The New Features of Fortran 2008

John Reid

WG5 Convener

JKR Associates

24 Oxford Road

Benson

Oxon

OX10 6LX

UK

j.k.reid@rl.ac.uk

Overview

The aim of this paper is to summarize the new features of the draft Fortran 2008 standard (ISO/IEC 2008). We take as our starting point Fortran 2003 (ISO/IEC 2008). An official extension for enhanced module facilities (ISO/IEC 2005) that has been published as a Type 2 Technical Report (TR), and WG5 is 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 proposed extension consists of coarrays for parallel computing. Since the author has already summarised coarrays in another WG5 paper (Reid 2008), we refer the reader to this for further details.

This is not an official document and has not been approved by either of the Fortran committees WG5 or J3.

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 of alternated between being minor (1978 and 1997) and major (1991 and 2004). WG5 decided in 2004 that the fifth revision 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, available at ftp://ftp.nag.co.uk/sc22wg5) 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. It is this version that will be described here.

The full document, N1723, is currently in the process of balloting as a Committee Draft Standard with a closing date of 31 August 2008. Comments should be submitted through your National Body and should be based on N1723. We hope this article will be helpful, but it is not an official document and has not been approved by WG5 or J3.

The schedule, see N1693, envisages some minor technical changes being made at the WG5 meeting in Tokyo in November 2008 in response to comments in the ballots, but no further technical changes being made thereafter. WG5 therefore decided that the informal name of the language will be Fortran 2008, though subsequent editorial polishing and ISO balloting will mean that it is not published until summer 2010.

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.

Submodules

Submodules allow the modularization of a large module, possible in Fortran 2003 only by breaking the module into several modules, which exposes the internal structure and risks name clashes. Submodules also allow a module to be separated into the part that defines its interface and the rest. The submodules may be changed and recompiled without the recompilation of program units that use it. In this way, time-consuming compilation cascades for very large programs are avoided.

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 is described in Chapter 13 of Metcalf, Reid, and Cohen (2004), so we do not need to describe it here.

Coarrays

Coarrays provide a simple extension to Fortran for parallel programming. 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 provide 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 N1724, so we do not need to describe this here.

Performance enhancements

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 standardizes different directives that have been recognized by different compilers for a very long time, with different exact meanings.

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;
- 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 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 le record or position in one iteration and read from the same record or position in a different iteration.

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.

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 occupies contiguous memory. For example, contiguous memory is needed to pass an array to a procedure with an implicit interface. Certain optimizations are possible if the compiler knows that the memory stride for the left-most subscript is 1, or if a whole array occupies a contiguous memory block. 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 are contiguous:
- a nonpointer 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 non-zero-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 species all of the characters of the parent-string,
  * only its final *part-ref* has non-zero rank,
  * 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.

Contiguity can be tested with the inquiry function is contiguous(a) where a is an array. It returns a default logical scalar with the value true if a is contiguous and false otherwise. If a 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.

Simply contiguous arrays

The concept of a *simply contiguous* array has been introduced for an array that is contiguous and satisfies additional rules that ensure 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,
  * has no substring range,
  * whose final *part-ref* has non-zero rank,
  * whose rightmost *part-name* has the contiguous attribute or is neither a pointer nor of assumed shape, and
  * 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.

Data enhancements

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.

Long integers

The processor is required to support at least one kind of integer with a range of 18 decimals, 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).

Allocatable components

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 possibility 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 and no memory leaks.

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 the initialization expression, for example,

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

Pointer initialization

A pointer may be initially associated with a target:

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

Kind of a forall index

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

```
forall ( integer(long) :: I = 1:huge, j = 1:2 )
```

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 deallocated or becomes deallocated, it is allocated with the dynamic type of the expression.

Accessing data objects

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,

```
impedance%re = 1.0
x%im = 2.0*y%im
```

Pointer functions

A reference to a pointer function is treated 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
```

Input/Output

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 (naiant = factor, file = 'factor', status = 'old')
```

which assigns a suitable value to the integer factor.

g0 edit descriptor

The g0 edit descriptor species that the processor should automatically choose a suitable field width. For real and complex data, it follows the rules of es$w$.de 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.

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.

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 any that of an I/O statement that is in execution (that is, any in the current call chain).

Execution control

The block construct

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

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

ensures that the `do` index `I` is separate from any other variable with the name `I`. Of course, the `block` construct must be properly nested with other constructs.

Exit statement

Some algorithms cannot be expressed in Fortran 2003 without `go to` statements or extra tests, but they could be expressed with `exit` if it could be applied to any labelled construct. Here is an example using the new block construct

```
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 need if `x` is equal to an element of `a`; otherwise, `r` is called.

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 initialization 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, it is made available as the exit status value.

Intrinsic procedures for bit processsing

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. Otherwise, the sequences are considered to be equal. 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 `integer` value true if and only if `I` is bitwise greater than or equal to `j`.
- `bgt (i,j)` returns the default `integer` value true if and only if `I` is bitwise greater than `j`.
- `ble (i,j)` returns the default `integer` value true if and only if `I` is bitwise less than or equal to `j`.
- `blt (i,j)` returns the default `integer` value true if and only if `I` is bitwise less than `j`.

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 this kind. `shift` is of type integer with a value in the range 1, 2, ..., `bsize`.

- `dshiftl (i,j,shift)` has value equal to the rightmost `bsize-shift` bits of `I` followed by the leftmost `shift` bits of `j`.
- `dshiftr (i,j,shift)` has value equal to the rightmost `shift` bits of `I` followed by the leftmost `bsize-shift` bits of `j`.

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`.

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 initialization expression.

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

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.
- `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 non-negative `integer`.

The result is of the same type as `I`.

Merging bits

The following elemental subroutine 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.

Bit transformational functions

The following are 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 iany(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 rank-one sections to yield a result of rank reduced by one.

Intrinsic procedures and modules

Storage size

*   storage size(a,[kind]) is an inquiry function and 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.

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 if present results in the search being limited to a particular radix.

Hyperbolic intrinsic functions

The hyperbolic trigonometric intrinsic functions cosh, sinh, and tanh may have complex arguments. Their inverses acosh(x), asinh(x), and atanh(x) have been added.

Bessel functions

The following are 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.

Tangent function

The intrinsic `atan2` may be accessed by the name `atan`.

Error and gamma functions

The following elemental functions have been added
- `erf` (x) returns the error function for a real value x, that is, $\frac{2}{\sqrt{\pi}} \int_0^x \exp(-t^2) dt$

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

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

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

- `hypot` (x,y) returns the Euclidean distance function $\sqrt{x^2 + y^2}$ 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.

Euclidean vector norms

The following are 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 $\sqrt{\Sigma x_i^2}$. 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 operators are applied to all the elements of x to yield a scalar or to the elements of rank-one sections 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 overow or underflow.

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 operators are applied to all the elements of `mask` to yield a scalar or to the elements of rank-one sections to yield a result of rank reduced by one.

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.

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.

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 initialization 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 rank-one sections to yield a result of rank reduced by one.

Constants

The intrinsic module `iso_Fortran_env` contains these 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.
- `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.
- `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.
- `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, -2 is return if there is a type of larger storage size or -1 otherwise.
- `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, -2 is return if there is a type of larger storage size or -1 otherwise.
- `iostat_inquire_internal_unit` is a default `integer` scalar holding the values returned in an `iostat=` specifier in an inquire statement if the le unit number identifies an internal unit (which can happen only during user-defined derived-type I/O).

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 controlled the translation phase.
- `compiler_version` is an inquiry function that returns a default `character` scalar that details the name and version of program that controlled the translation phase.

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.

Programs and procedures

Empty contains section

An empty `contains` section is permitted.

Internal procedure as an actual argument

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, intent(in) :: 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
```

Generic resolution by pointer or allocatable attribute

A pair of specific procedures in a generic interface are permitted to be distinguishable by virtue of a pointer argument one corresponding to an allocatable argument of the other.

Null pointer as a missing dummy argument

A null pointer that corresponds to an optional dummy argument is interpreted as an absent argument.

Elemental procedures that are not pure

An elemental procedure is not required to be pure. This must be explicitly declared with the prefix impure, for example,

```
impure elemental subroutine accumulate (a,sum)
real, intent(inout) :: a,sum
sum = sum + a
a = sum
end subroutine accumulate
```

may be used to replace the value of each in an array by the accumulated sum of elements in array element order:

```
real a(n), sum
:
sum = 0.0
call accumulate (a,sum)
```

Acknowledgements

I would like to express thanks to Nick Gould, Jonathan Hogg, and Dan Nagle for suggesting changes.

References

ISO/IEC (2008). ISO/IEC JTC1/SC22/WG5 N1723. Committee Draft revision of the Fortran Standard, ftp://ftp.nag.co.uk/sc22wg5/N1701-N1750/N1723.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.