ISO/IEC JTC1/SC22/WG5 N1828

The new features of Fortran 2008

John Reid, JKR Associates, UK

May 6, 2010

Abstract

The aim of this paper is to summarize the new features of the draft Fortran 2008 standard (ISO/IEC 2010). We take as our starting point Fortran 2003 (ISO/IEC 2004).

An official extension for enhanced module facilities has been published as a Type 2 Technical Report (ISO/IEC 2005) and WG5 was committed to include this in Fortran 2008.

For an informal description of Fortran 2003 and enhanced module facilities, see Metcalf, Reid, and Cohen (2004).

The major extension consists of coarrays for parallel computing. Since the author has already summarized coarrays in another WG5 paper (Reid 2010), we refer the reader to this for details.

This article is not an official document and has not been approved by PL22.3 (formerly J3) or WG5.

Contents

1 Introduction

Fortran is a computer language for scientific and technical programming that is tailored for efficient run-time execution on a wide variety of processors. It was first standardized in 1966 and the standard has since been revised four times (1978, 1991, 1997, 2004). The revisions alternated between being minor (1978 and 1997) and major (1991 and 2004). WG5 decided in 2004 that the fifth revision should be minor, to follow the pattern and allow time for vendors to implement Fortran 2003 and users to learn to use it.

The plan adopted in 2004 (see N1590[1]) involved the preliminary choice of significant features in 2005 and the final choice in 2006. When it came to the crunch, WG5 left some items for J3 to add if it had time. Reductions were made at the 2007 meeting and a further reduction was made at the February 2008 meeting. Further technical changes were made at the November 2008 meeting.

The full document, N1826, is about to be submitted for international ballot as a Final Draft International Standard (FDIS) and is expected to be published later this year. Only extremely minor changes, such as the correction of typos, are permitted after such a ballot. This article provides an informal description of the features. There are plans for further interoperability and coarray features to be defined in Technical Reports.

We use the convention of indicating the optional arguments of an intrinsic procedure by enclosing them in square brackets in the argument list. We also use square brackets for other optional syntax elements.

2 Submodules

The module facilities of Fortran 2003, while adequate for programs of modest size, have shortcomings for very large programs. They all arise from the fact that, although modules are an aid to modularization of the program, they are themselves difficult to modularize.

As a module grows larger, the only way to modularize it is to break it into several modules. This exposes the internal structure, raising the potential for unnecessary global name clashes and giving the user of the module access to what ought to be private data and/or procedures. Worse, if the subfeatures of the module are interconnected, they must remain together in a single module, however large.

Another significant shortcoming is that if a change is made to the code inside a module procedure, even a private one, typical use of `make` or a similar tool results in the recompilation of every file which accesses the module, directly or indirectly.

The solution is to allow a module procedure to have its interface defined in a module while its body is defined in a separate program unit called a submodule. A change in a submodule cannot

[1]We refer to ISO/IEC JTC1/SC22/WG5 documents by their 'N numbers'. They are all available from `ftp://ftp.nag.co.uk/sc22wg5/`

alter an interface in the module, and so does not cause the recompilation of program units that use the module.

A submodule has access via host association to entities in the module, and may have entities of its own in addition to providing implementations of module procedures.

Here is a simple example:

```
module points
   type :: point
      real :: x,y
   end type point
   interface
      real function point_dist (a,b)
         import :: point
         type(point), intent(in) :: a,b
      end function point_dist
   end interface
end module points
submodule (points) points_a
contains
   real function point_dist (a,b)
      type(point), intent(in) :: a,b
      point_dist = sqrt((a%x-b%x)**2+(a%y-b%y)**2)
   end function point_dist
end submodule points_a
```

The interface specified in the submodule must be exactly the same as that specified in the interface block. There is also a syntax that avoids the redeclaration altogether:

```
submodule (points) points_a
contains
   module procedure point_dist
      point_dist = sqrt((a%x-b%x)**2+(a%y-b%y)**2)
   end procedure point_dist
end submodule points_a
```

Submodules are themselves permitted to have submodules, which is useful for very large programs. If a change is made to a submodule, only it and its descendants will need recompilation.

A submodule of a submodule identifies its parent in its `submodule` statement by the combination of the name of its ancestor module and the name of its parent, for example, `points:points_a` for a submodule of the above submodule. This allows two submodules to have the same name if they are descendants of different modules.

The submodule feature was defined by a Technical Report (ISO/IEC 2005), with a promise that its features would be included in Fortran 2008, apart from the correction of any defects. It offers two further advantages:

- Defining the interfaces of procedures in a module and implementing them in submodules allows one to publish the details of the interfaces while withholding publication of the submodules and their trade secrets.
- Submodules allow one to package the procedures of a module as a library from which the system need only incorporate procedures that the program references, rather than incorporating the entire module.

The feature is described more fully in Chapter 13 of Metcalf, Reid, and Cohen (2004).

3 Coarrays

Coarrays provide a simple extension to Fortran for parallel programming on distributed-memory and shared-memory architectures. The program is treated as if it were replicated a fixed number of times and each replication is called an image. An additional set of subscripts provides access from any image to data on another image. Care has been taken to allow compilers to optimize both execution on an image and communication between images.

For coarrays, a summary is already available in N1824, so we do not need to describe this here.

4 Performance enhancements

4.1 do concurrent

The `do concurrent` form of the `do` loop allows the programmer to state that there are no data dependencies between the iterations of a `do` loop and hence enables optimizations such as vectorization, loop unrolling, and multi-threading. It standarizes different directives that have been recognized by different compilers for a very long time, with different exact meanings. The construct is intended for optimizations within a single image, so no image control statements (for synchronization between images) are permitted within it.

The `do` statement itself becomes

`do [,] concurrent` *forall-header*

It uses some of the `forall` syntax and the resulting `do` construct has some similarities with the `forall` construct, but there are important differences:

- the `forall` is essentially an array assignment statement, behaving as if all the right-hand side expressions were all evaluated before any assignments are made, and
- there are less restrictions on the kind of statements that may appear in a `do concurrent` construct.

Here is a simple example

```
        do concurrent (i=1:m)
           a(k+i) = a(k+i) + factor*a(l+i)
        end do
```

where the programmer knows that there is no overlap between the range of values accessed and the range of values altered.

The full set of restrictions is

- no `return` statement appears
- no branch in the construct has a target outside it (e.g. `exit` is not permitted)
- all procedures referenced are pure
- there are no references to the procedures `ieee_get_flag`, `ieee_set_halting_mode`, and `ieee_get_halting_mode`,
- A variable that is referenced in an iteration was either
  - previously defined during the iteration or
  - is neither defined nor becomes undefined during any other iteration.
- A pointer that is referenced in an iteration was previously pointer associated during the iteration, or does not have its pointer association changed during any iteration.
- An allocatable object that is allocated in more than one iteration shall be subsequently deallocated during the same iteration.
- An object that is allocated or deallocated in only one iteration shall not be deallocated, allocated, referenced, defined, or become undefined in a different iteration.
- An input/output statement shall not write data to a file record or position in one iteration and read from the same record or position in a different iteration.
- No image control statements appear.

A volatile variable is permitted, but is likely to be disastrous for optimization.

A variable that is defined or becomes undefined by more than one iteration becomes undefined when the construct terminates. A pointer that has its pointer association changed in more than one iteration has an association status of undefined when the construct terminates. Records written by output statements in the loop range to a sequential access file appear in the file in an indeterminate order.

4.2 Contiguous attribute

Certain optimizations are possible if the compiler knows that an array occupies a contiguous memory block, which is the case for an explicit-shape, assumed-size, or allocatable array and some sections of such an array. Many unnecessary operations happen because compilers cannot determine that a data item will always occupy contiguous memory. For example, contiguous memory is needed to pass an array to a procedure with an implicit interface. Thus, the mere use of pointers often results in substantially suboptimal code. Similar optimization problems exist for assumed-shape dummy arguments.

The contiguous attribute may be declared for a pointer or assumed-shape array, for example,

```
    real, pointer, contiguous :: ptr(:)
    integer, contiguous, dimension(:,:) :: ary
```

It can also be specified by a `contiguous` statement:

`contiguous [::]` *object-name-list*

where the other attributes are specified elsewhere.

The target of a pointer with the contiguous attribute must be contiguous. The actual argument corresponding to a contiguous assumed-shape array must be contiguous. The actual argument corresponding to a contiguous pointer array must have the contiguous attribute (so that a simpler descriptor can be employed).

Other objects that are contiguous:

- a non-pointer whole array that is not assumed-shape,
- an assumed-shape array that is argument associated with an array that is contiguous,
- an array allocated by an `allocate` statement,
- a pointer associated with a contiguous target, or
- a nonzero-sized array section provided that
  - its base object is contiguous,
  - it does not have a vector subscript,
  - the elements of the section, in array element order, are a subset of the base object elements that are consecutive in array element order,
  - if the array is of type character and a substring-range appears, the substring-range specifies all of the characters of the parent-string,
  - only its final *part-ref* has nonzero rank, and
  - it is not the real or imaginary part of an array of type `complex`.

An object is not contiguous if it is an array subobject and

- the object has two or more elements,
- the elements of the object in array element order are not consecutive in the elements of the base object,
- the object is not of type character with length zero, and
- the object is not of a derived type that has no ultimate components other than zero-sized arrays and characters with length zero.

Whether or not any other object is contiguous is processor dependent.

The contiguity of an array can be tested with the inquiry function `is_contiguous(array)`. It returns a default logical scalar with the value true if `array` is contiguous and false otherwise. If `array` is a pointer, it must be associated with a target.

Arrays in C are always contiguous, so `c_loc` was not available in Fortran 2003 for a pointer array. This restriction has been removed for cases where the target is contiguous.

4.3 Simply contiguous arrays

The concept of a *simply contiguous* array has been introduced for an array that is contiguous and satisfies additional rules that allow the compiler to determine that it is always contiguous. A section subscript list is simply contiguous if

- it has no vector subscripts,
- it has no strides,
- all but the last section triplet are colons, and
- any subscript follows all section triplets.

An array designator is simply contiguous if it is

- the name of an array with the contiguous attribute,
- the name of an array that is not a pointer or of assumed shape,
- a structure component whose final part name is an array that has the contiguous attribute or is not a pointer, or
- an array section
  - that does not select a real or imaginary part,
  - that has no substring range,
  - whose final *part-ref* has nonzero rank,
  - whose rightmost *part-name* has the contiguous attribute or is neither a pointer nor of assumed shape, and
  - that either does not have a section subscript list or has one that is simply contiguous.

An array is simply contiguous if and only if it is a simply contiguous array designator or a reference to a function that returns a pointer with the contiguous attribute.

Fortran 2003 allows a pointer assignment to associate a pointer of rank greater than one with a rank-one target:

```
        matrix(1:n,1:n) => base(:)
```

This is extended to simply contiguous targets of rank greater than one, for example,

```
        matrix(1:n,1:n) => base(:,:,i:j,2)
```

associates the elements of `matrix` in array element order with the elements of `base(:,:,i:j,2)` in array element order.

5 Data declaration

5.1 Maximum rank

The maximum rank has been increased to 15. In the case of a coarray, the limit of 15 applies to the sum of the rank and corank.

5.2 Long integers

The processor is required to support at least one kind of integer with a range of 18 decimal digits, for example

```
integer,parameter :: long = selected_int_kind(18)
integer(long) :: la, ll
```

This will normally be supported with 64-bit integers and is needed to ensure portability of software designed to run on machines with very large memories (now increasingly common).

5.3 Allocatable components of recursive type

A recursive type is permitted to be based on allocatable components. Here is a simple example of a type that holds a stack

```
type entry
   real :: value
   integer :: index
   type(entry), allocatable :: next
end type entry
```

Here is how to add a new entry at the top of the stack:

```
type (entry), allocatable :: top
top = entry ( new_value, new_index, top )
```

Alternative code is as follows:

```
type (entry), allocatable :: top, temp
temp = entry ( new_value, new_index, temp )
call move_alloc(top,temp%next)
call move_alloc(temp,top)
```

which avoids the possiblity of a deep copy being made into a temporary variable followed by another deep copy from it. It is reasonable to expect the compiler, when presented with the first version to perform the equivalent of the second, but it cannot be guaranteed. Similar considerations apply to the removal of the top entry by the code

```
top = top%entry
```

or

```
call move_alloc(top%next,temp)
call move_alloc(temp,top)
```

The usual efficiencies associated with allocatables are available: contiguous arrays, no aliasing (unless given the `target` attribute), and no memory leaks.

5.4 Implied-shape array

An implied-shape array is a named constant that is declared with each upper bound given as an asterisk. It takes its shape from its constant expression[2], for example,

```
integer, parameter :: order(0:*) = [0, 1, 2, 3]
```

5.5 Pointer initialization

A pointer may be initially associated with a target with the `save` attribute:

```
type (entry), target, save :: bottom
type (entry), pointer ::  top => bottom
```

and a pointer component may be default initialized as associated with a target.

5.6 Data statement restrictions lifted

Subscripts and nested implied-do limits in a `data` statement can be any constant expression instead of being limited to combinations of constants, implied-do variables, and intrinsic operations.

5.7 Kind of a forall index

The kind of a `forall` index may be specified in the header:

```
forall ( integer(long) :: i = 1:very_large, j = 1:2 )
```

Both `i` and `j` are of type `integer(long)` inside the `forall` construct.

5.8 Type statement for intrinsic types

The `type` statement may be used for declaring entities of intrinsic type, for example,

```
type(integer) :: i,j
type(complex(kind(0.0d0))) :: x,y
```

The interpretation is exactly the same as it would have been without the keyword `type` and the parentheses.

5.9 Declaring type-bound procedures

Multiple type-bound procedures can be declared in a single `procedure` statement.

[2]Constant expression were known as "initialization" expressions in Fortran 2003.

6 Data usage

6.1 Omitting an allocatable component in a structure constructor

It is permitted to omit an allocatable component in a structure constructor. It is given the allocation status of unallocated.

6.2 Copying the bounds of a source array in an allocate statement

In a Fortran 2003 `allocate` statement, if it is required to copy the bounds of an array, they have to be specified explicitly, for example,

```
allocate ( a(lbound(b,1):ubound(b,1)))
allocate ( a(lbound(b,1):ubound(b,1)), source=b )
```

In Fortran 2008, the bounds may taken from an array given by `mold=` or `source=`:

```
allocate ( a, mold=b )   ! Bounds copied
allocate ( a, source=b ) ! Bounds and value copied
```

6.3 Allocating a polymorphic variable

An `allocate` statement can give a polymorphic variable the shape and type of another variable without copying the value. This is done with `mold=` replacing `source=`. For example,

```
allocate (poly, mold=t2)
```

allocates the polymorphic variable to have the type and shape of `t2`.

Intrinsic assignment to an allocatable polymorphic variable is allowed. The variable must be type compatible with the expression and of the same rank. If it is allocated but the dynamic type differs from that of the expression, it is deallocated. If it is not allocated or becomes deallocated, it is allocated with the dynamic type of the expression.

6.4 Accessing real and imaginary parts

The real and imaginary parts of a complex entity can be accessed independently with a component-like syntax using the names `re` and `im`. For example,

```
complex impedance, x(n), y(n)
impedance%re = 1.0
x%im = 2.0*y%im
```

The new syntax allows the parts to be used as variables rather than only within expressions and makes array sections such as `x%im` and `y%im` available.

6.5 Pointer functions

A reference to a pointer function is treated as a variable and is permitted in any variable-definition context. For example, this function might calculate where to store values depending on a key

```
function storage(key) result(loc)
integer, intent(in) :: key
real, pointer :: loc
loc=>...
end function
```

which would allow a value to be set thus:

```
storage(5)=0.5
```

6.6 Elemental dummy argument restrictions lifted

There are no longer any additional restrictions for elemental procedures on the appearance of dummy arguments in specification expressions.

7 Input/Output

7.1 Finding a unit when opening a file

In an `open` statement, `newunit=` automatically selects a unit number that does not interfere with other unit numbers selected by the program, including preconnected files. For example,

```
integer factor
open (newunit = factor, file = 'factor', status = 'old')
```

which assigns a suitable value to the integer `factor`.

7.2 g0 edit descriptor

The `g0` edit descriptor specifies that the processor should automatically choose a suitable field width. For real and complex data, it follows the rules of `es`*w*.*d*`e`*e* format with the values of *w*, *d*, and *e* chosen by the processor. For integers, it behaves as `i0`. For logicals, it behaves as `l1`. For characters, it behaves as `a`.

7.3 Unlimited format item

A list of edit descriptors in parentheses may be preceded by an asterisk, which has the effect of repeating the list indefinitely, that is, as if it were replaced by a very large integer. For example,

```
     write( 10, '( "iarray =", *( i0, :, ","))') iarray
```

produces a single record with a header and a comma separated list of integer values. Note that this feature can be used with `g0` format to process I/O lists with various types (or derived types) present.

7.4 Recursive input/output

It is useful to be able to perform I/O in a subprogram invoked during the processing of an I/O statement (e.g., for tracing and diagnostic purposes). This is now permitted for an external unit that is distinct from that of an I/O statement that is in execution (that is, any in the current call chain).

8 Execution control

8.1 The block construct

The `block` construct allows entities to be declared and given the scope of the block, for example,

```
     block
        integer :: i
        real :: a(n)
        do i = 1,n
           a(i) = i
        end do
           :
     end block
```

ensures that the `do` index `i` and the automatic array `a` are separate from variables with the same names that are outside the block. Of course, the `block` construct must be properly nested with other constructs.

8.2 Exit statement

In Fortran 2008, the `exit` statement may be employed in almost any construct. This will avoid the need for `go to` statements or extra tests for some algorithms that cannot be expressed in Fortran 2003 without them. Here is an example using the new `block` construct (Section 8.1)

```
        outer: block
           do i = 1, num_in_set
             if ( x == a(i) ) exit outer
           end do
           call r
        end block outer
```

Here, no action is needed if `x` is equal to an element of `a`; otherwise, `r` is called.

The exceptional case is an `exit` statement that belongs to a `do concurrent` (Section 4.1) or `critical` construct (a coarray feature) or an outer construct. In all these cases, a transfer out of the construct is inappropriate and is not permitted.

8.3 Stop code

The optional stop code on a `stop` statement is limited in Fortran 2003 to a string of one to five digits or a character constant. This has been extended to any `integer` or `character` constant expression, available as ever 'in a processor-dependent manner'. In addition, it is recommended that

- it is made available by formatted output on the unit `error_unit` of the intrinsic module `iso_Fortran_env` and
- that if it is an `integer` and the processor supports command line execution (see Section 9.9), it is made available as the exit status value returned by `execute_command_line`.

9 Intrinsic procedures and modules

9.1 Bit processsing

9.1.1 Bit sequence comparison

Bit sequences are compared from left to right, one bit at a time, until unequal bits are found or all bits have been compared and found to be equal. If unequal bits are found, the sequence with zero in the unequal position is considered to be less than the sequence with one in the unequal position. When bit sequences of unequal length are compared, the shorter sequence is considered to be extended by padding with zero bits on the left. The following elemental functions have been added. `i` and j are each of type integer or a binary, octal, or hexadecimal constant.

`bge (i,j)` returns the default `logical` value true if and only if `i` is bitwise greater than or equal to `j`.

`bgt (i,j)` returns the default `logical` value true if and only if `i` is bitwise greater than `j`.

`ble (i,j)` returns the default `logical` value true if and only if `i` is bitwise less than or equal to `j`.

`blt (i,j)` returns the default `logical` value true if and only if `i` is bitwise less than `j`.

9.1.2 Combined shifting

The following elemental functions have been added for combined shifting. `i` and j are each of type `integer` or a binary, octal, or hexadecimal constant. At least one must be `integer`. If they are both `integer`, they must have the same kind. Let `bsize` be the `bit_size` of this kind of integer. The result is of type `integer` and of this kind. `shift` is of type integer with a value in the range 0, 1, 2, ..., `bsize`.

`dshiftl (i,j,shift)` returns the rightmost `bsize-shift` bits of `i` followed by the leftmost `shift` bits of `j`.

`dshiftr (i,j,shift)` returns the rightmost `shift` bits of `i` followed by the leftmost `bsize-shift` bits of `j`.

9.1.3 Counting bits

The following elemental functions have been added for counting bits. `i` is an `integer` and the result is a default `integer`.

`leadz(i)` returns the number of leading zero bits in `i`.

`popcnt(i)` returns the number of one bits in `i`.

`poppar(i)` returns the value 1 if `popcnt(i)` is odd and the value 0 otherwise.

`trailz(i)` returns the number of trailing zero bits in `i`.

9.1.4 Masking bits

The following elemental functions have been added for masking bits.

`maskl(i[,kind])` returns an integer whose leftmost `i` bits are 1 and the rest are zero.

`maskr(i[,kind])` returns an integer whose rightmost `i` bits are 1 and the rest are zero.

`i` is an `integer`.

`kind` is a scalar integer constant expression.

The result is of type `integer` and kind `kind` if `kind` is present and of type default `integer` otherwise.

9.1.5 Shifting bits

The following elemental functions have been added for shifting bits.

`shifta(i,shift)` has the effect of shifting the bits of `i` to the right by `shift` places and replicating the leftmost bit `shift` times in the vacated positions.

`shiftl(i,shift)` has the effect of shifting the bits of `i` to the left by `shift` places and changing the rightmost `shift` bits to zero. It is the same as `ishft(i,shift)`.

`shiftr(i,shift)` has the effect of shifting the bits of `i` to the right by `shift` places and changing the leftmost `shift` bits to zero. It is the same as `ishft(i,-shift)`.

`i` is an `integer`.

`shift` is a nonnegative `integer`.

The result is of type `integer` with the same kind as `i`.

9.1.6 Merging bits

The following elemental function has been added for merging bits.

`merge_bits(i,j,mask)` returns an `integer` that merges the bits of `i` and `j` under the control of `mask`. One of `i` and `j` must be of type `integer`. If both are of type `integer`, they must be of the same kind. One may be a binary, octal, or hexadecimal constant. `mask` is of type `integer` and of the same kind as `i` or `j`, or is a binary, octal, or hexadecimal constant. A bit of the result is the corresponding bit of `i` if the corresponding bit of `mask` is 1 and is the corresponding bit of `j` otherwise.

9.1.7 Bit transformational functions

The following transformational functions have been added for bit operations.

`iall(array,dim,[,mask])` or `iall(array[,mask])` performs bitwise and operations.

`iany(array,dim,[,mask])` or `iany(array[,mask])` performs bitwise or operations.

`iparity(array,dim,[,mask])` or `iparity(array[,mask])` performs bitwise exclusive or operations.

`array` is an `integer` array.

`dim` is an `integer` scalar.

`mask` conforms with `array` and is of type `logical`.

These functions are modelled on the transformational function `sum`, but use the operators of the functions `iand`, `ior`, and `ieor` instead of `+`. The result has the type and kind of `array` and the shape is determined from the shape of `array` and the value of `dim` or its absence, just as for `sum`. The operators are applied to all the elements of `array` to yield a scalar or to all the elements of each rank-one section that spans dimension `dim` to yield a result of rank reduced by one.

9.2 Storage size

`storage_size(a,[kind])` is an inquiry function that returns an integer of kind `kind` if `kind` is present or of default kind otherwise. It returns the storage size in bits of a scalar of the dynamic type and kind of `a`, which may be a scalar or an array of any type.

9.3 Taking radix into account when selecting a real kind

An additional optional argument `radix` has been added to `selected_real_kind` at the end of the argument list. It is an `integer` scalar and its presence results in the search being limited to a particular radix.

9.4 Extensions to trigonometric and hyperbolic intrinsic functions

The intrinsic functions `acos`, `asin`, `atan`, `cosh`, `sinh`, `tan`, and `tanh` may have complex arguments. The inverse functions `acosh(x)`, `asinh(x)`, and `atanh(x)` have been added. The intrinsic function `atan2` may be accessed by the name `atan`.

9.5 Bessel functions

The following elemental functions have been added

`bessel_j0 (x)` returns the Bessel function of the first kind and order zero for a real value `x`.

`bessel_j1 (x)` returns the Bessel function of the first kind and order one for a real value `x`.

`bessel_jn (n,x)` returns the Bessel function of the first kind and order `n` for a real value `x`. The argument `n` must be an integer with a nonnegative value.

`bessel_y0 (x)` returns the Bessel function of the second kind and order zero for a real value `x` that is positive.

`bessel_y1 (x)` returns the Bessel function of the second kind and order one for a real value `x` that is positive.

`bessel_yn (n,x)` returns the Bessel function of the second kind and order `n` for real values `x` that are positive. The argument `n` must be an integer with a nonnegative value.

In addition, the functions `bessel_jn` and `bessel_yn` are overloaded with these transformational functions

`bessel_jn (n1,n2,x)` returns a rank-one array of Bessel functions of the first kind and orders `n1`, `n1+1`, ...`n2` for real values x. The arguments `n1` and `n2` must be integers with nonnegative values.

`bessel_yn (n1,n2,x)` returns a rank-one array of Bessel functions of the second kind and orders `n1`, `n1+1`, ... `n2` for real values x. The arguments `n1` and `n2` must be integers with nonnegative values.

9.6 Error and gamma functions

The following elemental functions have been added

`erf (x)` returns the error function for a real value `x`, that is, $\frac{2}{\sqrt{\pi}}\int_0^{\mathtt{x}} \exp(-t^2)dt$.

`erfc (x)` returns the complementary error function for a real value `x`, that is, `1-erf(x)` $= \frac{2}{\sqrt{\pi}}\int_{\mathtt{x}}^{\infty} \exp(-t^2)dt$.

`erfc_scaled (x)` returns the exponentially-scaled complementary error function for a real value `x`, that is, $\exp(\mathtt{x}^2)\frac{2}{\sqrt{\pi}}\int_{\mathtt{x}}^{\infty} \exp(-t^2)dt$.

`gamma (x)` returns the gamma function for a real value `x`, that is, $\int_0^{\infty} t^{\mathtt{x}-1}\exp(-t)dt$.

`hypot (x,y)` returns the Euclidean distance function $\sqrt{\mathtt{x}^2+\mathtt{y}^2}$ for real values `x` and `y`, without undue overflow or underflow.

`log_gamma (x)` returns the logarithm of the absolute value of the gamma function for a real value `x` that is not a negative integer or zero.

9.7 Euclidean vector norms

The following transformational function has been added.

`norm2(x[,dim])` calculates Euclidean vector norms.

`x` is a `real` array.

`dim` is an `integer` scalar.

This function is modelled on the transformational function `sum`, but without the optional argument `mask` and replacing summation by calculation of the Euclidean vector norm $\sqrt{\sum x_i^2}$. The result has the type and kind of `x` and the shape is determined from the shape of `x` and the value of `dim` or its absence, just as for `sum`. The operator is applied to all the elements of `x` to yield a scalar or to the elements of each rank-one section that spans dimension `dim` to yield a result of rank reduced by one. It is recommended that the result be reasonably accurate even if computing some of the squares of the elements would result in overflow or underflow.

9.8 Parity

The following transformational function has been added.

`parity(mask[,dim])` tests for the number of true values being odd.

`mask` is a `logical` array.

`dim` is an `integer` scalar.

This function is modelled on the transformational function `all`, replacing the test for all values being true with the test for the number of true values being odd. The result has the type and kind of `mask` and the shape is determined from the shape of `mask` and the value of `dim` or its absence, just as for `all`. The operator is applied to all the elements of `mask` to yield a scalar or to the elements of each rank-one section that spans dimension `dim` to yield a result of rank reduced by one.

9.9 Execute command line

`call execute_command_line(command[,wait,exitstat,cmdstat,cmdmsg])` starts execution of another program if the processor supports command line execution.

`command` has intent `in` and is a scalar default `character` holding the command line to be executed.

`wait` has intent `in` and is a scalar default `logical`. If present with the value false, and the processor supports asynchronous execution of the command, the command is executed asynchronously; otherwise it is executed synchronously.

`exitstat` has intent `inout` and is a scalar default `integer`. If the command is executed synchronously, it is assigned the value of the processor-dependent exit status. Otherwise, the value is unchanged.

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

`cmdmsg` has intent `inout` and is a scalar default `character`. If an error condition occurs, it is assigned a processor-dependent explanatory message. Otherwise, it is unchanged.

When the command is executed synchronously, the subroutine returns after the command line has completed execution. Otherwise, it returns without waiting. If an error condition occurs and `cmdstat` is not present, error termination of execution is initiated.

9.10 Location of maximum or minimum value in an array

The intrinsics `maxloc` and `minloc` have had an additional optional argument `back` added at the end of the argument list. The effect of its presence with the value true is that if there is more than one element that satisfies the condition, the last in array element order is taken instead of the first.

9.11 Find location in an array

The following transformational function has been added

`findloc(array,value[,mask,kind,back])` or

`findloc(array,value,dim[,mask,kind,back])` finds the location in `array` of an element with value `value`.

`array`	is an array of intrinsic type.
`value`	is a scalar of a type that may be used for intrinsic comparisons.
`dim`	is a scalar `integer`.
`mask`	conforms with `array` and is of type `logical`.
`kind`	is a scalar constant expression.
`back`	is a scalar `logical`.

This function is modelled on the transformational function `maxloc`, replacing the search for a maximum value with a search for the value `value`. The result has the type and kind of `array` and the shape is determined from the shape of `mask` and the value of `dim` or its absence, just as for `maxloc`. The search is applied to all the elements of `array` to yield a scalar or to the elements of each rank-one section that spans dimension `dim` to yield a result of rank reduced by one.

9.12 String comparison

The string comparison functions `lge`, `lgt`, `lle`, and `llt` are permitted to have arguments of ASCII kind as well as default character kind.

9.13 Constants

The intrinsic module `iso_Fortran_env` contains these new constants:

`character_kinds` is a default `integer` array holding the kind values supported by the processor for variables of type `character`. Its size equals the number of kinds supported.

`int8`, `int16`, `int32`, and `int64` are default `integer` scalars holding the kind values for integers of storage size 8, 16, 32, and 64 bits. If there is no such type, the value is -2 if there is a type of larger storage size or -1 otherwise.

`integer_kinds` is a default `integer` array holding the kind values supported by the processor for variables of type `integer`. Its size equals the number of kinds supported.

`iostat_inquire_internal_unit` is a default `integer` scalar holding the value returned in an `iostat=` specifier in an `inquire` statement if the file unit number identifies an internal unit (which can happen only during user-defined derived-type I/O).

`logical_kinds` is a default `integer` array holding the kind values supported by the processor for variables of type `logical`. Its size equals the number of kinds supported.

`real_kinds` is a default `integer` array holding the kind values supported by the processor for variables of type `real`. Its size equals the number of kinds supported.

`real32`, `real64`, and `real128` are default `integer` scalars holding the kind values for reals of storage size 32, 64, and 128 bits. If there is no such type, the value is -2 if there is a type of larger storage size or -1 otherwise.

9.14 Module procedures

The intrinsic module `iso_Fortran_env` contains these functions:

`compiler_options` is an inquiry function that returns a default `character` scalar that details the options that the compiler was given.

`compiler_version` is an inquiry function that returns a default `character` scalar that details the name and version of the compiler used.

The intrinsic module `iso_C_binding` contains this function:

`C_sizeof(x)` is an inquiry function that returns an `integer`. If `x` is scalar, the result is the value that the companion processor returns as the result of applying the C `sizeof` operator to an object of a type that interoperates with the type and type parameters of `x`. If `x` is an array, it is the result for an element multiplied by the number of elements.

In the intrinsic module `ieee_arithmetic`, an additional optional argument `radix` has been added to `ieee_selected_real_kind` at the end of the argument list. It is an `integer` scalar and its presence results in the search being limited to a particular radix.

10 Programs and procedures

10.1 Save attribute for module and submodule data

A variable, common block, or procedure pointer that is declared in a module or submodule automatically has the `save` attribute.

10.2 Empty contains section

An empty `contains` section is permitted in a procedure or type definition.

10.3 Form of the end statement for an internal or module procedure

The keyword `function` or `subroutine` is not required to appear in an `end` statement for an internal or module procedure.

10.4 Internal procedure as an actual argument or pointer target

An internal procedure may be passed as an actual argument or be the target of a procedure pointer, which permits it to be invoked from outside of its host. This will be very convenient for users of library codes because the internal procedure may have access to any data accessible in the host. We illustrate this with a procedure to calculate $\int_a^b f(x)dx$ with the interface

```
interface
   real function integrate(f, a, b) result(integral)
      interface
         real function f(x) ! Integrand
            real, value :: x
         end function f
      end interface
      real, intent(in) :: a, b ! Bounds
   end function integrate
end interface
```

Here it is being used in a function to calculate $\int_a^b x^n dx$:

```
real function my_integration(n, a, b) result(integral)
! Integrate f(x)=x**n over [a,b]
    integer, intent(in) :: n
    real, intent(in) :: a, b
    integral = integrate(my_f, a, b)
 contains
     real function my_f(x) ! Integrand
```

```
            real, value :: x
            my_f = x**n ! n is taken from the host.
         end function my_f
     end function my_integration
```

10.5 Null pointer or unallocated allocatable as an absent dummy argument

A null pointer or unallocated allocatable that corresponds to an optional non-allocatable non-pointer dummy argument is interpreted as an absent argument.

10.6 Pointer dummy arguments

A pointer dummy argument with intent `in` may be argument associated with a non-pointer actual argument with the `target` attribute. During the execution of the procedure it is pointer associated with the actual argument.

10.7 Generic resolution by pointer/allocatable or data/procedure

A pair of specific procedures in a generic interface are permitted to be distinguishable by virtue of a pointer argument without intent `in` of one corresponding to an allocatable argument of the other or a data argument of one corresponding to a procedure argument of the other.

10.8 Elemental procedures that are not pure

An elemental procedure is not required to be pure. This must be explicitly declared with the prefix `impure` and the effect is that the procedure is called for corresponding array elements in array element order. For example,

```
module PRNG
   integer, save :: seed(4)
contains
   impure elemental subroutine random (x)
      real :: x
         ... ! Code that updates the seed and
         ... ! calculates a pseudo-random value x
   end subroutine random
end module PRNG
```

is a module that calculates pseudo-random numbers, based on a seed that is an integer array of size 4. If `random` is called for an array, the scalar subroutine is called repeatedly, updating the seed each time.

10.9 Entry statement becomes obsolescent

The `entry` statement becomes obsolescent. A procedure with entry points may be replaced by a module with a separate module procedure for each entry and shared code in a private module procedure.

11 Source form

11.1 Semicolon at line start

The restriction of not permitting a line to start with a semicolon has been removed.

12 Acknowledgements

I would like to express thanks to Malcolm Cohen, Aleksandar Donev, Nick Gould, Jim Giles, Alla Gorelik, Jonathan Hogg, Erik Kruyt, Bill Long, Steve Morgan, Dan Nagle, Jane Sleightholme, Van Snyder, and Stan Whitlock for suggesting improvements.

13 References

WG5 (2010). FDIS revision of the Fortran Standard. ISO/IEC/JTC1/SC22/WG5-N1826, see

`ftp://ftp.nag.co.uk/sc22wg5/N1801-N1850/N1826.pdf`

ISO/IEC (2004). ISO/IEC 1539-1:2004(E) Information technology - Programming languages - Fortran - Part 1: Base language. ISO, Geneva.

ISO/IEC (2005). ISO/IEC TR 19767 Information technology - Programming languages - Fortran - Enhanced Module Facilities. ISO, Geneva.

Metcalf, Michael, Reid, John, and Cohen, Malcolm (2004). Fortran 95/2003 explained. Oxford University Press.

Reid (2010). ISO/IEC JTC1/SC22/WG5 N1824. Coarrays in the next Fortran Standard, `ftp://ftp.nag.co.uk/sc22wg5/N1801-N1850/N1824.pdf`