE` cannot be overridden during type extension.

When a type-bound procedure is invoked, it is the dynamic type of the variable which determines which actual procedure to call.

The other attributes that a type-bound procedure may have are `PUBLIC`, `PRIVATE`, and `DEFERRED` (the latter only for abstract types, which are described later).

3.4.3 Generic type-bound procedures

A generic type-bound procedure is a set of specific type-bound procedures, in the same way that an ordinary generic procedure is a set of specific ordinary procedures. It is declared with the `GENERIC` statement, e.g.

```
GENERIC :: generic_name => specific_name_1, specific_name_2, specific_name_3
```

Generic type-bound procedures may also be operators or assignment, e.g.

```
GENERIC :: OPERATOR(+) => add_t_t, add_t_r, add_r_t
```

Such type-bound generic operators cannot have the `NOPASS` attribute; the dynamic type of the passed-object dummy argument determines which actual procedure is called.

When a type is extended, the new type inherits all the generic type-bound procedures without exception, and the new type may extend the generic with additional specific procedures. To override procedures in the generic, simply override the specific type-bound procedure. For example, in

```
TYPE mycomplex
  ...
CONTAINS
  PROCEDURE :: myc_plus_r => myc1_plus_r
  PROCEDURE,PASS(B) :: r_plus_myc => r_plus_myc1
  GENERIC :: OPERATOR(+) => myc_plus_r, r_plus_myc
END TYPE
...
TYPE,EXTENDS(mycomplex) :: mycomplex_2
  ...
CONTAINS
  PROCEDURE :: myc_plus_r => myc2_plus_r
  PROCEDURE,PASS(B) :: r_plus_myc => r_plus_myc2
END TYPE
```

the type `mycomplex_2` inherits the generic operator '`+`'; invoking the generic (`+`) invokes the specific type-bound procedure, which for entities of type `mycomplex_2` will invoke the overriding actual procedure (`myc2_plus_r` or `r_plus_myc2`).

3.5 Abstract derived types [5.1]

An extensible derived type can be declared to be `ABSTRACT`, e.g.

```
TYPE, ABSTRACT :: mytype
```

An abstract type cannot be instantiated; i.e. it is not allowed to declare a non-polymorphic variable of abstract type, and a polymorphic variable of abstract type must be allocated to be a non-abstract extension of the type.

Abstract type may contain `DEFERRED` type-bound procedures, e.g.

```
...
CONTAINS
  PROCEDURE(interface_name),DEFERRED :: tbpname
```

No binding ("`=> name`") is allowed or implied by a deferred procedure binding. The `interface_name` must be the name of an abstract interface or a procedure with an explicit interface, and defines the interface of the deferred type-bound procedure.

When extending an abstract type, the extended type must also be abstract unless it overrides all of the deferred type-bound procedures with normal bindings.

3.6 Object-bound procedures [5.2]

These are procedure pointer components, and act similarly to type-bound procedures except that the binding is per-object not per-type. The syntax of a procedure pointer component declaration is:

`PROCEDURE(` [*proc-interface*] `)` `,` *proc-component-attr-spec-list* :: *proc-decl-list*

where

- each *proc-component-attr-spec* is one of `NOPASS`, `PASS`, `PASS(`*arg-name*`)`, `POINTER`, `PRIVATE` or `PUBLIC`, and
- each *proc-decl* is a component name optionally followed by default initialisation to null ('`=> NULL()`').

The `POINTER` attribute is required.

Note that object-bound procedures have a passed-object dummy argument just like type-bound procedures; if this is not wanted, the `NOPASS` attribute must be used (and this is required if the interface is implicit, i.e. when `proc-interface` is missing or is a type specification).

The following example demonstrates using a list of subroutines with no arguments.

```
TYPE action_list
  PROCEDURE(),NOPASS,POINTER :: action => NULL()
  TYPE(action_list),POINTER :: next
END TYPE
TYPE(t),TARGET :: top
TYPE(t),POINTER :: p
EXTERNAL sub1,sub2
top%action = sub1
ALLOCATE(top%next)
top%next%action = sub2
...
p => top
DO WHILE (ASSOCIATED(p))
  IF (ASSOCIATED(p%action)) CALL p%action
  p => p%next
END DO
```

4 ALLOCATABLE extensions

In Fortran 2003 the `ALLOCATABLE` attribute is permitted not just on local variables but also on components, dummy variables, and function results. These are the same as described in the ISO Technical Report ISO/IEC TR 15581:1999.

Also, the `MOVE_ALLOC` intrinsic subroutine has been added, as well as automatic reallocation on assignment.

4.1 Allocatable Dummy Arrays [4.x]

A dummy argument can be declared to be an allocatable array, e.g.

```
SUBROUTINE s(dum)
  REAL,ALLOCATABLE :: dum(:,:)
  ...
END SUBROUTINE
```

Having an allocatable dummy argument means that there must be an explicit interface for any reference: i.e. if the procedure is not an internal or module procedure there must be an accessible interface block in any routine which references that procedure.

Any actual argument that is passed to an allocatable dummy array must itself be an allocatable array; it must also have the same type, kind type parameters, and rank. For example:

```
REAL,ALLOCATABLE :: x(:,:)
CALL s(x)
```

The actual argument need not be allocated before calling the procedure, which may itself allocate or deallocate the argument. For example:

```
PROGRAM example2
  REAL,ALLOCATABLE :: x(:,:)
  OPEN(88,FILE='myfile',FORM='unformatted')
  CALL read_matrix(x,88)
  !
  ... process x in some way
  !
  REWIND(88)
  CALL write_and_delete_matrix(x,88)
END
!
MODULE module
CONTAINS
  !
  ! This procedure reads the size and contents of an array from an
  ! unformatted unit.
  !
  SUBROUTINE read_matrix(variable,unit)
    REAL,ALLOCATABLE,INTENT(OUT) :: variable(:,:)
    INTEGER,INTENT(IN) :: unit
    INTEGER dim1,dim2
    READ(unit) dim1,dim2
    ALLOCATE(variable(dim1,dim2))
    READ(unit) variable
    CLOSE(unit)
  END SUBROUTINE
  !
  ! This procedures writes the size and contents of an array to an
  ! unformatted unit, and then deallocates the array.
  !
  SUBROUTINE write_and_delete_matrix(variable,unit)
    REAL,ALLOCATABLE,INTENT(INOUT) :: variable(:,:)
    INTEGER,INTENT(IN) :: unit
    WRITE(unit) SIZE(variable,1),SIZE(variable,2)
    WRITE(unit) variable
    DEALLOCATE(variable)
  END SUBROUTINE
END
```

4.2 Allocatable Function Results [4.x]

The result of a function can be declared to be an allocatable array, e.g.

```
FUNCTION af() RESULT(res)
  REAL,ALLOCATABLE :: res
```

On invoking the function, the result variable will be unallocated. It must be allocated before returning from the function. For example:

```
!
! The result of this function is the original argument with adjacent
! duplicate entries deleted (so if it was sorted, each element is unique).
!
FUNCTION compress(array)
  INTEGER,ALLOCATABLE :: compress(:)
  INTEGER,INTENT(IN) :: array(:)
  IF (SIZE(array,1)==0) THEN
    ALLOCATE(compress(0))
  ELSE
    N = 1
    DO I=2,SIZE(array,1)
      IF (array(I)/=array(I-1)) N = N + 1
    END DO
    ALLOCATE(compress(N))
    N = 1
    compress(1) = array(1)
    DO I=2,SIZE(array,1)
      IF (array(I)/=compress(N)) THEN
        N = N + 1
        compress(N) = array(I)
      END IF
    END DO
  END IF
END
```

The result of an allocatable array is automatically deallocated after it has been used.

4.3 Allocatable Structure Components [4.x]

A structure component can be declared to be allocatable, e.g.

```
MODULE matrix_example
  TYPE MATRIX
    REAL,ALLOCATABLE :: value(:,:)
  END TYPE
END MODULE
```

An allocatable array component is initially not allocated, just like allocatable array variables. On exit from a procedure containing variables with allocatable components, all the allocatable components are automatically deallocated. This is in contradistinction to pointer components, which are not automatically deallocated. For example:

```
SUBROUTINE sub(n,m)
  USE matrix_example
  TYPE(matrix) a,b,c
  !
  ! a%value, b%value and c%value are all unallocated at this point.
```

```
  !
  ALLOCATE(a%value(n,m),b%value(n,m))
  !
  ... do some computations, then
  !
  RETURN
  !
  ! Returning from the procedure automatically deallocates a%value, b%value,
  ! and c%value (if they are allocated).
  !
END
```

Deallocating a variable that has an allocatable array component deallocates the component first; this happens recursively so that all ALLOCATABLE subobjects are deallocated with no memory leaks.

Any allocated allocatable components of a function result are automatically deallocated after the result has been used.

```
PROGRAM deallocation_example
  TYPE inner
    REAL,ALLOCATABLE :: ival(:)
  END TYPE
  TYPE outer
    TYPE(inner),ALLOCATABLE :: ovalue(:)
  END TYPE
  TYPE(outer) x
  !
  ! At this point, x%ovalue is unallocated
  !
  ALLOCATE(x%ovalue(10))
  !
  ! At this point, x%ovalue(i)%ival are unallocated, i=1,10
  !
  ALLOCATE(x%ovalue(2)%ival(1000),x%ovalue(5)%ival(9999))
  !
  ! Only x%ovalue(2)%ival and x%ovalue(5)%ival are allocated
  !
  DEALLOCATE(x%ovalue)
  !
  ! This has automatically deallocated x%ovalue(2)%ival and x%ovalue(5)%ival
  !
END
```

In a structure constructor for such a type, the expression corresponding to an allocatable array component can be

- the NULL() intrinsic, indicating an unallocated array,
- an allocatable array which may be allocated or unallocated, or
- any other array expression, indicating an allocated array.

```
SUBROUTINE constructor_example
  USE matrix_example
  TYPE(matrix) a,b,c
  REAL :: array(10,10) = 1
  REAL,ALLOCATABLE :: alloc_array(:,:)
  a = matrix(NULL())
  !
  ! At this point, a%value is unallocated
  !
```

```
    b = matrix(array*2)
    !
    ! Now, b%value is a (10,10) array with each element equal to 2.
    !
    c = matrix(alloc_array)
    !
    ! Now, c%value is unallocated (because alloc_array was unallocated).
    !
  END
```

Intrinsic assignment of such types does a "deep copy" of the allocatable array components; it is as if the allocatable array component were deallocated (if necessary), then if the component in the expression was allocated, the variable's component is allocated to the right size and the value copied.

```
  SUBROUTINE assignment_example
    USE matrix_example
    TYPE(matrix) a,b
    !
    ! First we establish a value for a
    !
    ALLOCATE(a%value(10,20))
    a%value(3,:) = 30
    !
    ! And a value for b
    !
    ALLOCATE(b%value(1,1))
    b%value = 0
    !
    ! Now the assignment
    !
    b = a
    !
    ! The old contents of b%value have been deallocated, and b%value now has
    ! the same size and contents as a%value.
    !
  END
```

4.4 Allocatable Component Example

This example shows the definition and use of a simple module that provides polynomial arithmetic. To do this it makes use of intrinsic assignment for allocatable components, the automatically provided structure constructors and defines the addition (+) operator. A more complete version of this module would provide other operators such as multiplication.

```
!
! Module providing a single-precision polynomial arithmetic facility
!
MODULE real_poly_module
  !
  ! Define the polynomial type with its constructor.
  ! We will use the convention of storing the coefficients in the normal
  ! order of highest degree first, thus in an N-degree polynomial, COEFF(1)
  ! is the coefficient of X**N, COEFF(N) is the coefficient of X**1, and
  ! COEFF(N+1) is the scalar.
  !
  TYPE,PUBLIC :: real_poly
    REAL,ALLOCATABLE :: coeff(:)
```

```
  END TYPE
  !
  PUBLIC OPERATOR(+)
  INTERFACE OPERATOR(+)
    MODULE PROCEDURE rp_add_rp,rp_add_r,r_add_rp
  END INTERFACE
  !
CONTAINS
  TYPE(real_poly) FUNCTION rp_add_r(poly,real)
    TYPE(real_poly),INTENT(IN) :: poly
    REAL,INTENT(IN) :: real
    INTEGER isize
    IF (.NOT.ALLOCATED(poly%coeff)) STOP 'Undefined polynomial value in +'
    isize = SIZE(poly%coeff,1)
    rp_add_r%coeff(isize) = poly%coeff(isize) + real
  END FUNCTION
  TYPE(real_poly) FUNCTION r_add_rp(real,poly)
    TYPE(real_poly),INTENT(IN) :: poly
    REAL,INTENT(IN) :: real
    r_add_rp = rp_add_r(poly,real)
  END FUNCTION
  TYPE(real_poly) FUNCTION rp_add_rp(poly1,poly2)
    TYPE(real_poly),INTENT(IN) :: poly1,poly2
    INTEGER I,N,N1,N2
    IF (.NOT.ALLOCATED(poly1%coeff).OR..NOT.ALLOCATED(poly2%coeff)) &
      STOP 'Undefined polynomial value in +'
    ! Set N1 and N2 to the degrees of the input polynomials
    N1 = SIZE(poly1%coeff) - 1
    N2 = SIZE(poly2%coeff) - 1
    ! The result polynomial is of degree N
    N = MAX(N1,N2)
    ALLOCATE(rp_add_rp%coeff(N+1))
    DO I=0,MIN(N1,N2)
      rp_add_rp%coeff(N-I+1) = poly1%coeff(N1-I+1) + poly2%coeff(N2-I+1)
    END DO
    ! At most one of the next two DO loops is ever executed
    DO I=N1+1,N
      rp_add_rp%coeff(N-I+1) = poly2%coeff(N2-I+1)
    END DO
    DO I=N2+1,N
      rp_add_rp%coeff(N-I+1) = poly1%coeff(N1-I+1)
    END DO
  END FUNCTION
END MODULE
!
! Sample program
!
PROGRAM example
  USE real_poly_module
  TYPE(real_poly) p,q,r
  p = real_poly((/1.0,2.0,4.0/))   ! x**2 + 2x + 4
  q = real_poly((/1.0,-5.5/))      ! x - 5.5
  r = p + q                        ! x**2 + 3x - 1.5
  print 1,'The coefficients of the answer are:',r%coeff
1 format(1x,A,3F8.2)
END
```

When executed, the above program prints:

```
 The coefficients of the answer are:    1.00    3.00   -1.50
```

4.5 The MOVE_ALLOC intrinsic subroutine [5.2]

This subroutine moves an allocation from one allocatable variable to another. This can be used to expand an allocatable array with only one copy operation, and allows full control over where in the new array the values should go. For example:

```
REAL,ALLOCATABLE :: a(:),tmp(:)
...
ALLOCATE(a(n))
...
! Here we want to double the size of A, without losing any of the values
! that are already stored in it.
ALLOCATE(tmp(size(a)*2))
tmp(1:size(a)) = a
CALL move_alloc(from=tmp,to=a)
! TMP is now deallocated, and A has the new size and values.
```

To have the values end up somewhere different, just change the assignment statement, for example to move them all to the end:

```
tmp(size(a)+1:size(a)*2) = a
```

4.6 Allocatable scalars [5.2]

The ALLOCATABLE attribute may now be applied to scalar variables and components, not just arrays. This is most useful in conjunction with polymorphism (`CLASS`) and/or deferred type parameters (e.g. `CHARACTER(:)`); for more details see the "Typed allocation", "Sourced allocation" and "Automatic reallocation" sections.

4.7 Automatic reallocation [5.2]

If, in an assignment to a whole allocatable array, the expression being assigned is an array of a different size or shape, the allocatable array is reallocated to have the correct shape (in Fortran 95 this assignment would have been an error). For example:

```
ALLOCATE(a(10))
...
a = (/ (i,i=1,100) /)
! A is now size 100
```

Similarly, if an allocatable variable has a deferred type parameter (these are described in a later section), and is either unallocated or has a value different from that of the expression, the allocatable variable is reallocated to have the same value for that type parameter. This allows for true varying-length character variables:

```
CHARACTER(:),ALLOCATABLE :: name
...
name = 'John Smith'
! LEN(name) is now 10, whatever it was before.
name = '?'
! LEN(name) is now 1.
```

Note that since a subobject of an allocatable object is not itself allocatable, this automatic reallocation can be suppressed by using substrings (for characters) or array sections (for arrays), e.g.

```
name(:) = '?'             ! Normal assignment with truncation/padding.
a(:) = (/ (i,i=1,100) /)  ! Asserts that A is already of size 100.
```

5 Other data-oriented enhancements

5.1 Parameterised derived types [6.0 for kind type parameters, 6.1 for length]

Derived types may now have type parameters. Like those of the intrinsic types, they come in two flavours: "kind"-like ones which must be known at compile time (called "kind" type parameters), and ones like character length which may vary at runtime (called "length" type parameters).

5.1.1 Basic Syntax and Semantics

A derived type which has type parameters must list them in the type definition, give them a type, and specify whether they are "kind" or "length" parameters. For example,

```
TYPE real_matrix(kind,n,m)
  INTEGER,KIND :: kind
  INTEGER(int64),LEN :: n,m
```

All type parameters must be explicitly specified to be of type `INTEGER`, but the kind of integer may vary. Type parameters are always scalar, never arrays. Within the type definition, "kind" type parameters may be used in constant expressions, and any type parameter may be used in a specification expression (viz array bound, character length, or "length" type parameter value). For example, the rest of the above type definition might look like:

```
  REAL(kind) value(n,m)
END TYPE real_matrix
```

When declaring entities of such a derived type, the type parameters must be given after the name. For example,

```
TYPE(real_matrix(KIND(0d0),100,200)) :: my_real_matrix_variable
```

Similarly, the type parameters must be given when constructing values of such a type; for example,

```
my_real_matrix_variable = &
  real_matrix(kind(0d0),100,200)((/ (i*1.0d0,i=1,20000) /))
```

To examine the value of a derived type parameter from outside the type definition, the same notation is used as for component accesses, e.g.

```
print *,'Columns =',my_real_matrix_variable%m
```

Thus type parameter names are in the same class as component names and type-bound procedure names. However, a type parameter cannot be changed by using its specifier on the left-hand-side of an assignment. Furthermore, the intrinsic type parameters may also be examined using this technique, for example:

```
REAL :: array(:,:)
CHARACTER(*),INTENT(IN) :: ch
PRINT *,array%kind,ch%len
```

prints the same values as for `KIND(array)` and `LEN(ch)`. Note that a kind parameter enquiry is always scalar, even if the object is an array.

A derived type parameter does not actually have to be used at all within the type definition, and a kind type parameter might only be used within specification expressions. For example,

```
TYPE fixed_byte(n)
  INTEGER,KIND :: n
```

```
  INTEGER(1) :: value(n)
END TYPE
TYPE numbered_object(object_number)
  INTEGER,LEN :: object_number
END TYPE
```

Even though the `fixed_byte` parameter `n` is not used in a constant expression, a constant value must always be specified for it because it has been declared to be a "kind" type parameter. Similarly, even though `object_number` has not been used at all, a value must always be specified for it. This is not quite as useless as it might seem: each `numbered_object` has a single value for `object_number` even if the `numbered_object` is an array.

5.1.2 More Semantics

A derived type with type parameters can have default values for one or more of them; in this case the parameters with default values may be omitted from the type specifiers. For example,

```
TYPE char_with_maxlen(maxlen,kind)
  INTEGER,LEN :: maxlen = 254
  INTEGER,KIND   :: kind = SELECTED_CHAR_KIND('ascii')
  INTEGER        :: len = 0
  CHARACTER(len=maxlen,kind=kind) :: value
END TYPE
...
TYPE(char_with_maxlen) temp
TYPE(char_with_maxlen(80)) card(1000)
TYPE(char_with_maxlen(kind=SELECTED_CHAR_KIND('iso 10646'))) ucs4_temp
```

Note that although kind type parameters can be used in constant expressions and thus in default initialisation, components that are variable-sized (because they depend on length type parameters) cannot be default-initialised at all. Thus `value` in the example above cannot be default-initialised.

Further note that unlike intrinsic types, there are no automatic conversions for derived type assignment with different type parameter values, thus given the above declarations,

```
card(1) = card(2) ! This is ok, maxlen==80 for both sides.
temp = card       ! This is not allowed - maxlen 254 vs. maxlen 80.
```

5.1.3 Assumed type parameters

Assumed type parameters for derived types work similarly to character length, except that they are only allowed for dummy arguments (not for named constants). For example, the following subroutine works on any `char_with_maxlen` variable.

```
SUBROUTINE stars(x)
  TYPE(char_with_maxlen(*)) x
  x%value = REPEAT('*',x%maxlen)
END SUBROUTINE
```

5.1.4 Deferred type parameters

Deferred type parameters are completely new to Fortran 2003; these are available both for `CHARACTER` and for parameterised derived types, and work similarly to deferred array bounds. A variable with a deferred type parameter must have the `ALLOCATABLE` or `POINTER` attribute. The value of a deferred type parameter for an allocatable variable is that determined by allocation (either by a typed allocation, or by an intrinsic assignment with automatic reallocation). For a pointer, the value of a deferred type parameter is the value of the type parameter of its target. For example, using the type `real_matrix` defined above,

```
TYPE(real_matrix(KIND(0.0),100,200)),TARGET :: x
TYPE(real_matrix(KIND(0.0),:,:)),POINTER :: y, z
ALLOCATE(real_matrix(KIND(0.0),33,44) :: y)  ! Typed allocation.
z => x                                       ! Assumes from the target.
PRINT *,y%n,z%n                              ! Prints 33 and 100.
```

Note that it is not allowed to reference the value of a deferred type parameter of an unallocated allocatable or of a pointer that is not associated with a target.

If a dummy argument is allocatable or a pointer, the actual argument must have deferred exactly the same type parameters as the dummy. For example,

```
SUBROUTINE sub(rm_dble_ptr)
  TYPE(real_matrix(KIND(0d0),*,:)),POINTER :: rm_dble_ptr
...
TYPE(real_matrix(KIND(0d0),100,200)),POINTER :: x
TYPE(real_matrix(KIND(0d0),100,:)),POINTER :: y
TYPE(real_matrix(KIND(0d0),:,:)),POINTER :: z
CALL sub(x)  ! Invalid - X%M is not deferred (but must be).
CALL sub(y)  ! This is ok.
CALL sub(z)  ! Invalid - X%N is deferred (but must not be).
```

5.2 Finalisation [5.3]

An extensible derived type can have "final subroutines" associated with it; these subroutines are automatically called whenever an object of the type is about to be destroyed, whether by deallocation, procedure return, being on the left-hand-side of an intrinsic assignment, or being passed to an `INTENT(OUT)` dummy argument.

A final subroutine of a type must be a subroutine with exactly one argument, which must be an ordinary dummy variable of that type (and must not be `INTENT(OUT)`). It may be scalar or an array, and when an object of that type is destroyed the final subroutine whose argument has the same rank as the object is called. The final subroutine may be elemental, in which case it will handle any rank of object that has no other subroutine handling it. Note that if there is no final subroutine for the rank of an object, no subroutine will be called.

Final subroutines are declared in the type definition after the `CONTAINS` statement, like type-bound procedures. They are declared by a `FINAL` statement, which has the syntax

`FINAL` [`::`] *name* [`,` *name*]...

where each *name* is a subroutine that satisfies the above rules.

A simple type with a final subroutine is as follows.

```
TYPE flexible_real_vector
  LOGICAL :: value_was_allocated = .FALSE.
  REAL,POINTER :: value(:) => NULL()
CONTAINS
  FINAL destroy_frv
END TYPE
...
ELEMENTAL SUBROUTINE destroy_frv(x)
  TYPE(flexible_real_vector),INTENT(INOUT) :: x
  IF (x%value_was_allocated) DEALLOCATE(x%value)
END SUBROUTINE
```

If an object being destroyed has finalisable components, any final subroutine for the object-as-a-whole will be called before finalising any components. If the object is an array, each component will be finalised separately (and any final subroutine called will be the one for the rank of the component, not the rank of the object).

For example, in

```
TYPE many_vectors
  TYPE(flexible_real_vector) scalar
  TYPE(flexible_real_vector) array(2,3)
CONTAINS
  FINAL :: destroy_many_vectors_1
END TYPE
...
SUBROUTINE destroy_many_vectors_1(array1)
  TYPE(many_vectors) array1(:)
  PRINT *,'Destroying a',SIZE(array1),'element array of many vectors'
END SUBROUTINE
...
TYPE(many_vector) mv_object(3)
```

when `mv_object` is destroyed, firstly '`destroy_many_vectors_1`' will be called with `mv_object` as its argument; this will print

```
Destroying a 3 element array of many vectors
```

Secondly, for each element of the array, both the `scalar` and `array` components will be finalised by calling `destroy_frv` on each of them. These may be done in any order (or, since they are elemental, potentially in parallel).

Note that final subroutines are not inherited through type extension; instead, when an object of extended type is destroyed, first any final subroutine it has will be called, then any final subroutine of the parent type will be called on the parent component, and so on.

5.3 The PROTECTED attribute [5.0]

The `PROTECTED` attribute may be specified by the `PROTECTED` statement or with the `PROTECTED` keyword in a type declaration statement. It protects a module variable against modification from outside the module.

5.3.1 Syntax

The syntax of the `PROTECTED` statement is:

`PROTECTED` [`::`] *name* [`,` *name*] ...

The `PROTECTED` attribute may only be specified for a variable in a module.

5.3.2 Semantics

Variables with the `PROTECTED` attribute may only be modified within the defining module. Outside of that module they are not allowed to appear in a variable definition context (e.g. on the left-hand-side of an assignment statement), similar to `INTENT(IN)` dummy arguments.

This allows the module writer to make the values of some variables generally available without relinquishing control over their modification.

5.3.3 Example

```
MODULE temperature_module
  REAL,PROTECTED :: temperature_c = 0, temperature_f = 32
CONTAINS
  SUBROUTINE set_temperature_c(new_value_c)
    REAL,INTENT(IN) :: new_value_c
```

```
    temperature_c = new_value_c
    temperature_f = temperature_c*(9.0/5.0) + 32
  END SUBROUTINE
  SUBROUTINE set_temperature_f(new_value_f)
    REAL,INTENT(IN) :: new_value_f
    temperature_f = new_value_f
    temperature_c = (temperature_f - 32)*(5.0/9.0)
  END SUBROUTINE
END
```

The PROTECTED attribute allows users of temperature_module to read the temperature in either Farenheit or Celsius, but the variables can only be changed via the provided subroutines which ensure that both values agree.

5.4 Pointer enhancements

5.4.1 INTENT for pointers [5.1]

A POINTER dummy argument may now have the INTENT attribute. This attribute applies to the pointer association status, not to the target of the pointer.

An INTENT(IN) pointer can be assigned to, but cannot be pointer-assigned, nullified, allocated or deallocated. An INTENT(OUT) pointer receives an undefined association status on entry to the procedure. An INTENT(INOUT) pointer has no restrictions on its use, but the actual argument must be a pointer variable, not a pointer function reference.

5.4.2 Pointer bounds specification [5.2]

The bounds of a pointer can be changed (from the default) in a pointer assignment by including them on the left-hand-side. For example,

```
REAL,TARGET :: x(-100:100,-10:10)
REAL,POINTER :: p(:,:)
p(1:,1:) => x
```

The upper bound is formed by adding the extent (minus 1) to the lower bound, so in the above example, the bounds of P will be 1:201,1:21. Note that when setting the lower bound of any rank in a pointer assignment, the values must be explicitly specified (there is no default of 1 like there is in array declarators) and they must be specified for all dimensions of the pointer.

5.4.3 Rank-remapping Pointer Assignment [5.0]

This feature allows a multi-dimensional pointer to point to a single-dimensional object. For example:

```
REAL,POINTER :: diagonal(:),matrix(:,:),base(:)
...
ALLOCATE(base(n*n))
matrix(1:n,1:n) => base
diagonal => base(::n+1)
!
! DIAGONAL now points to the diagonal elements of MATRIX.
!
```

Note that when rank-remapping, the values for both the lower and upper bounds must be explicitly specified for all dimensions, there are no defaults.

5.5 Individual component accessibility [5.1]

It is now possible to set the accessibility of individual components in a derived type. For example,

```
TYPE t
  LOGICAL, PUBLIC :: flag
  INTEGER, PRIVATE :: state
END TYPE
```

The structure constructor for the type is not usable from outside the defining module if there is any private component that is not inherited, allocatable or default-initialised (see Structure constructor syntax enhancements).

5.6 Public entities of private type [5.1]

It is now possible to export entities (named constants, variables, procedures) from a module even if they have private type or (for procedures) have arguments of private type. For example,

```
MODULE m
  TYPE, PRIVATE :: hidden_type
    CHARACTER(6) :: code
  END TYPE
  TYPE(hidden_type), PUBLIC, PARAMETER :: code_green = hidden_type('green')
  TYPE(hidden_type), PUBLIC, PARAMETER :: code_yellow = hidden_type('yellow')
  TYPE(hidden_type), PUBLIC, PARAMETER :: code_red = hidden_type('red')
END
```

6 C interoperability [mostly 5.1]

6.1 The ISO_C_BINDING module

The intrinsic module ISO_C_BINDING contains

- for each C type (e.g. `float`), a named constant for use as a KIND parameter for the corresponding Fortran type,
- types `C_PTR` and `C_FUNPTR` for interoperating with C object pointers and function pointers,
- procedures for manipulating Fortran and C pointers.

6.1.1 The kind parameters

The kind parameter names are for using with the corresponding Fortran types; for example, `INTEGER` for integral types and `REAL` for floating-point types. This is shown in the table below. Note that only `c_int` is guaranteed to be available; if there is no compatible type the value will be negative.

C type	Fortran type and kind
_Bool	LOGICAL(c_bool)
char	CHARACTER(c_char) — For characters as text.
double	REAL(c_double)
double _Complex	COMPLEX(c_double_complex) or COMPLEX(c_double)
float	REAL(c_float)
float _Complex	COMPLEX(c_float_complex) or COMPLEX(c_float)
int	INTEGER(c_int)
int16_t	INTEGER(c_int16_t)
int32_t	INTEGER(c_int32_t)
int64_t	INTEGER(c_int64_t)
int8_t	INTEGER(c_int8_t)
int_fast16_t	INTEGER(c_int_fast16_t)
int_fast32_t	INTEGER(c_int_fast32_t)
int_fast64_t	INTEGER(c_int_fast64_t)
int_fast8_t	INTEGER(c_int_fast8_t)
int_least16_t	INTEGER(c_int_least16_t)
int_least32_t	INTEGER(c_int_least32_t)
int_least64_t	INTEGER(c_int_least64_t)
int_least8_t	INTEGER(c_int_least8_t)
intmax_t	INTEGER(c_intmax_t)
intptr_t	INTEGER(c_intptr_t)
long	INTEGER(c_long)
long double	REAL(c_long_double)
long double _Complex	COMPLEX(c_long_double_complex) or COMPLEX(c_long_double)
long long	INTEGER(c_long_long)
short	INTEGER(c_short)
signed char	INTEGER(c_signed_char) — For characters as integers.
size_t	INTEGER(c_size_t)

6.1.2 Using C_PTR and C_FUNPTR

These are derived type names, so you use them as Type(c_ptr) and Type(c_funptr). Type(c_ptr) is essentially equivalent to the C void *; i.e. it can contain any object pointer. Type(c_funptr) does the same thing for function pointers.

For C arguments like 'int *', you don't need to use Type(c_ptr), you can just use a normal dummy argument (in this case of type Integer(c_int)) without the VALUE attribute. However, for more complicated pointer arguments such as pointer to pointer, or for variables or components that are pointers, you need to use Type(c_ptr).

Null pointer constants of both Type(c_ptr) and Type(c_funptr) are provided: these are named C_NULL_PTR and C_NULL_FUNPTR respectively.

To create a Type(c_ptr) value, the function C_LOC(X) is used on a Fortran object X (and X must have the TARGET attribute). Furthermore, the Fortran object cannot be polymorphic, a zero-sized array, an assumed-size array, or an array pointer. To create a Type(c_funptr) value, the function C_FUNLOC is used on a procedure; this procedure must have the BIND(C) attribute.

To test whether a Type(c_ptr) or Type(c_funptr) is null, the C_ASSOCIATED(C_PTR_1) function can be used; it returns .TRUE. if and only if C_PTR_1 is not null. Two Type(c_ptr) or two Type(c_funptr) values can be compared using C_ASSOCIATED(C_PTR_1,C_PTR_2) function; it returns .TRUE. if and only if C_PTR_1 contains the same C address as C_PTR_2.

The subroutine C_F_POINTER(CPTR,FPTR) converts the TYPE(C_PTR) value CPTR to the scalar Fortran pointer FPTR; the latter can have any type (including non-interoperable types) but must not be polymorphic. The subroutine C_F_POINTER(CPTR,FPTR,SHAPE) converts a TYPE(C_PTR) value into the Fortran array pointer FPTR, where SHAPE is an integer array of rank 1, with the same number of elements as the rank of FPTR; the lower bounds of the resultant FPTR will all be 1.

The subroutine C_F_PROCPOINTER(CPTR,FPTR) is provided. This converts the TYPE(C_FUNPTR) CPTR to the Fortran procedure pointer FPTR.

Note that in all the conversion cases it is up to the programmer to use the correct type and other information.

6.2 BIND(C) types

Derived types corresponding to C `struct` types can be created by giving the type the `BIND(C)` attribute, e.g.

```
TYPE,BIND(C) :: mytype
```

The components of a BIND(C) type must have types corresponding to C types, and cannot be pointers or allocatables. Furthermore, a BIND(C) type cannot be a `SEQUENCE` type (it already acts like a `SEQUENCE` type), cannot have type-bound procedures, cannot have final procedures, and cannot be extended.

6.3 BIND(C) variables

Access to C global variables is provided by giving the Fortran variable the `BIND(C)` attribute. Such a variable can only be declared in a module, and cannot be in a `COMMON` block. By default, the C name of the variable is the Fortran name converted to all lowercase characters; a different name may be specified with the `NAME=` clause, e.g.

```
INTEGER,BIND(C,NAME="StrangelyCapiTalisedCName") :: x
```

Within Fortran code, the variable is referred to by its Fortran name, not its C name.

6.4 BIND(C) procedures

A Fortran procedure that can be called from C can be defined using the BIND(C) attribute on the procedure heading. By default its C name is the Fortran name converted to lowercase; a different name may be specified with the `NAME=` clause. For example

```
SUBROUTINE sub() BIND(C,NAME='Sub')
...
```

Again, the C name is for use only from C, the Fortran name is used from Fortran. If the C name is all blanks (or a zero-length string), there is no C name. Such a procedure can still be called from C via a procedure pointer (i.e. by assigning it to a `TYPE(C_FUNPTR)` variable).

A `BIND(C)` procedure must be a module procedure or external procedure with an explicit interface; it cannot be an internal procedure or statement function.

A `BIND(C)` procedure may be a subroutine or a scalar function with a type corresponding to a C type. Each dummy argument must be a variable whose type corresponds to a C type, and cannot be allocatable, assumed-shape, optional or a pointer. If the dummy argument does not have the `VALUE` attribute, it corresponds to a C dummy argument that is a pointer.

Here is an example of a Fortran procedure together with its reference from C:

```
SUBROUTINE find_minmax(x,n,max,min) BIND(C,NAME='FindMinMax')
  USE iso_c_binding
  REAL(c_double) x(*),max,min
  INTEGER(c_int),VALUE :: n
  INTRINSIC maxval,minval
  max = MAXVAL(x(:n))
  min = MINVAL(x(:n))
END
```

```
extern void FindMinMax(double *x,int n,double *maxval,double *minval);
double x[100],xmax,xmin;
```

```
int n;
...
FindMinMax(x,n,&xmax,&xmin);
```

This also allows C procedures to be called from Fortran, by describing the C procedure to be called in an interface block. Here is an example:

```
/* This is the prototype for a C library function from 4.3BSD. */
int getloadavg(double loadavg[],int nelem);
```

```
PROGRAM show_loadavg
  USE iso_c_binding
  INTERFACE
    FUNCTION getloadavg(loadavg,nelem) BIND(C)
      IMPORT c_double,c_int
      REAL(c_double) loadavg(*)
      INTEGER(c_int),VALUE :: nelem
      INTEGER(c_int) getloadavg
    END FUNCTION
  END INTERFACE
  REAL(c_double) averages(3)
  IF (getloadavg(averages,3)/=3) THEN
    PRINT *,'Unexpected error'
  ELSE
    PRINT *,'Load averages:',averages
  END IF
END
```

6.5 Enumerations

An enumeration defines a set of integer constants of the same kind, and is equivalent to the C `enum` declaration. For example,

```
  ENUM,BIND(C)
    ENUMERATOR :: open_door=4, close_door=17
    ENUMERATOR :: lock_door
  END ENUM
```

is equivalent to

```
  enum {
    open_door=4, close_door=17, lock_door
  };
```

If a value is not given for one of the enumerators, it will be one greater than the previous value (or zero if it is the first enumerator in the list). The kind used for a particular set of enumerators can be discovered by using the `KIND` intrinsic on one of the enumerators.

Note that the `BIND(C)` clause is required; the standard only defines enumerations for interoperating with C.

7 IEEE arithmetic support [4.x except as otherwise noted]

7.1 Introduction

Three intrinsic modules are provided to support use of IEEE arithmetic, these are: `IEEE_ARITHMETIC`, `IEEE_EXCEPTIONS` and `IEEE_FEATURES`. This extension is small superset of the one described by the ISO Technical Report ISO/IEC TR 15580:1999.

7.2 Exception flags, modes and information flow

The model of IEEE arithmetic used by Fortran 2003 is that there is a global set of flags that indicate whether particular floating-point exceptions (such as overflow) have occurred, several operation modes which affect floating-point operations and exception handling, and a set of rules for propagating the flags and modes to obtain good performance and reliability.

The propagation rule for the exception flags is that the information "flows upwards". Thus each procedure starts with the flags clear, and when it returns any flag that is set will cause the corresponding flag in the caller to become set. This ensures that procedures that can be executed in parallel will not interfere with each other via the IEEE exception flags. When the computer hardware only supports a single global set of flags, this model needs enforcement on in procedures that themselves examine the flags (by `IEEE_GET_FLAG` or `IEEE_GET_STATUS`).

The propagation rule for modes, is that the mode settings "flow downwards". This enables code motion optimisations within all routines, and the only cost is that procedures which modify the modes must restore them (to the state on entry) when they return.

The modes that are available are:

- separately, whether each floating point exception terminates the program or allows execution to continue (providing a default result and raising an exception flag);
- rounding mode for floating-point operations;
- underflow mode.

7.3 Procedures in the modules

All the procedures provided by these modules are generic procedures, and not specific: that means that they cannot be passed as actual arguments.

In the descriptions of the procedures, where it says `REAL(*)` it means any kind of `REAL` (this is not standard Fortran syntax). Conversely, where it says `LOGICAL` it means default `LOGICAL` only, not any other kind of `LOGICAL`.

The functions whose names begin '`IEEE_SUPPORT_`' are all enquiry functions. Many of these take a `REAL(*)` argument `X`; only the kind of `X` is used by the enquiry function, so `X` is permitted to be undefined, unallocated, disassociated, or an undefined pointer.

Note that a procedure must not be invoked on a data type that does not support the feature the procedure uses; the "support" enquiry functions can be used to detect this.

7.4 The IEEE_FEATURES module

This module defines the derived type `IEEE_FEATURES_TYPE`, and up to 11 constants of that type representing IEEE features: these are as follows.

`IEEE_DATATYPE`	whether any IEEE datatypes are available
`IEEE_DENORMAL`	whether IEEE denormal values are available*
`IEEE_DIVIDE`	whether division has the accuracy required by IEEE*
`IEEE_HALTING`	whether control of halting is supported
`IEEE_INEXACT_FLAG`	whether the inexact exception is supported*
`IEEE_INF`	whether IEEE positive and negative infinities are available*
`IEEE_INVALID_FLAG`	whether the invalid exception is supported*
`IEEE_NAN`	whether IEEE NaNs are available*
`IEEE_ROUNDING`	whether all IEEE rounding modes are available*
`IEEE_SQRT`	whether SQRT conforms to the IEEE standard*
`IEEE_UNDERFLOW_FLAG`	whether the underflow flag is supported*

(*) for at least one kind of `REAL`.

Those feature types which are required by the user procedure should be explicitly referenced by the `USE` statement with an `ONLY` clause, e.g.

```
  USE,INTRINSIC :: IEEE_FEATURES,ONLY:IEEE_SQRT
```

This ensures that if the feature specified is not available the compilation will fail.

The type `IEEE_FEATURES_TYPE` is not in itself useful.

7.5 IEEE_EXCEPTIONS

Provides data types, constants and generic procedures for handling IEEE floating-point exceptions.

7.5.1 Types and constants

`TYPE IEEE_STATUS_TYPE`

Variables of this type can hold a floating-point status value; it combines all the mode settings and flags.

`TYPE IEEE_FLAG_TYPE`

Values of this type specify individual IEEE exception flags; constants for these are available as follows.

`IEEE_DIVIDE_BY_ZERO`	division by zero flag
`IEEE_INEXACT`	inexact result flag
`IEEE_INVALID`	invalid operation flag
`IEEE_OVERFLOW`	overflow flag
`IEEE_UNDERFLOW`	underflow flag

In addition, two array constants are available for indicating common combinations of flags:

```
  TYPE(IEEE_FLAG_TYPE),PARAMETER :: &
     IEEE_USUAL(3) = (/ IEEE_DIVIDE_BY_ZERO,IEEE_INVALID,IEEE_OVERFLOW /), &
     IEEE_ALL(5) = (/ IEEE_DIVIDE_BY_ZERO,IEEE_INVALID,IEEE_OVERFLOW, &
                      IEEE_UNDERFLOW,IEEE_INEXACT /)
```

7.5.2 Procedures

The procedures provided by `IEEE_EXCEPTIONS` are as follows.

```
ELEMENTAL SUBROUTINE IEEE_GET_FLAG(FLAG,FLAG_VALUE)
  TYPE(IEEE_FLAG_TYPE),INTENT(IN) :: FLAG
  LOGICAL,INTENT(OUT) :: FLAG_VALUE
```

Sets `FLAG_VALUE` to `.TRUE.` if the exception flag indicated by `FLAG` is currently set, and to `.FALSE.` otherwise.

```
ELEMENTAL SUBROUTINE IEEE_GET_HALTING_MODE(FLAG,HALTING)
  TYPE(IEEE_FLAG_TYPE),INTENT(IN) :: FLAG
  LOGICAL,INTENT(OUT) :: HALTING
```

Sets `HALTING` to `.TRUE.` if the program will be terminated on the occurrence of the floating-point exception designated by `FLAG`, and to `.FALSE.` otherwise.

```
PURE SUBROUTINE IEEE_GET_STATUS(STATUS_VALUE)
  TYPE(IEEE_STATUS_TYPE),INTENT(OUT) :: STATUS_VALUE
```

Sets `STATUS_VALUE` to the current floating-point status; this contains all the current exception flag and mode settings.

```
PURE SUBROUTINE IEEE_SET_FLAG(FLAG,FLAG_VALUE)
  TYPE(IEEE_FLAG_TYPE),INTENT(IN) :: FLAG
  LOGICAL,INTENT(IN) :: FLAG_VALUE
```

Sets the exception flag designated by `FLAG` to `FLAG_VALUE`. `FLAG` may be an array of any rank, as long as it has no duplicate values, in which case `FLAG_VALUE` may be scalar or an array with the same shape.

```
PURE SUBROUTINE IEEE_SET_HALTING_MODE(FLAG,HALTING)
  TYPE(IEEE_FLAG_TYPE),INTENT(IN) :: FLAG
  LOGICAL,INTENT(IN) :: HALTING
```

Sets the halting mode for the exception designated by `FLAG` to `HALTING`. `FLAG` may be an array of any rank, as long as it has no duplicate values, in which case `HALTING` may be scalar or an array with the same shape.

```
PURE SUBROUTINE IEEE_SET_STATUS(STATUS_VALUE)
  TYPE(IEEE_STATUS_TYPE),INTENT(IN) :: STATUS_VALUE
```

Sets the floating-point status to that stored in `STATUS_VALUE`. This must have been previously obtained by calling `IEEE_GET_STATUS`.

```
PURE LOGICAL FUNCTION IEEE_SUPPORT_FLAG(FLAG)
  TYPE(IEEE_FLAG_TYPE),INTENT(IN) :: FLAG
```

Returns whether the exception flag designated by `FLAG` is supported for all kinds of `REAL`.

```
PURE LOGICAL FUNCTION IEEE_SUPPORT_FLAG(FLAG,X)
  TYPE(IEEE_FLAG_TYPE),INTENT(IN) :: FLAG
  REAL(*),INTENT(IN) :: X
```

Returns whether the exception flag designated by `FLAG` is supported for `REAL` with the kind of `X`.

```
PURE LOGICAL FUNCTION IEEE_SUPPORT_HALTING(FLAG)
  TYPE(IEEE_FLAG_TYPE),INTENT(IN) :: FLAG
```

Returns whether control of the "halting mode" for the exception designated by `FLAG` is supported.

7.6 IEEE_ARITHMETIC module

Provides additional functions supporting IEEE arithmetic: it includes the entire contents of `IEEE_EXCEPTIONS`.

7.6.1 IEEE datatype selection

The `IEEE_SELECTED_REAL_KIND` function is similar to the `SELECTED_REAL_KIND` intrinsic function, but selects among IEEE-compliant `REAL` types ignoring any that are not compliant.

7.6.2 Enquiry functions

```
PURE LOGICAL FUNCTION IEEE_SUPPORT_DATATYPE()
```

Returns whether all real variables `X` satisfy the conditions for `IEEE_SUPPORT_DATATYPE(X)`.

```
PURE LOGICAL FUNCTION IEEE_SUPPORT_DATATYPE(X)
  REAL(*),INTENT(IN) :: X
```

Returns whether variables with the kind of `X` satisfy the following conditions:

- the numbers with absolute value between `TINY(X)` and `HUGE(X)` are exactly those of an IEEE floating-point format;
- the `+`, `-` and `*` operations conform to IEEE for at least one rounding mode;
- the functions `IEEE_COPY_SIGN`, `IEEE_LOGB`, `IEEE_NEXT_AFTER`, `IEEE_REM`, `IEEE_SCALB` and `IEEE_UNORDERED` may be used.

```
PURE LOGICAL FUNCTION IEEE_SUPPORT_DENORMAL()

PURE LOGICAL FUNCTION IEEE_SUPPORT_DENORMAL(X)
  REAL(*),INTENT(IN) :: X
```

Returns whether for all real kinds, or variables with the kind of `X`, subnormal values (with absolute value between zero and `TINY`) exist as required by IEEE and operations on them conform to IEEE.

```
PURE LOGICAL FUNCTION IEEE_SUPPORT_DIVIDE()

PURE LOGICAL FUNCTION IEEE_SUPPORT_DIVIDE(X)
  REAL(*),INTENT(IN) :: X
```

Returns whether intrinsic division (`/`) conforms to IEEE, for all real kinds or variables with the kind of `X` respectively.

```
PURE LOGICAL FUNCTION IEEE_SUPPORT_INF()

PURE LOGICAL FUNCTION IEEE_SUPPORT_INF(X)
  REAL(*),INTENT(IN) :: X
```

Returns whether for all real kinds, or variables with the kind of `X`, positive and negative infinity values exist and behave in conformance with IEEE.

```
PURE LOGICAL FUNCTION IEEE_SUPPORT_IO()

PURE LOGICAL FUNCTION IEEE_SUPPORT_IO(X)
  REAL(*),INTENT(IN) :: X
```

[5.2] Returns whether for all real kinds, or variables with the kind of `X`, conversion to and from text during formatted input/output conforms to IEEE, for the input/output rounding modes `ROUND='DOWN'`, `'NEAREST'`, `'UP'` and `'ZERO'` (and the corresponding edit descriptors `RD`, `RN`, `RU` and `RZ`).

```
PURE LOGICAL FUNCTION IEEE_SUPPORT_NAN()

PURE LOGICAL FUNCTION IEEE_SUPPORT_NAN(X)
  REAL(*),INTENT(IN) :: X
```

Returns whether for all real kinds, or variables with the kind of `X`, positive and negative "Not-a-Number" values exist and behave in conformance with IEEE.

```
PURE LOGICAL FUNCTION IEEE_SUPPORT_ROUNDING(ROUND_VALUE)
  TYPE(IEEE_ROUND_TYPE),INTENT(IN) :: ROUND_VALUE

PURE LOGICAL FUNCTION IEEE_SUPPORT_ROUNDING(ROUND_VALUE,X)
  TYPE(IEEE_ROUND_TYPE),INTENT(IN) :: ROUND_VALUE
  REAL(*),INTENT(IN) :: X
```

Returns whether for all real kinds, or variables with the kind of X, the rounding mode designated by ROUND_VALUE may be set using IEEE_SET_ROUNDING_MODE and conforms to IEEE.

```
PURE LOGICAL FUNCTION IEEE_SUPPORT_SQRT()

PURE LOGICAL FUNCTION IEEE_SUPPORT_SQRT(X)
  REAL(*),INTENT(IN) :: X
```

Returns whether the intrinsic function SQRT conforms to IEEE, for all real kinds or variables with the kind of X respectively.

```
PURE LOGICAL FUNCTION IEEE_SUPPORT_STANDARD()

PURE LOGICAL FUNCTION IEEE_SUPPORT_STANDARD(X)
  REAL(*),INTENT(IN) :: X
```

Returns whether for all real kinds, or variables with the kind of X, all the facilities described by the IEEE modules except for input/output conversions (see IEEE_SUPPORT_IO) are supported and conform to IEEE.

```
PURE LOGICAL FUNCTION IEEE_SUPPORT_UNDERFLOW_CONTROL()

PURE LOGICAL FUNCTION IEEE_SUPPORT_UNDERFLOW_CONTROL(X)
  REAL(*),INTENT(IN) :: X
```

[5.2] Returns whether for all real kinds, or variables with the kind of X, the underflow mode can be controlled with IEEE_SET_UNDERFLOW_MODE.

7.6.3 Rounding mode

```
TYPE IEEE_ROUND_TYPE
```

Values of this type specify the IEEE rounding mode. The following predefined constants are provided.

IEEE_DOWN	round down (towards minus infinity)
IEEE_NEAREST	round to nearest (ties to even)
IEEE_TO_ZERO	round positive numbers down, negative numbers up
IEEE_UP	round up (towards positive infinity)
IEEE_OTHER	any other rounding mode

```
PURE SUBROUTINE IEEE_GET_ROUNDING_MODE(ROUND_VALUE)
  TYPE(IEEE_ROUND_TYPE),INTENT(OUT) :: ROUND_VALUE
```

Set ROUND_VALUE to the current rounding mode.

```
PURE SUBROUTINE IEEE_SET_ROUNDING_MODE(ROUND_VALUE)
  TYPE(IEEE_ROUND_TYPE),INTENT(IN) :: ROUND_VALUE
```

Set the rounding mode to that designated by ROUND_VALUE.

7.6.4 Underflow mode

The underflow mode is either "gradual underflow" as specified by the IEEE standard, or "abrupt underflow".

With gradual underflow, the space between -TINY(X) and TINY(X) is filled with equally-spaced "subnormal" numbers; the spacing of these numbers is equal to the spacing of model numbers above TINY(X) (and equal to the smallest

subnormal number). This gradually reduces the precision of the numbers as they get closer to zero: the smallest number has only one bit of precision, so any calculation with such a number will have a very large relative error.

With abrupt underflow, the only value between `-TINY(X)` and `TINY(X)` is zero. This kind of underflow is nearly universal in non-IEEE arithmetics and is widely provided by hardware even for IEEE arithmetic. Its main advantage is that it can be much faster.

```
SUBROUTINE IEEE_GET_UNDERFLOW_MODE(GRADUAL)
  LOGICAL,INTENT(OUT) :: GRADUAL
```

Sets `GRADUAL` to `.TRUE.` if the current underflow mode is gradual underflow, and to `.FALSE.` if it is abrupt underflow.

```
SUBROUTINE IEEE_SET_UNDERFLOW_MODE(GRADUAL)
  LOGICAL,INTENT(IN) :: GRADUAL
```

Sets the underflow mode to gradual underflow if `GRADUAL` is `.TRUE.`, and to abrupt underflow if it is `.FALSE.`.

7.6.5 Number Classification

```
TYPE IEEE_CLASS_TYPE
```

Values of this type indicate the IEEE class of a number; this is equal to one of the provided constants:

`IEEE_NEGATIVE_DENORMAL`	negative subnormal number x, in the range $-$`TINY(`x`)`$<x<0$
`IEEE_NEGATIVE_INF`	$-\infty$ (negative infinity)
`IEEE_NEGATIVE_NORMAL`	negative normal number x, in the range $-$`HUGE(`x`)`$\leq x\leq -$`TINY(`x`)`
`IEEE_NEGATIVE_ZERO`	-0 (zero with the sign bit set)
`IEEE_POSITIVE_DENORMAL`	positive subnormal number x, in the range $0<x<$`TINY(`x`)`
`IEEE_POSITIVE_INF`	$+\infty$ (positive infinity)
`IEEE_POSITIVE_NORMAL`	positive normal number x, in the range `TINY(`x`)`$\leq x\leq$`HUGE(`x`)`
`IEEE_POSITIVE_ZERO`	$+0$ (zero with the sign bit clear)
`IEEE_QUIET_NAN`	Not-a-Number (usually the result of an invalid operation)
`IEEE_SIGNALING_NAN`	Not-a-Number which raises the invalid signal on reference
[5.2] `IEEE_OTHER_VALUE`	any value that does not fit one of the above categories

The comparison operators `.EQ.` (`=`) and `.NE.` (`/=`) are provided for comparing values of this type.

```
ELEMENTAL TYPE(IEEE_CLASS_TYPE) FUNCTION IEEE_CLASS(X)
  REAL(*),INTENT(IN) :: X
```

returns the classification of the value of `X`.

```
ELEMENTAL REAL(*) FUNCTION IEEE_VALUE(X,CLASS)
  REAL(*),INTENT(IN) :: X
  TYPE(IEEE_CLASS_TYPE),INTENT(IN) :: CLASS
```

Returns a "sample" value with the kind of `X` and the classification designated by `CLASS`.

7.6.6 Test functions

The following procedures are provided for testing IEEE values.

```
ELEMENTAL LOGICAL FUNCTION IEEE_IS_FINITE(X)
  REAL(*),INTENT(IN) :: X
```

Returns whether X is "finite", i.e. not an infinity, NaN, or `IEEE_OTHER_VALUE`.

```
ELEMENTAL LOGICAL FUNCTION IEEE_IS_NAN(X)
  REAL(*),INTENT(IN) :: X
```

Returns whether X is a NaN.

```
ELEMENTAL LOGICAL FUNCTION IEEE_IS_NEGATIVE(X)
  REAL(*),INTENT(IN) :: X
```

Returns whether X is negative; it differs the comparison `X<0` only in the case of negative zero, where it returns `.TRUE.`.

```
ELEMENTAL LOGICAL FUNCTION IEEE_UNORDERED(X,Y)
  REAL(*),INTENT(IN) :: X,Y
```

Returns the value of '`IEEE_IS_NAN(X) .OR. IEEE_IS_NAN(Y)`'.

7.6.7 Arithmetic functions

```
ELEMENTAL REAL(*) FUNCTION IEEE_COPY_SIGN(X,Y)
  REAL(*),INTENT(IN) :: X,Y
```

Returns `X` with the sign bit of `Y`.

```
ELEMENTAL REAL(*) FUNCTION IEEE_LOGB(X)
  REAL(*),INTENT(IN) :: X
```

If `X` is zero, returns $-\infty$ if infinity is supported and `-HUGE(X)` otherwise. For nonzero `X`, returns `EXPONENT(X)-1`

```
ELEMENTAL REAL(*) FUNCTION IEEE_NEXT_AFTER(X,Y)
  REAL(*),INTENT(IN) :: X,Y
```

Returns the nearest number to `X` that is closer to `Y`, or `X` if `X` and `Y` are equal.

```
ELEMENTAL REAL(*) FUNCTION IEEE_REM(X,Y)
  REAL(*),INTENT(IN) :: X,Y
```

Returns the exact remainder resulting from the division of `X` by `Y`.

```
ELEMENTAL REAL(*) FUNCTION IEEE_RINT(X)
  REAL(*),INTENT(IN) :: X
```

Returns `X` rounded to an integer according to the current rounding mode.

```
ELEMENTAL REAL(*) FUNCTION IEEE_SCALB(X,I)
  REAL(*),INTENT(IN) :: X
  INTEGER(*),INTENT(IN) :: I
```

Returns `SCALE(X,I)`, i.e. X*2^I, without computing 2^I separately.

8 Input/output Features

8.1 Stream input/output [5.1]

A stream file is a file that is opened with the `ACCESS='STREAM'` specifier. A stream file is either a formatted stream or an unformatted stream.

A formatted stream file is equivalent to a C text stream; this acts much like an ordinary sequential file, except that there is no limit on the length of a record. Just as in C, when writing to a formatted stream, an embedded newline character in the data causes a new record to be created. The new intrinsic enquiry function `NEW_LINE(A)` returns this character for the kind (character set) of `A`; if the character set is ASCII, this is equal to `IACHAR(10)`. For example,

```
OPEN(17,FORM='formatted',ACCESS='stream',STATUS='new')
WRITE(17,'(A)'), 'This is record 1.'//NEW_LINE('A')//'This is record 2.'
```

An unformatted stream file is equivalent to a C binary stream, and has no record boundaries. This makes it impossible to `BACKSPACE` an unformatted stream. Data written to an unformatted stream is transferred to the file with no formatting, and data read from an unformatted stream is transferred directly to the variable as it appears in the file.

When reading or writing a stream file, the `POS=` specifier may be used to specify where in the file the data is to be written. The first character of the file is at position 1. The `POS=` specifier may also be used in an `INQUIRE` statement, in which case it returns the current position in the file. When reading or writing a formatted stream, the `POS=` in a `READ` or `WRITE` shall be equal to 1 (i.e. the beginning of the file) or to a value previously discovered through `INQUIRE`.

Note that unlike unformatted sequential files, writing to an unformatted stream file at a position earlier than the end of the file does not truncate the file. (However, this truncation does happen for formatted streams.)

Finally, the `STREAM=` specifier has been added to the `INQUIRE` statement. This specifier takes a scalar default character variable, and assigns it the value `'YES'` if the file may be opened for stream input/output (i.e. with `ACCESS='STREAM'`), the value `'NO'` if the file cannot be opened for stream input/output, and the value `'UNKNOWN'` if it is not known whether the file may be opened for stream input/output.

8.2 The BLANK= and PAD= specifiers [5.1]

The `BLANK=` and `PAD=` specifiers, which previously were only allowed on an `OPEN` statement (and `INQUIRE`), are now allowed on a `READ` statement. These change the `BLANK=` or `PAD=` mode for the duration of that `READ` statement only.

8.3 Decimal Comma [5.1]

It is possible to read and write numbers with a decimal comma instead of a decimal point. Support for this is provied by the `DECIMAL=` specifier and the `DC` and `DP` edit descriptors. The `DECIMAL=` specifier may appear in `OPEN`, `READ`, `WRITE` and `INQUIRE` statements; possible values are `'POINT'` (the default) and `'COMMA'`. For an unconnected or unformatted unit, `INQUIRE` returns `'UNDEFINED'`. The `DC` edit descriptor temporarily sets the mode to `DECIMAL='COMMA'`, and the `DP` edit descriptor temporarily sets the mode to `DECIMAL='POINT'`.

When the mode is `DECIMAL='COMMA'`, all floating-point output will produce a decimal comma instead of a decimal point, and all floating-point input will expect a decimal comma. For example,

```
PRINT '(1X,"Value cest ",DC,F0.2)',1.25
```

will produce the output

```
Value cest 1,25
```

Additionally, in this mode, a comma cannot be used in list-directed input to separate items; instead, a semi-colon may be used.

8.4 The DELIM= specifier [5.1]

The DELIM= specifier, which previously was only allowed on an OPEN statement (and INQUIRE), is now allowed on a WRITE statement. It changes the DELIM= mode for the duration of that WRITE statement; note that this only has any effect if the WRITE statement uses list-directed or namelist output. For example,

```
WRITE(*,*,DELIM='QUOTE') "That's all folks!"
```

will produce the output

```
'That''s all folks!'
```

8.5 The ENCODING= specifier [5.1]

The ENCODING= specifier is permitted on OPEN and INQUIRE statements. Standard values for this specifier are 'UTF-8' and the default value of 'DEFAULT'; the 'UTF-8' value is only allowed on compilers that support a Unicode character kind (see SELECTED_CHAR_KIND). This release of the NAG Fortran Compiler supports 'DEFAULT' and 'ASCII'.

8.6 The IOMSG= specifier [5.1]

The IOMSG= specifier has been added to all input/output statements. This takes a scalar default character variable, which in the event of an error is assigned an explanatory message. (Note that this is only useful if the statement contains an IOSTAT= or ERR= specifier, otherwise the program will be terminated on error anyway.) If no error occurs, the value of the variable remains unchanged.

8.7 The IOSTAT= specifier [5.1]

This now accepts any kind of integer variable (previously this was required to be default integer).

8.8 The SIGN= specifier [5.1]

The SIGN= specifier has been added to the OPEN, WRITE and INQUIRE statements; possible values are 'PLUS', 'SUPPRESS' and 'PROCESSOR_DEFINED' (the default). For the NAG Fortran Compiler, SIGN='PROCESSOR_DEFINED' has the same effect as SIGN='SUPPRESS'.

The effect of SIGN='PLUS' is the same as the SP edit descriptor, the effect of SIGN='SUPPRESS' is the same as the SS edit descriptor, and the effect of SIGN='PROCESSOR_DEFINED' is the same as the S edit descriptor.

8.9 Intrinsic functions for testing IOSTAT= values [5.1]

The intrinsic functions IS_IOSTAT_END and IS_IOSTAT_EOR test IOSTAT= return values, determining whether a value indicates an end-of-file condition or an end-of-record condition. These are equivalent to testing the IOSTAT= return value against the named constants IOSTAT_END and IOSTAT_EOR respectively; these constants are available through the ISO_FORTRAN_ENV module.

8.10 Input/output of IEEE infinities and NaNs [5.1]

Input and output of IEEE infinities and NaNs is possible: the output format is

- -Infinity (or -Inf if it will not fit) for negative infinity;
- Infinity (or Inf if it will not fit) for positive infinity, or +Infinity (or +Inf) with SP or SIGN='PLUS' mode;

- `NaN` for a NaN.

Furthermore, the output is right-justified within the output field. For list-directed output the output field is the minimum size to hold the result.

Input of IEEE infinities and NaNs is now possible; these take the same form as the output described above, except that:

- case is not significant,
- a NaN may be preceded by a sign, which is ignored, and
- a NaN may be followed by alphanumeric characters enclosed in parentheses (the NAG Fortran Compiler also ignores these).

The result of reading a NaN value in NAG Fortran is always a quiet NaN, never a signalling one.

8.11 Output of floating-point zero [5.1]

List-directed and namelist output of floating-point zero is now done using F format instead of E format. (The Fortran 90 and 95 standards both specified E format.)

8.12 NAMELIST and internal files [5.1]

Namelist input/output is now permitted to/from internal files.

8.13 Variables permitted in NAMELIST

All variables except for assumed-size arrays are now permitted to appear in a namelist group [6.0 for allocatable and pointer, 5.3.1 for the rest]. Note that an allocatable variable that appears in a namelist group must be allocated, and a pointer variable that appears in a namelist group must be associated, when a `READ` or `WRITE` statement with that namelist is executed. Also, if a variable is polymorphic or has an ultimate component that is allocatable or a pointer, it is only permitted in a namelist when it will be processed by defined input/output (see below).

8.14 Recursive input/output [5.2]

Input/output to internal files is now permitted while input/output to another internal file or an external file is in progress. This occurs when a function in an input/output list executes an internal file `READ` or `WRITE` statement.

Input/output to an external file while external file input/output is already in progress remains prohibited, except for the case of nested data transfer (see "Defined input/output").

8.15 Asynchronous input/output

8.15.1 Basic syntax [5.1]

Asynchronous input/output syntax is accepted; this consists of the `ASYNCHRONOUS=` specifier on `OPEN`, `READ`, `WRITE` and `INQUIRE`, the `ID=` specifier on `READ`, `WRITE` and `INQUIRE`, and the `PENDING=` specifier on `INQUIRE`.

Except for the `INQUIRE` statement, the `ASYNCHRONOUS=` specifier takes a scalar default character expression; this must evaluate either to `'YES'` or `'NO'`, treating lowercase the same as uppercase. In the `READ` and `WRITE` statements, this character expression must be a constant expression, and the statement must refer to an external file whose connection allows asynchronous input/output; if `ASYNCHRONOUS='YES'` is specified, the data transfer may occur asynchronously. In the `OPEN` statement, this specifier determines whether asynchronous data transfer is allowed for that file: the default setting is `'NO'`. For the `INQUIRE` statement, the `ASYNCHRONOUS=` specifier takes a scalar default character variable, and

sets it to 'YES' if the file is currently connected for asynchronous input/output, 'NO' if the current connection does not allow asynchronous input/output and 'UNKNOWN' if the file is not connected.

For the READ and WRITE statements, the ID= specifier takes a scalar integer variable. This specifier is only permitted if ASYNCHRONOUS='YES' also appears. The integer variable is assigned the "identifier" of the asynchronous data transfer that the READ or WRITE initiates; this value can be used in INQUIRE and WAIT statements to track the progress of the asynchronous data transfer.

For the INQUIRE statement, the ID= specifier takes a scalar integer expression whose value must be that returned from ID= on a READ or WRITE statement for that file, and is only permitted in conjunction with the PENDING= specifier. The PENDING= specifier takes a scalar default logical variable and sets it to .TRUE. if the specified asynchronous data transfer is still underway and to .FALSE. if it has completed. If PENDING= is used without ID=, the enquiry is about all outstanding asynchronous data transfer on that file.

After initiating an asynchronous data transfer, the variables affected must not be referenced or defined until after the transfer is known to have finished. For an asynchronous WRITE of a local variable, this means not returning from the procedure until after ensuring the transfer is complete. An asynchronous data transfer is known to have been finished if there is a subsequent synchronous data transfer, an INQUIRE statement which returns .FALSE. for PENDING=, or a WAIT statement has been executed; in each case, for that file.

8.15.2 Basic Example

The following example uses two buffers, buf1 and buf2, alternately reading into one while processing the other, while there is still data in the file to process (each dataset being followed by a single logical value which indicates whether more is to come).

```
REAL :: buf1(n,m),buf2(n,m)
LOGICAL more,stillwaiting
READ (unit) buf1
DO
  READ (unit) more ! Note: synchronous
  IF (more) READ (unit,ASYNCHRONOUS='YES') buf2
  CALL process(buf1)
  IF (.NOT.more) EXIT
  READ (unit) more ! Note: synchronous
  IF (more) READ (unit,ASYNCHRONOUS='YES') buf1
  CALL process(buf2)
  IF (.NOT.more) EXIT
END DO
```

Note that the synchronous READ statements automatically "wait" for any outstanding asynchronous data transfer to complete before reading the logical value; this ensures that the dataset will have finished being read into its buffer and is safe to process.

8.15.3 The ASYNCHRONOUS attribute [5.2]

A READ or WRITE statement with ASYNCHRONOUS='YES' automatically gives the ASYNCHRONOUS attribute to any variable that appears in its input/output list, in a SIZE= specifier, or which is part of a namelist specified by NML=. This is adequate for asynchronous data transfers that initiate and complete within a single procedure. However, it is inadequate for transfers to/from module variables or dummy arguments if the procedure returns while the transfer is still underway.

The ASYNCHRONOUS attribute may be explicitly specified in a type declaration statement or in an ASYNCHRONOUS statement. The latter has the syntax

ASYNCHRONOUS [::] *variable-name* [, *variable-name*]...

If a variable with the ASYNCHRONOUS attribute is a dummy array and is not an assumed-shape array or array pointer, any associated actual argument cannot be an array section, an assumed-shape array or array pointer. Furthermore,

if a dummy argument has the ASYNCHRONOUS attribute the procedure must have an explicit interface. Both of these restrictions apply whether the attribute was given explicitly or implicitly.

8.15.4 The WAIT statement [5.2]

The WAIT statement provides the ability to wait for an asynchronous data transfer to finish without performing any other input/output operation. It has the form

WAIT (*wait-spec* [, *wait-spec*]...)

where *wait-spec* is one of the following:

UNIT = *file-unit-number*
END = *label*
EOR = *label*
ERR = *label*
ID = *scalar-integer-variable*
IOMSG = *scalar-default-character-variable*
IOSTAT = *scalar-integer-variable*

The UNIT= specifier must appear, but the 'UNIT =' may be omitted if it is the first specifier in the list. The ID= specifier takes a scalar integer expression whose value must be that returned from ID= on a READ or WRITE statement for the file; if the ID= specifier does not appear the WAIT statement refers to all pending asynchronous data transfer for that file.

On completion of execution of the WAIT statement the specified asynchronous data transfers have been completed. If the specified file is not open for asynchronous input/output or is not connected, the WAIT statement has no effect (it is not an error unless the ID= specifier was used with an invalid value).

Here is an example of using the WAIT statement.

```
REAL array(1000,1000,10),xferid(10)
! Start reading each segment of the array
DO i=1,10
  READ (unit,id=xfer(i)) array(:,:,i)
END DO
...
! Now process each segment of the array
DO i=1,10
  WAIT (unit,id=xfer(i))
  CALL process(array(:,:,i)
END DO
```

8.15.5 Execution Semantics

At this time all actual input/output operations remain synchronous, as allowed by the standard.

8.16 Scale factor followed by repeat count [5.1]

The comma that was previously required between a scale factor (*n*P) and a repeat count (e.g. the '3' in 3E12.2), is now optional. This trivial extension was part of Fortran 66 that was removed in Fortran 77, and reinstated in Fortran 2003.

8.17 FLUSH statement [5.2]

Execution of a FLUSH statement causes the data written to a specified file to be made available to other processes, or causes data placed in that file by another process to become available to the program. The syntax of the FLUSH

statement is similar to the BACKSPACE, ENDFILE and REWIND statements, being one of the two possibilities

FLUSH *file-unit-number*
FLUSH (*flush-spec* [, *flush-spec*]...)

where *file-unit-number* is the logical unit number (a scalar integer expression), and *flush-spec* is one of the following:

UNIT = *file-unit-number*
IOSTAT = *scalar-integer-variable*
IOMSG = *scalar-default-character-variable*
ERR = *label*

The UNIT= specifier must appear, but the 'UNIT =' may be omitted if it is the first *flush-spec* in the list.

Here is an example of the use of a FLUSH statement.

```
WRITE (pipe) my_data
FLUSH (pipe)
```

8.18 Defined input/output [6.2]

The generic identifiers READ(FORMATTED), READ(UNFORMATTED), WRITE(FORMATTED) and WRITE(UNFORMATTED) provide the ability to replace normal input/output processing for an item of derived type. In the case of formatted input/output, the replacement will occur for list-directed formatting, namelist formatting, and for an explicit format with the DT edit descriptor: it does not affect any other edit descriptors in an explicit format. Using defined input/output, it is possible to perform input/output on derived types containing pointer components and allocatable components, since the user-defined procedure will be handling it.

Here is a type definition with defined input/output procedures.

```
TYPE tree
  TYPE(tree_node),POINTER :: first
CONTAINS
  PROCEDURE :: fmtread=>tree_fmtread
  PROCEDURE :: fmtwrite=>tree_fmtwrite
  GENERIC,PUBLIC :: READ(formatted)=>fmtread, WRITE(formatted)=>fmtwrite
END TYPE
```

Given the above type definition, whenever a TYPE(tree) object is an effective item in a formatted input/output list, the module procedure tree_fmtread will be called (in a READ statement) or the module procedure tree_fmtwrite will be called (in a WRITE statement) to perform the input or output of that object. Note that a generic interface block may also be used to declare procedures for defined input/output; this is the only option for sequence or BIND(C) types but is not recommended for extensible types.

The procedures associated with each input/output generic identifier must have the same characteristics as the ones listed below, where *type-declaration* is CLASS(*derived-type-spec*) for an extensible type and TYPE(*derived-type-spec*) for a sequence or BIND(C) type. Note that if the derived type has any length type parameters, they **must** be "assumed" (specified as '*').

```
SUBROUTINE formatted_read(var,unit,iotype,vlist,iostat,iomsg)
type-declaration,INTENT(INOUT) ::  var
INTEGER,INTENT(IN) ::  unit
CHARACTER(*),INTENT(IN) ::  iotype
INTEGER,INTENT(IN) ::  vlist(:)
INTEGER,INTENT(OUT) ::  iostat
CHARACTER(*),INTENT(INOUT) ::  iomsg

SUBROUTINE unformatted_read(var,unit,iostat,iomsg)
type-declaration,INTENT(INOUT) ::  var
```

```
INTEGER,INTENT(IN) ::  unit
INTEGER,INTENT(OUT) ::  iostat
CHARACTER(*),INTENT(INOUT) ::  iomsg

SUBROUTINE formatted_write(var,unit,iotype,vlist,iostat,iomsg)
type-declaration,INTENT(IN) ::  var
INTEGER,INTENT(IN) ::  unit
CHARACTER(*),INTENT(IN) ::  iotype
INTEGER,INTENT(IN) ::  vlist(:)
INTEGER,INTENT(OUT) ::  iostat
CHARACTER(*),INTENT(INOUT) ::  iomsg

SUBROUTINE unformatted_write(var,unit,iostat,iomsg)
type-declaration,INTENT(IN) ::  var
INTEGER,INTENT(IN) ::  unit
INTEGER,INTENT(OUT) ::  iostat
CHARACTER(*),INTENT(INOUT) ::  iomsg
```

In each procedure, `unit` is either a normal unit number if the parent input/output statement used a normal unit number, a negative number if the parent input/output statement is for an internal file, or a processor-dependent number (which might be negative) for the '`*`' unit. The `iostat` argument **must** be assigned a value before returning from the defined input/output procedure: either zero to indicate success, the negative number `IOSTAT_EOR` (from the intrinsic module `ISO_FORTRAN_ENV`) to signal an end-of-record condition, the negative number `IOSTAT_END` to signal an end-of-file condition, or a positive number to indicate an error condition. The `iomsg` argument **must** be left alone if no error occurred, and must be assigned an explanatory message if `iostat` is set to a nonzero value.

For the formatted input/output procedures, the `iotype` argument will be set to '`LISTDIRECTED`' if list-directed formatting is being done, '`NAMELIST`' if namelist formatting is being done, and '`DT`' concatenated with the *character-literal* if the `DT` edit descriptor is being processed. The `vlist` argument contains the list of values in the `DT` edit descriptor if present, and is otherwise a zero-sized array. Note that the syntax of the `DT` edit descriptor is:

`DT` [*character-literal*] [`(` *value* [`,` *value*]... `)`]

where blanks are insignificant, *character-literal* is a default character literal constant with no kind parameter, and each *value* is an optionally signed integer literal constant with no kind parameter. For example, 'DT', 'DT"z8,i4,e10.2"', 'DT(100,-3,+4,666)' and 'DT"silly example"(0)' are all syntactically correct `DT` edit descriptors: it is up to the user-defined procedure to interpret what they might mean.

During execution of a defined input/output procedure, there must be no input/output for an external unit (other than for the `unit` argument), but input/output for internal files is permitted. No file positioning commands are permitted. For unformatted input/output, all input/output occurs within the current record, no matter how many separate data transfer statements are executed by the procedure; that is, file positioning both before and after "nested" data transfer is suppressed. For formatted input/output, this effect is approximately equivalent to the nested data transfer statements being considered to be nonadvancing; explicit record termination (using the slash (`/`) edit descriptor, or transmission of newline characters to a stream file) is effective, and record termination may be performed by a nested list-directed or namelist input/output statement.

If `unit` is associated with an external file (i.e. non-negative, or equal to one of the constants `ERROR_UNIT`, `INPUT_UNIT` or `OUTPUT_UNIT` from the intrinsic module `ISO_FORTRAN_ENV`), the current settings for the pad mode, sign mode, etc., can be discovered by using `INQUIRE` with `PAD=`, `SIGN=`, etc. on the `unit` argument. If `unit` is negative (associated with an internal file), `INQUIRE` will raise the error condition `IOSTAT_INQUIRE_INTERNAL_UNIT`.

Finally, defined input/output is not compatible with asynchronous input/output; all input/output statements involved with defined input/output must be synchronous.

9 Miscellaneous Fortran 2003 Features

9.1 Abstract interfaces and the PROCEDURE statement [5.1]

Abstract interfaces have been added, together with the procedure declaration statement. An abstract interface is defined in an interface block that has the `ABSTRACT` keyword, i.e.

```
ABSTRACT INTERFACE
```

Each interface body in an abstract interface block defines an abstract interface instead of declaring a procedure. The name of an abstract interface can be used in the procedure declaration statement to declare a specifc procedure with that interface, e.g.

```
PROCEDURE(aname) :: spec1, spec2
```

declares `SPEC1` and `SPEC2` to be procedures with the interface (i.e. type, arguments, etc.) defined by the abstract interface `ANAME`.

The procedure declaration statement can also be used with the name of any procedure that has an explicit interface, e.g.

```
PROCEDURE(x) y
```

declares `Y` to have the same interface as `X`. Also, procedures with implicit interfaces can be declared by using `PROCEDURE` with a type specification instead of a name, or by omitting the name altogether.

The following attributes can be declared at the same time on the procedure declaration statement: `BIND(C...)`, `INTENT(`*intent*`)`, `OPTIONAL`, `POINTER`, `PRIVATE`, `PUBLIC`, `SAVE`. For example,

```
PROCEDURE(aname),PRIVATE :: spec3
```

Note that `POINTER` declares a procedure pointer (see next section), and that `INTENT` and `SAVE` are only allowed for procedure pointers not for ordinary procedures. The NAG Fortran Compiler also allows the `PROTECTED` attribute to be specified on the procedure declaration statement: this is an extension to the published Fortran 2003 standard.

9.2 Named procedure pointers [5.2]

A procedure pointer is a procedure with the `POINTER` attribute; it may be a named pointer or a structure component (the latter are described elsewhere). The usual way of declaring a procedure pointer is with the procedure declaration statement, by including the `POINTER` clause in that statement: for example,

```
PROCEDURE(aname),POINTER :: p => NULL()
```

declares `P` to be a procedure pointer with the interface `ANAME`, and initialises it to be a disassociated pointer.

A named procedure pointer may also be declared by specifying the `POINTER` attribute in addition to its normal procedure declaration: for example, a function declared by a type declaration statement will be a function pointer if the `POINTER` attribute is included in the type declaration:

```
REAL, EXTERNAL, POINTER :: funptr
```

The POINTER statement can also be used to declare a procedure pointer, either in conjunction with an interface block, an `EXTERNAL` statement, or a type declaration statement, for example:

```
INTERFACE
  SUBROUTINE sub(a,b)
```

```
    REAL,INTENT(INOUT) :: a,b
  END SUBROUTINE
END INTERFACE
POINTER sub
```

Procedure pointers may also be stored in derived types as procedure pointer components. The syntax and effects are slightly different, making them act like "object-bound procedures", and as such are described in the object-oriented programming section.

9.3 Intrinsic modules [4.x]

The Fortran 2003 standard classifies modules as either intrinsic or non-intrinsic. A non-intrinsic module is the normal kind of module (i.e. user-defined); an intrinsic module is one that is provided as an intrinsic part of the Fortran compiler.

There are five **standard** modules in Fortran 2003: IEEE_ARITHMETIC, IEEE_EXCEPTIONS, IEEE_FEATURES, ISO_C_BINDING and ISO_FORTRAN_ENV.

A program is permitted to have a non-intrinsic module with the same name as that of an intrinsic module: to this end, the `USE` statement has been extended: '`USE,INTRINSIC ::`' specifies that an intrinsic module is required, whereas '`USE,NON_INTRINSIC ::`' specifies that a non-intrinsic module is required. If these are not used, the compiler will select an intrinsic module only if no user-defined module is found. For example,

```
USE,INTRINSIC :: iso_fortran_env
```

uses the standard intrinsic module `ISO_FORTRAN_ENV`, whereas

```
USE,NON_INTRINSIC :: iso_fortran_env
```

uses a user-defined module with that name. Note that the double-colon '`::`' is required if either specifier is used.

9.4 Renaming user-defined operators on the USE statement [5.2]

It is now possible to rename a user-defined operator on the USE statement, similarly to how named entities can be renamed. For example,

```
USE my_module, OPERATOR(.localid.)=>OPERATOR(.remotename.)
```

would import everything from `MY_MODULE`, but the `.REMOTENAME.` operator would have its name changed to `.LOCALID.`.

Note that this is only available for user-defined operator names; the intrinsic operators `.AND.` et al cannot have their names changed in this way, nor can `ASSIGNMENT(=)` be renamed. The local name must be an operator if and only if the remote (module entity) name is an operator: that is, both of

```
USE my_module, something=>OPERATOR(.anything.)
USE my_module, OPERATOR(.something.)=>anything
```

are invalid (a syntax error will be produced).

9.5 The ISO_FORTRAN_ENV module [5.1]

The standard intrinsic module `ISO_FORTRAN_ENV` is now available. It contains the following default `INTEGER` named constants.

`CHARACTER_STORAGE_SIZE`
: size of a character storage unit in bits.

`ERROR_UNIT`
: logical unit number for error reporting ("stderr").

`FILE_STORAGE_SIZE`
: size of the file storage unit used by `RECL=` in bits.

`INPUT_UNIT`
: default ('`*`') unit number for `READ`.

`IOSTAT_END`
: `IOSTAT=` return value for end-of-file.

`IOSTAT_EOR`
: `IOSTAT=` return value for end-of-record.

`NUMERIC_STORAGE_SIZE`
: size of a numeric storage unit in bits.

`OUTPUT_UNIT`
: unit used by `PRINT`, the same as the '`*`' unit for `WRITE`.

9.6 The IMPORT statement [5.1]

The `IMPORT` statement has been added. This has the syntax

`IMPORT` [[`::`] *name* [`,` *name*]...]

and is only allowed in an interface body, where it imports the named entities from the host scoping unit (normally, these entities cannot be accessed from an interface body). If no names are specified, normal host association rules are in effect for this interface body.

The `IMPORT` statement must follow any `USE` statements and precede all other declarations, in particular, `IMPLICIT` and `PARAMETER` statements. Anything imported with `IMPORT` must have been declared prior to the interface body.

9.7 Length of names and statements

Names are now ([4.x]) permitted to be 63 characters long (instead of 31), and statements are now ([5.2]) permitted to have 255 continuation lines (instead of 39).

9.8 Array constructor syntax enhancements

Square brackets (`[ ]`) can now ([5.1]) be used in place of the parenthesis-slash pairs (`(/ /)`) for array constructors. This allows expressions to be more readable when array constructors are being mixed with ordinary parentheses.

```
RESHAPE((/(i/2.0,i=1,100)/),(/2,3/))  ! Old way
RESHAPE([(i/2.0,i=1,100)],[2,3])      ! New way
```

Array constructors may now ([5.2]) begin with a type specification followed by a double colon (`::`); this makes zero-sized constructors easy (and eliminates potential ambiguity with character length), and also provides assignment conversions thus eliminating the need to pad all character strings to the same length.

```
[ Logical :: ]                            ! Zero-sized logical array
[ Double Precision :: 17.5, 0, 0.1d0 ]    ! Conversions
[ Character(200) :: 'Alf', 'Bernadette' ] ! Padded to length 200
```

9.9 Structure constructor syntax enhancements [5.3]

There are three enhancements that have been made to structure constructors in Fortran 2003:

1. component names can be used as keywords, the same way that dummy argument names can be used as argument keywords;
2. values can be omitted for components that have default initialisation; and
3. type names can be the same as generic function names, and references are resolved by choosing a suitable function (if the syntax matches the function's argument list) and treating as a structure constructor only if no function matches the actual arguments.

A fourth enhancement is made in the Fortran 2008 standard: a value can be omitted for a component that is allocatable.

This makes structure constructors more like built-in generic functions that can be overridden when necessary. Here is an example showing all three enhancements.

```
TYPE quaternion
  REAL x=0,ix=0,jx=0,kx=0
END TYPE
...
INTERFACE quaternion
  MODULE PROCEDURE quat_from_complex
END INTERFACE
...
TYPE(quaternion) FUNCTION quat_from_complex(c) RESULT(r)
  COMPLEX c
  r%x = REAL(c)
  r%y = AIMAG(c)
  r%z = 0
  r%a = 0
END FUNCTION
...
COMPLEX c
TYPE(quaternion) q
q = quaternion(3.14159265)  ! Structure constructor, value (~pi,0,0,0).
q = quaternion(jx=1)        ! Structure constructor, value (0,0,1,0).
q = quaternion(c)           ! "Constructor" function quat_from_complex.
```

Also, if the type is an extended type an ancestor component name can be used to provide a value for all those inherited components at once.

These extensions mean that even if a type has a private component, you can use the structure constructor if

- the component is allocatable (it will be unallocated),
- the component is default-initialised (it will have the default value), or
- the component is inherited and you use an ancestor component name to provde a value for it and the other components inherited from that ancestor.

9.10 Deferred character length [5.2]

The length of a character pointer or allocatable variable can now be declared to be deferred, by specifying the length as a colon: for example,

```
CHARACTER(LEN=:),POINTER :: ch
```

The length of a deferred-length pointer (or allocatable variable) is determined when it is allocated (see next section) or pointer-associated; for example

```
CHARACTER,TARGET :: t1*3,t2*27
CHARACTER(:),POINTER :: p
p => t1
PRINT *,LEN(p)
p => t2
PRINT *,LEN(p)
```

will first print 3 and then 27. It is not permitted to ask for the `LEN` of a disassociated pointer that has deferred length.

Note that deferred length is most useful in conjunction with the new features of typed allocation, sourced allocation, scalar allocatables and automatic reallocation.

9.11 The ERRMSG= specifier [5.1]

The `ALLOCATE` and `DEALLOCATE` statements now accept the `ERRMSG=` specifier. This specifier takes a scalar default character variable, which in the event of an allocation or deallocation error being detected will be assigned an explanatory message. If no error occurs the variable is left unchanged. Note that this is useless unless the `STAT=` specifier is also used, as otherwise the program will be terminated on error anyway.

For example,

```
ALLOCATE(w(n),STAT=ierror,ERRMSG=message)
IF (ierror/=0) THEN
  PRINT *,'Error allocating W: ',TRIM(message)
  RETURN
END IF
```

9.12 Intrinsic functions in constant expressions [5.2 partial; 5.3 complete]

It is now allowed to use any intrinsic function with constant arguments in a constant expression. (In Fortran 95 real and complex intrinsic functions were not allowed.) For example,

```
MODULE m
  REAL,PARAMETER :: e = EXP(1.0)
END
```

9.13 Specification functions can be recursive [6.2]

A function that is used in a specification expression is now permitted to be recursive (defined with the `RECURSIVE` attribute). For example

```
PURE INTEGER FUNCTION factorial(n) RESULT(r)
  INTEGER,INTENT(IN) :: n
  IF (n>1) THEN
    r = n*factorial(n-1)
  ELSE
    r = 1
  END IF
END FUNCTION
```

can now be used in a specification expression. Note that a specification function must not invoke the procedure that invoked it.

9.14 Access to the command line [5.1]

The intrinsic procedures COMMAND_ARGUMENT_COUNT, GET_COMMAND and GET_COMMAND_ARGUMENT have been added. These duplicate functionality previously only available via the procedures IARGC and GETARG from the F90_UNIX_ENV module.

```
INTEGER FUNCTION command_argument_count()
```

Returns the number of command-line arguments. Unlike IARGC in the F90_UNIX_ENV module, this returns 0 even if the command name cannot be retrieved.

```
SUBROUTINE get_command(command,length,status)
  CHARACTER(*),INTENT(OUT),OPTIONAL :: command
  INTEGER,INTENT(OUT),OPTIONAL :: length,status
```

Accesses the command line which invoked the program. This is formed by concatenating the command name and the arguments separated by blanks. This might differ from the command the user actually typed, and should be avoided (use GET_COMMAND_ARGUMENT instead).

If COMMAND is present, it receives the command (blank-padded or truncated as appropriate). If LENGTH is present, it receives the length of the command. If STATUS is present, it is set to -1 if COMMAND is too short to hold the whole command, a positive number if the command cannot be retrieved, and zero otherwise.

```
SUBROUTINE get_command_argument(number,value,length,status)
  INTEGER,INTENT(IN) :: number
  CHARACTER(*),INTENT(OUT),OPTIONAL :: value
  INTEGER,INTENT(OUT),OPTIONAL :: length,status
```

Accesses command-line argument number NUMBER, where argument zero is the program name. If VALUE is present, it receives the argument text (blank-padded or truncated as appropriate if the length of the argument differs from that of VALUE). If LENGTH is present, it receives the length of the argument. If STATUS is present, it is set to zero for success, -1 if VALUE is too short, and a positive number if an error occurs.

Note that it is an error for NUMBER to be less than zero or greater than the number of arguments (returned by COMMAND_ARGUMENT_COUNT).

9.15 Access to environment variables [5.1]

The intrinsic procedure GET_ENVIRONMENT_VARIABLE has been added. This duplicates the functionality previously only available via the procedure GETENV in the F90_UNIX_ENV module.

```
  SUBROUTINE get_environment_variable(name,value,length,status,trim_name)
    CHARACTER(*),INTENT(IN) :: name
    CHARACTER(*),INTENT(OUT),OPTIONAL :: value
    INTEGER,INTENT(OUT),OPTIONAL :: length,status
    LOGICAL,INTENT(IN),OPTIONAL :: trim_name
  END
```

Accesses the environment variable named by NAME; trailing blanks in NAME are ignored unless TRIM_NAME is present with the value .FALSE.. If VALUE is present, it receives the text value of the variable (blank-padded or truncated as appropriate if the length of the value differs from that of VALUE). If LENGTH is present, it receives the length of the value. If STATUS is present, it is assigned the value 1 if the environment variable does not exist, -1 if VALUE is too short, and zero for success. Other positive values might be assigned for unusual error conditions.

9.16 Character kind selection [5.1]

The intrinsic function SELECTED_CHAR_KIND has been added. At this time the only character set supported is 'ASCII'.

9.17 Argument passing relaxation [5.1]

A `CHARACTER` scalar actual argument may now be passed to a routine which expects to receive a `CHARACTER` array, provided the array is explicit-shape or assumed-size (i.e. not assumed-shape, allocatable, or pointer). This is useful for C interoperability.

9.18 The MAXLOC and MINLOC intrinsic functions [5.1]

The `MAXLOC` and `MINLOC` intrinsic functions now return zeroes for empty set locations, as required by Fortran 2003 (Fortran 95 left this result processor-dependent).

9.19 The VALUE attribute [4.x]

The VALUE attribute specifies that an argument should be passed by value.

9.19.1 Syntax

The `VALUE` attribute may be specified by the `VALUE` statement or with the `VALUE` keyword in a type declaration statement.

The syntax of the `VALUE` statement is:

`VALUE` [`::`] *name* [`,` *name*] ...

The `VALUE` attribute may only be specified for a scalar dummy argument; if the dummy argument is of type `CHARACTER`, its character length must be constant and equal to one.

Procedures with a `VALUE` dummy argument must have an explicit interface.

9.19.2 Semantics

A dummy argument with the `VALUE` attribute is "passed by value"; this means that a local copy is made of the argument on entry to the routine and so modifications to the dummy argument do not affect the associated actual argument and vice versa.

A `VALUE` dummy argument may be `INTENT(IN)` but cannot be `INTENT(INOUT)` or `INTENT(OUT)`.

9.19.3 Example

```
PROGRAM value_example
  INTEGER :: i = 3
  CALL s(i)
  PRINT *,i  ! This will print the value 3
CONTAINS
  SUBROUTINE s(j)
    INTEGER,VALUE :: j
    j = j + 1  ! This changes the local J without affecting the actual argument
    PRINT *,j  ! This will print the value 4
  END SUBROUTINE
END
```

This example is not intended to be particularly useful, just to illustrate the functionality.

9.20 The VOLATILE attribute [5.0]

This is a horrible attribute which specifies that a variable can be modified by means outside of Fortran. Its semantics are basically the same as that of the C '`volatile`' type qualifier; essentially it disables optimisation for access to that variable.

9.21 Enhanced complex constants [5.2]

The real or imaginary part may now be a named constant, it is not limited to being a literal constant. For example:

```
REAL,PARAMETER :: minusone = -1.0
COMPLEX,PARAMETER :: c = (0,minusone)
```

This is not particularly useful, since the same effect can be achieved by using the `CMPLX` intrinsic function.

9.22 The `ASSOCIATE` construct [5.2]

The `ASSOCIATE` construct establishes a temporary association between the "associate names" and the specified variables or values, during execution of a block. Its syntax is

`ASSOCIATE (` *association* [`,` *association*]... `)`
block
`END ASSOCIATE`

where *block* is a sequence of executable statements and constructs, and *association* is one of

name `=>` *expression*
name `=>` *variable*
name

The last of those is short for '*name* `=>` *name*'. The scope of each "associate name" is the *block* of the `ASSOCIATE` construct. An associate name is never allocatable or a pointer, but otherwise has the same attributes as the *variable* or *expression* (and it has the `TARGET` attribute if the *variable* or *expression* is a pointer). If it is being associated with an expression, the expression is evaluated on execution of the `ASSOCIATE` statement and its value does not change during execution of the *block* — in this case, the associate name is not permitted to appear on the left-hand-side of an assignment or any other context which might change its value. If it is being associated with a variable, the associate name can be treated as a variable.

The type of the associate name is that of the expression or variable with which it is associated. For example, in

```
ASSOCIATE(zoom=>NINT(SQRT(a+b)), alt=>state%mapval(:,i)%altitude)
  alt%x = alt%x*zoom
  alt%y = alt%y*zoom
END ASSOCIATE
```

`ALT` is associated with a variable and therefore can be modified whereas `ZOOM` cannot. The expression for `ZOOM` is of type `INTEGER` and therefore `ZOOM` is also of type `INTEGER`.

9.23 Binary, octal and hexadecimal constants [5.2]

In Fortran 95 these were restricted to `DATA` statements, but in Fortran 2003 these are now allowed to be arguments of the intrinsic functions `CMPLX`, `DBLE`, `INT` and `REAL`. The interpretation is processor-dependent, but the intent is that this specifies the internal representation of the complex or real value. The NAG Fortran compiler requires these constants to have the correct length for the specified kind of complex or real, viz 32 or 64 bits as appropriate.

For example, on a machine where default REAL is IEEE single precision,

```
REAL(z"41280000")
```

has the value `10.5`.

9.24 Character sets [5.1; 5.3]

The support for multiple character sets, especially for Unicode (ISO 10646) has been improved.

The default character set is now required to include lowercase letters and all the 7-bit ASCII printable characters.

The `ENCODING=` specifer for the `OPEN` and `INQUIRE` statements is described in the input/output section.

A new intrinsic function `SELECTED_CHAR_KIND(NAME)` has been added: this returns the character kind for the named character set, or -1 if there is no kind for that character set. Standard character set names are `'DEFAULT'` for the default character kind, `'ASCII'` for the 7-bit ASCII character set and `'ISO_10646'` for the UCS-4 (32-bit Unicode) character set. The name is not case-sensitive. Note that although the method of requesting UCS-4 characters is standardised, the compiler is not required to support them (in which case -1 will be returned); the NAG Fortran Compiler supports UCS-4 in release 5.3 (as well as UCS-2 and JIS X 0213).

Assignment of a character value of one kind to a character value of a different kind is permitted if each kind is one of default character, ASCII character, or UCS-4 character. Assignment to and from a UCS-4 character variable preserves the original value.

Internal file input/output to variables of UCS-4 character kind is allowed (if the kind exists), including numeric conversions (e.g. the E edit descriptor), and conversions from/to default character and ASCII character. Similarly, writing default character, ASCII character and UCS-4 character values to a UTF-8 file and reading them back is permitted and preserves the value.

Finally, the intrinsic function `IACHAR` (for converting characters to the ASCII character set) accepts characters of any kind (in Fortran 95 it only accepted default kind).

9.25 Intrinsic function changes for 64-bit machines [5.2]

Especially to support machines with greater than 32-bit address spaces, but with 32-bit default integers, several intrinsic functions now all have an optional `KIND` argument at the end of the argument list, to specify the kind of integer they return. The functions are: `COUNT`, `INDEX`, `LBOUND`, `LEN`, `LEN_TRIM`, `SCAN`, `SHAPE`, `SIZE`, `UBOUND` and `VERIFY`.

9.26 Miscellaneous intrinsic procedure changes [5.2]

The intrinsic subroutine `DATE_AND_TIME` no longer requires the three character arguments (`DATE`, `TIME` and `ZONE`) to have a minimum length: if the actual argument is too small, it merely truncates the value assigned.

The intrinsic functions `IACHAR` and `ICHAR` now accept an optional `KIND` argument to specify the kind of integer to which to convert the character value. This serves no useful purpose since there are no character sets with characters bigger than 32 bits.

The intrinsic functions `MAX`, `MAXLOC`, `MAXVAL`, `MIN`, `MINLOC` and `MINVAL` all now accept character values; the comparison used is the native (`.LT.`) one, not the ASCII (`LLT`) one.

The intrinsic subroutine `SYSTEM_CLOCK` now accepts a `COUNT_RATE` argument of type real; this is to handle systems whose clock ticks are not an integral divisor of 1 second.

10 References

The Fortran 2003 standard, IS 1539-1:2004(E), is available from ISO as well as from many national standards bodies. A number of books describing the new standard are available; the recommended reference book is "Modern Fortran Explained" by Metcalf, Reid & Cohen, Oxford University Press, 2011 (ISBN 978-0-19-960141-7).