The New Features of Fortran 2003

John Reid
WG5 Convener
JKR Associates
24 Oxford Road
Benson
Oxon
OX10 6LX
UK
j.k.reid@rl.ac.uk

Introduction

The aim of this paper is to summarize the new features of the Fortran 2003 standard (WG5 2004). We take as our starting point Fortran 95 plus the two official extensions (Cohen 2001, Reid 2001) that have been published as Type 2 Technical Reports (TRs). These provide features for

- Allocatable dummy arguments and type components, and
- Support for the five exceptions of the IEEE Floating Point Standard (IEEE 1989) and for other features of this Standard.

There was a firm commitment to include the features of these TRs in Fortran 2003, apart from changes that follow from errors and omissions found during implementation. Therefore, these features are not described here. For an informal description, see chapters 11 and 12 of Metcalf, Reid, and Cohen (2004).

Fortran 2003 is a major extension of Fortran 95. This contrasts with Fortran 95, which was a minor extension of Fortran 90. Beside the two TR items, the major changes concern object orientation and interfacing with C. Allocatable arrays are very important for optimization – after all, good execution speed is Fortran's forte. Exception handling is needed to write robust code. Object orientation provides an effective way to separate programming into independent tasks and to build upon existing codes; we describe these features in Section 2. Interfacing with C is needed to allow programmers ease of access to system routines which are often written in C and to allow C programmers to call efficient Fortran codes; we describe these features in Section 5. There are also many less major enhancements, described in Sections 3 and 4.

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

1 Introduction and overview of the new features

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 revision of 1991 was major and those of 1978 and 1997 were relatively minor. The fourth revision is major and was made following a meeting of ISO/IEC JTC1/SC22/WG5 in 1997 that considered all the requirements of users, as expressed through their national bodies.

The significant enhancements in the 1991 revision were dynamic storage, structures, derived types, pointers, type parameterization, modules, and array language. The main thrust of the 1997 revision was in connection with alignment with HPF (High Performance Fortran).

The major enhancements for this revision are

- Derived type enhancements: parameterized derived types, improved control of accessibility, improved structure constructors, and finalizers.
- Object oriented programming support: type extension and inheritance, polymorphism, dynamic type allocation, and type-bound procedures.
- Data manipulation enhancements: allocatable components, deferred type parameters, VOLATILE attribute, explicit type specification in array constructors and allocate statements, pointer enhancements, extended initialization expressions, and enhanced intrinsic procedures.

- Input/output enhancements: asynchronous transfer, stream access, user specified transfer operations for derived types, user specified control of rounding during format conversions, named constants for preconnected units, the flush statement, regularization of keywords, and access to error messages.
- Procedure pointers.
- Support for the exceptions of the IEEE Floating Point Standard (IEEE 1989).
- Interoperability with the C programming language.
- Support for international usage: access to ISO 10646 4-byte characters and choice of decimal or comma in numeric formatted input/output.
- Enhanced integration with the host operating system: access to command line arguments, environment variables, and processor error messages.

In addition, there are numerous minor enhancements.

Except in extremely minor ways, this revision is upwards compatible with the previous standard, that is, a program that conforms to Fortran 95 will conform to Fortran 2003.

The enhancements are in response to demands from users and will keep Fortran appropriate for the needs of present-day programmers without losing the vast investment in existing programs.

2 Data enhancements and object orientation

2.1 Parameterized derived types

An obvious deficiency of Fortran 95 is that whereas each of the intrinsic types has a kind parameter and character type has a length parameter, it is not possible to define a derived type that is similarly parameterized. This deficiency is remedied with a very flexible facility that allows any number of 'kind' and 'length' parameters. A kind parameter is a constant (fixed at compile time) and may be used for a kind parameter of a component of intrinsic (or derived) type. A length parameter is modelled on the length parameter for type character and may be used for declaring character lengths of character components and bounds of array components. The names of the type parameters are declared on the `TYPE` statement of the type definition, like the dummy arguments of a function or subroutine, and they must be declared as `KIND` or `LEN`. A default value may be given in an initialization expression (known at compile time). Here is an example for a matrix type

```
TYPE matrix(kind,m,n)
      INTEGER, KIND :: kind=KIND(0.0D0)! gives a default value
      INTEGER, LEN :: m,n ! no default value given
      REAL(kind) :: element(m,n)
END TYPE
```

Explicit values for the type parameters are normally specified when an object of the type is declared. For example,

```
TYPE(matrix(KIND(0.0D0),10,20)) :: a
```

declares a double-precision matrix of size 10 by 20. However, for a pointer or allocatable object, a colon may be used for a length parameter to indicate a deferred value:

```
TYPE(matrix(KIND(0.0),:,:)),ALLOCATABLE :: a
```

The actual value is determined when the object is allocated or pointer assigned. For a dummy argument, an asterisk may be used to indicate an assumed value; the actual value is taken from the actual argument. For a kind parameter, the value must be an initialization expression.

The keyword syntax of procedure calls may be used:

```
TYPE(matrix(kind=KIND(0.0),m=10,n=20)) :: a
```

and the same syntax is used for declaring components of another derived type:

```
TYPE double_matrix(kind,m,n)
      INTEGER, KIND :: kind
```

```
      INTEGER, LEN :: m,n
      TYPE(matrix(kind,m,n)) :: a,b
END TYPE
```

If a type parameter has a default value, it may be omitted from the list of parameter values. For enquiries about the values of type parameters, the syntax of component selection is provided:

```
a%kind, a%m
```

Of course, this syntax may not be used to alter the value of a type parameter, say by appearing on the left of an assignment statement. This syntax is also available for enquiring about a type parameter of an object of intrinsic type:

```
LOGICAL :: L
WRITE (*,*) L%KIND ! Same value as KIND(L)
```

2.2 Procedure pointers

A pointer or pointer component may be a procedure pointer. It may have an explicit or implicit interface and its association with a target is as for a dummy procedure, so its interface is not permitted to be generic or elemental. The statement

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

declares `p` to be a procedure pointer that is initially null and has the same interface as the procedure `proc`.

If no suitable procedure is to hand to act as a template, an ‘abstract interface’ may be declared thus

```
ABSTRACT INTERFACE
      REAL FUNCTION f(a,b,c)
            REAL, INTENT(IN) :: a,b,c
      END FUNCTION
END INTERFACE
```

without there being any actual procedure `f`.

As for data pointers, procedure pointers are either uninitialized or initialized to null. The statement

```
PROCEDURE ( ), POINTER :: p
```

declares `p` to be an uninitialized pointer with an implicit interface. It may be associated with a subroutine or a function. The statement

```
PROCEDURE (TYPE(matrix(KIND(0.0D0),m=10,n=20))), POINTER :: p
```

is similar but specifies that the pointer may be associated only with a function whose result is of the given type and type parameters.

A function may have a procedure pointer result.

Pointer assignment takes the same form as for data pointers:

```
p => proc
```

The interfaces must agree in the same way as for actual and dummy procedure arguments. The right-hand side may be a procedure, a procedure pointer, or a reference to a function whose result is a procedure pointer.

Having procedure pointers fills a big hole in the Fortran 95 language. It permits ‘methods’ to be carried along with objects (dynamic binding):

```
TYPE matrix(kind,m,n)
      INTEGER, KIND :: kind
      INTEGER, LEN :: m,n
      REAL(kind) :: element(m,n)
      PROCEDURE (lu), POINTER :: solve
```

```
END TYPE
      :
TYPE(matrix(KIND(0.0D0),m=10,n=20))) :: a
      :
CALL a%solve(....
      :
```

If the method is always the same, a better way to carry it along with an object is through binding it to the type.

2.3 Finalization

A derived type may have 'final' subroutines bound to it. Their purpose is to perform clean-up operations such as the deallocation of the targets of pointer components when an object of the type ceases to exist. Each final subroutine is a module procedure with a single argument of the derived type to which will be passed an object that is about to cease to exist. The usual rules of argument association apply, so the object has the type and kind type parameters of the dummy argument and has the same rank unless the subroutine is elemental. The dummy argument is required not to have `INTENT(OUT)` and its array shape and length type parameters must be assumed.

An example of the syntax for declaring module subroutines to be final is

```
TYPE T
      : ! Component declarations
CONTAINS
      FINAL :: finish1, finish2
END TYPE T
```

A derived type is finalizable if it has any final subroutines or if it has a component that is of a type that is finalizable but is neither a pointer nor allocatable. A nonpointer data object is finalizable if its type is finalizable. When such an object ceases to exist, a finalization subroutine is called for it if there is one with the right kind type parameters and rank; failing this, an elemental one with the right kind type parameters is called. Next, each finalizable component is finalized; if any is an array, each finalizable component of each element is finalized separately. For a nested type, working top-down like this means that the final subroutine has only to concern itself with components that are not finalizable.

2.4 Procedures bound by name to a type

A procedure may be bound to a type and accessed by component selection syntax from a scalar object of the type rather as if it were a procedure pointer component with a fixed target.

An example of the syntax is

```
TYPE T
      : ! Component declarations
CONTAINS
      PROCEDURE :: proc => my_proc
      PROCEDURE :: proc2
END TYPE T
```

which binds `my_proc` with the name `proc` and `proc2` with its own name. Each procedure must be a module procedure or an external procedure with an explicit interface. If `a` is a scalar variable of type `T`, an example of a type-bound call is

```
CALL a%proc(x,y)
```

The `GENERIC` statement provides access to several such procedures by a single generic name:

```
GENERIC :: gen => proc1, proc2, proc3
```

where `proc1`, `proc2`, and `proc3` are the names of specific bindings. The usual rules about disambiguating procedure calls apply to all the procedures accessible through a single generic binding name.

2.5 The PASS attribute

A procedure that is accessed via a component or by being bound by name usually needs to access the scalar object through which it was invoked. By default, it is assumed that it is passed to the first argument. For example, the call

```
CALL a%proc(x,y)
```

would pass `a` to the first argument of the procedure, `x` to the second, and `y` to the third. This requires that the first argument is a scalar of the given type and the procedure is said to have the `PASS` attribute. If this behaviour is not wanted, the `NOPASS` attribute must be specified explicitly:

```
PROCEDURE, NOPASS, POINTER :: p
```

The usual PASS attribute may be explicitly confirmed:

```
PROCEDURE, PASS :: proc2
```

or may be attached to another argument:

```
PROCEDURE, PASS(arg) :: proc3
```

The passed-object dummy argument must not be a pointer, must not be allocatable, and all its length type parameters must be assumed.

2.6 Procedures bound to a type as operators

A procedure may be bound to a type as an operator or a defined assignment. In this case, the procedure is accessible wherever an object of the type is accessible. The syntax is through `GENERIC` statements in the contained part of a type declaration:

```
TYPE matrix(kind,m,n)
      INTEGER, KIND :: kind
      INTEGER, LEN :: m,n
      REAL(kind) :: element(m,n)
CONTAINS
      GENERIC :: OPERATOR(+) => plus1, plus2, plus3
      GENERIC :: ASSIGNMENT(=) => assign1, assign2
      ! plus1 and assign1 are for matrices alone.
      ! The others are for mixtures with other types.
END TYPE
      :
TYPE(matrix(KIND(0.0D0),m=10,n=20))) :: a,b,c
      :
a = b + c ! Invokes plus1, then assign1.
      :
```

One or both of the arguments must be of the type to which the procedure is bound. The usual rules about disambiguating procedure calls apply to all the procedures accessible in a scoping unit through a single operator.

2.7 Type extension

A derived type that does not have the `SEQUENCE` or `BIND(C)` attribute (the `BIND(C)` attribute makes the type interoperable, see Section 5.4) is extensible. For example, the type `matrix` of Section 2.6 may be extended:

```
TYPE, EXTENDS(matrix) :: factored_matrix
      LOGICAL :: factored=.FALSE.
      REAL(matrix%kind) :: factors(matrix%m,matrix%n)
END TYPE
```

Conversely, a derived type that has the `SEQUENCE` or `BIND(C)` attribute is not extensible.

An extended type is extensible, too, so the term 'parent type' is used for the type from which an extension is made. All the type parameters, components, and bound procedures of the parent type are inherited by the extended type and they are known by the same names. For example,

```
TYPE(factored_matrix(kind(0.0),10,10)) :: f
```

declares a real factored 10x10 matrix. The values of its type parameters are given by `f%kind`, `f%m`, and `f%n`. The inherited component may be referenced as `f%element`.

In addition, the extended type has the parameters, components, and bound procedures that are declared in its own definition. Here, we have the additional components `f%factored` and `f%factors`.

The extended type also has a component, called the parent component, whose type and type parameters are those of its parent type and whose name is the name of the parent type. We actually made use of this in the definition of the type `factored_matrix`. The inherited parameters, components, and bound procedures may be accessed as a whole, f%matrix, as well as directly, `f%n` and `f%element`. They may also be accessed individually through the parent as `f%matrix%n` and `f%matrix%element`.

There is an ordering of the nonparent components that is needed in structure constructors and for input/output. It consists of the inherited components in the component order of the parent type, followed by the new components. In our example, it is `element, factored, factors.`

In a structure constructor, values may be given for a parent component or for the inherited components. No component may be explicitly specified more than once and any component that does not have a default value must be specified exactly once.

2.8 Overriding a type-bound procedure

A specific procedure bound by name is permitted to have the name and attributes of a procedure bound to the parent, apart from the type of the `PASS` argument (Section 2.5), if any. It must not be `PRIVATE` if the parent's binding is `PUBLIC`. In this case, it overrides the procedure bound to the parent type.

Such overriding may be prohibited in the parent type:

```
PROCEDURE, NON_OVERRIDABLE :: proc2
```

2.9 Enumerations

An enumeration is a set of integer constants (enumerators) that is appropriate for interoperating with C (Section 5). The kind of the enumerators corresponds to the integer type that C would choose for the same set of constants. Here is an example:

```
ENUM, BIND(C)
      ENUMERATOR :: RED = 4, BLUE = 9
      ENUMERATOR YELLOW
END ENUM
```

This declares an enumerator with constants 4, 9, 10.

If a value is not specified for an enumerator, it is taken as one greater than the previous enumerator or zero if it is the first.

2.10 ASSOCIATE construct

The ASSOCIATE construct associates named entities with expressions or variables during the execution of its block. Here are some simple examples:

```
ASSOCIATE ( Z => EXP(-(X**2+Y**2)) * COS(THETA) )
      PRINT *, A+Z, A-Z
END ASSOCIATE
```

```
ASSOCIATE ( XC => AX%B(I,J)%C, ARRAY => AX%B(I,:)%C )
      XC%DV = XC%DV + PRODUCT(XC%EV(1:N))
      ARRAY = ARRAY + 1.0
END ASSOCIATE
```

Each name is known as an 'associate name'. The association is as for argument association with a dummy argument that does not have the `POINTER` or `ALLOCATABLE` attribute but has the `TARGET` attribute if the variable does. Any expressions in the `ASSOCIATE` statement are evaluated when it is executed and their values are used thereafter. An associated object must not be used in a situation that might lead to its value changing unless it is associated with a variable.

The construct may be nested with other constructs in the usual way.

2.11 Polymorphic entities

A polymorphic entity is declared to be of a certain type by using the `CLASS` keyword in place of the `TYPE` keyword and is able to take this type or any of its extensions during execution. The type at a particular point of the execution is called the 'dynamic type'. The entity must have the pointer or allocatable attribute or be a dummy argument. It gets its dynamic type from allocation, pointer assignment, or argument association.

The feature allows code to be written for objects of a given type and used later for objects of an extended type. An entity is said to be *type compatible* with entities of the same declared type or of any declared type that is an extension of its declared type.

An object may be declared with the `CLASS(*)` specifier and is then 'unlimited polymorphic'. It is not considered to have the same declared type as any other entity, but is type compatible with all entities.

A dummy argument that is neither allocatable nor a pointer is required to be type compatible with its actual argument. This, for example, allows a Fortran 95 procedure to be passed an argument of an extended type; it simply ignores the extensions. A dummy argument that is allocatable or a pointer is required to have the same declared type as its actual argument and if either is polymorphic the other must be too. A dummy argument with the `PASS` attribute (Section 2.5) is required to be polymorphic if its type is extensible.

Derived-type intrinsic assignment is extended to allow the right-hand side (but not the left-hand side) to be polymorphic. The declared types must conform in the usual way, but the right-hand side may have a dynamic type that is an extension of the type of the left-hand side, in which case the components of the left-hand side are copied from the corresponding components of the right-hand side.

For a pointer assignment, the general rule is that the pointer is required to be type compatible with the target and its kind type parameters must have the same values as the corresponding type parameters of the target. If it is polymorphic, it assumes the dynamic type of the target. There is one exception to this rule: the pointer may be of a sequence derived type when the target is unlimited polymorphic and has that derived type as its dynamic type.

Access is permitted directly to type parameters, components, and bound procedures for the declared type only. However, further access is available through the `SELECT TYPE` construct, described in the next section.

The inquiry functions `SAME_TYPE_AS(A,B)` and `EXTENDS_TYPE_OF(A,MOLD)` are available to determine whether `A` and `B` have the same dynamic type and whether the dynamic type of `A` is an extension of that of `MOLD`.

2.12 SELECT TYPE construct

The `SELECT TYPE` construct selects for execution at most one of its constituent blocks, depending on the dynamic type of a variable or an expression, known as the 'selector'. Here is a simple example with a variable as its selector:

```
CLASS (matrix(kind(0.0),10,10)) :: f
      :
SELECT TYPE (ff => f)
      TYPE IS (matrix)
            : ! Block of statements
```

```
      TYPE IS (factored_matrix)
            : ! Block of statements
END SELECT
```

The first block is executed if the dynamic type of `f` is `matrix` and the second block is executed if it is `factored_matrix`. The association of the selector `f` with its associate name `ff` is exactly as in an `ASSOCIATE` construct (Section 2.10). In the second block, we may use ff to access the extensions thus: `ff%factored, ff%factor.`

Here is a fuller example of the `SELECT TYPE` construct:

```
TYPE :: POINT
      REAL :: X, Y
END TYPE POINT
TYPE, EXTENDS(POINT) :: POINT_3D
      REAL :: Z
END TYPE POINT_3D
TYPE, EXTENDS(POINT) :: COLOR_POINT
      INTEGER :: COLOR
END TYPE COLOR_POINT

TYPE(POINT), TARGET :: P
TYPE(POINT_3D), TARGET :: P3
TYPE(COLOR_POINT), TARGET :: C
CLASS(POINT), POINTER :: P_OR_C

P_OR_C => C
SELECT TYPE ( A => P_OR_C )
TYPE IS ( POINT_3D )
      PRINT *, A%X, A%Y, A%Z
CLASS IS ( POINT )
      PRINT *, A%X, A%Y ! This block gets executed
END SELECT
```

Within each block, the associate name has the declared type or class given on the `TYPE IS` or `CLASS IS` statement. The block is chosen as follows:

- If a `TYPE IS` block matches, it is taken;
- otherwise, if a single `CLASS IS` block matches, it is taken;
- otherwise, if several `CLASS IS` blocks match, one must be an extension of all the others and it is taken.

A sequence derived type must not be specified in this way.

There may also be a `CLASS DEFAULT` block. This is selected if no other block is selected; the associate name then has the same declared and dynamic types as the selector.

2.13 Deferred bindings and abstract types

There are situations where it is expected that a procedure bound to a type will be invoked only for extensions of the type that override the procedure. This typically happens when the type-bound procedure does not have a default or natural implementation, but rather only a well-defined purpose and interface. Such behaviour may be ensured by declaring the type as abstract and the binding as deferred. An interface is required for the bound procedure. Here is a simple example, using an abstract interface (Section 2.2):

```
TYPE, ABSTRACT :: FILE_HANDLE
CONTAINS
      PROCEDURE (OPEN_FILE), DEFERRED, PASS :: OPEN
      ...
```

```
END TYPE
ABSTRACT INTERFACE
      SUBROUTINE OPEN_FILE(HANDLE)
            CLASS(FILE_HANDLE),INTENT(INOUT) :: HANDLE
      END SUBROUTINE OPEN_FILE
END INTERFACE
```

In this example, the intention is that extensions of the type would have components that hold data about the file and `OPEN` would be overridden by a procedure that uses these data to open it.

Polymorphic entities may be declared with an abstract type, but it is not permitted to declare, allocate, or construct a nonpolymorphic object of such a type. Also, a deferred binding is permitted only in an abstract type.

3 Miscellaneous enhancements

3.1 Structure constructors

The value list of a structure constructor may use the syntax of an actual argument list with keywords that are component names. Components that were given default values in the type definition may be omitted. This implies that structure constructors can be used for types that have private components, so long as the private components have default values. Of course, no component may be given an explicit value more than once and explicit values override default values. If the type has type parameters, those without default values must be specified:

```
a = matrix(KIND(0.0),m=10,n=20) (element = 0.0)
```

A generic name may be the same as a derived type name, provided it references a function. This has the effect of overriding or overloading the constructor for the type.

3.2 The allocate statement

The allocatable attribute is no longer restricted to arrays and a source variable may be specified to provide values for deferred type parameters (Section 2.1) and an initial value for the object itself. For example,

```
TYPE(matrix(KIND(0.0D0),m=10,n=20)) :: a
TYPE(matrix(KIND(0.0D0),m=:,n=:)),ALLOCATABLE :: b, c
      :
ALLOCATE(b,SOURCE=a)
ALLOCATE(c,SOURCE=a)
```

allocates the scalar objects b and c to be 10 by 20 matrices with the value of a. With `SOURCE` present, the allocate statement allocates just one object. The `SOURCE` expression must be of the type of the object being allocated or an extension of this type and is limited to being a scalar or an array of the same rank as the object being allocated.

Alternatively, the deferred type parameters may be specified by a type declaration within the allocate statement:

```
ALLOCATE ( matrix(KIND(0.0D0),m=10,n=20) :: b,c )
```

If this feature is used in an allocate statement, initialization from a source variable is not available. One or the other must be used if the type has any deferred type parameters. If either is used, each allocatable object in the list must have the same non-deferred type parameters as the source variable or the type declaration.

The allocate statement may also specify the dynamic type of a polymorphic object:

```
CLASS (matrix(kind(0.0),10,10)) :: a,b,c,d
      :
ALLOCATE(factored_matrix(kind(0.0),10,10) :: b,c)
ALLOCATE(d,SOURCE=a) ! d takes its dynamic type from a
```

An ALLOCATE or DEALLOCATE statement may optionally contain an ERRMSG= specifier that identifies a default character scalar variable. If an error occurs during execution of the statement, the processor assigns an explanatory message to the variable. If no such condition occurs, the value of the variable is not changed.

3.3 Assignment to an allocatable array

Assignment to an allocatable variable is treated in the same way as to an allocatable array component (not described in this summary since it is a feature of Technical Report TR 15581, Cohen 2001); that is, the destination variable is allocated to the correct shape if it is unallocated or reallocated to the correct shape if it is allocated with another shape. Thus, instead of having to say:

```
S = SIZE(COMPRESS(B))
IF (ALLOCATED(A)) THEN
      IF(SIZE(A)/=S) DEALLOCATE (A)
END IF
IF (.NOT.ALLOCATED(A)) ALLOCATE (A(S))
A = COMPRESS(B)
```

one may say

```
A = COMPRESS(B)
```

and have the same effect. If the allocateable object has deferred type parameters as in the example of Section 2.1,

```
TYPE(matrix(KIND(0.0),:,:)) , ALLOCATABLE :: A
```

those type parameters are also automatically given the appropriate values.

3.4 Transferring an allocation

The intrinsic subroutine MOVE_ALLOC(FROM,TO) has been introduced to move an allocation from one allocatable object to another. The arguments must both be allocatable and their ranks must be the same. TO must be type compatible (Section 2.11) with FROM and polymorphic if FROM is. Each non-deferred parameter of the declared type of TO must have the same value as the corresponding parameter of FROM. After the call, the FROM argument becomes deallocated.

This will permit the reallocation of an allocatable object with programmer control on how the old data are spread into the new object; here is an example:

```
REAL,ALLOCATABLE :: GRID(:),TEMPGRID(:)
      ...
ALLOCATE(GRID(-N:N) ! initial allocation of GRID
      ...
ALLOCATE(TEMPGRID(-2*N:2*N)) ! allocate bigger grid
TEMPGRID(::2) = GRID ! distribute values to new locations
CALL MOVE_ALLOC(TO=GRID,FROM=TEMPGRID)
```

This therefore provides a reallocation facility that avoids the problem that has beset all previous attempts: deciding how to spread the old data into the new object.

3.5 More control of access from a module

More detailed control of access from a module is possible. The individual components of a derived type may be declared PUBLIC or PRIVATE:

```
TYPE, EXTENDS(person) :: s_person   ! Parent component has the same
                                    ! accessibility as the parent type
      CHARACTER(:), ALLOCATABLE, PUBLIC :: name
      INTEGER, PRIVATE :: age
```

```
END TYPE
```

An entity is permitted to be PUBLIC even if its type is PRIVATE. The accessibility of the components of a type is independent of the accessibility of the type. Thus, it is possible to have a PUBLIC type with a PRIVATE component and a PRIVATE type with a PUBLIC component.

The bindings to a type (Sections 2.4 to 2.6) may be declared PUBLIC or PRIVATE:

```
PROCEDURE, PUBLIC, PASS, POINTER :: p
GENERIC, PUBLIC :: gen => proc1, proc2
GENERIC, PRIVATE :: OPERATOR(+) => plus1, plus2, plus3
```

Note, however, that a final subroutine may not be declared PUBLIC or PRIVATE. It is always available for the finalization of any variable of the type.

The PROTECTED attribute may be applied to a variable or a pointer declared in a module, and specifies that its value or pointer status may be altered only within the module itself. It may be declared with the attribute:

```
REAL, PROTECTED :: a(10)
```

or given it by a PROTECTED statement:

```
PROTECTED :: a, b
```

If any object has the PROTECTED attribute, all of its subobjects have the attribute.

If a pointer has the PROTECTED attribute, its pointer association status is protected, but not the value of its target.

This feature is very useful for constructing reliable software. It parallels INTENT(IN) for a dummy argument. The value is made available, but changing it is not permitted. The PROTECTED attribute may be applied only in the module of original declaration; if a module uses an unprotected variable from another module, it cannot apply the PROTECTED attribute to it.

3.6 Renaming operators on the USE statement

In Fortran 2003, it is permissible to rename operators that are not intrinsic operators:

```
USE MY_MODULE, OPERATOR(.MyAdd.) => OPERATOR(.ADD.)
```

This allows .MyAdd. to denote the operator .ADD. accessed from the module.

3.7 Pointer assignment

Pointer assignment for arrays has been extended to allow lower bounds to be specified:

```
p(0:,0:) => a
```

The lower bounds may be any scalar integer expressions.

Remapping of the elements of a rank-one array is permitted:

```
p(1:m,1:2*m) => a(1:2*m*m)
```

The mapping is in array-element order and the target array must be large enough. The bounds may be any scalar integer expressions.

The limitation to rank-one arrays is because pointer arrays need not occupy contiguous storage:

```
a => b(1:10:2)
```

but all the gaps have the same length in the rank-one case.

Length type parameters of the pointer may be deferred (declared with a colon). Pointer assignment gives these the values of the corresponding parameters of the target. All the pointer's other type parameters must have the same values as the corresponding type parameters of the target.

3.8 Pointer INTENT

INTENT was not permitted to be specified in Fortran 95 for pointer dummy arguments because of the ambiguity of whether it should refer to the pointer association status, the value of the target, or both. `INTENT` is permitted in Fortran 2003; it refers to the pointer association status and has no bearing on the value of the target.

3.9 The VOLATILE attribute

The `VOLATILE` attribute has been introduced for a variable to indicate that its value might change by means not specified in the program, for example, by another program that is executing in parallel. For a pointer, the attribute refers both to the association status and to the target. For an allocatable object, it refers to everything about it. Whether an object has the `VOLATILE` attribute may vary between scoping units. If an object has the `VOLATILE` attribute, all of its subobjects also have the attribute. For a dummy argument, the `VOLATILE` attribute is incompatible with `INTENT(IN)` or the `VALUE` attribute (Section 5.6) and is not permitted.

The effect is that the compiler is required to reference and define the memory that can change by other means rather than rely on values in cache or other temporary memory.

3.10 The IMPORT statement

In Fortran 95, interface bodies ignore their environment, that is, nothing from the host is accessible. For example, a type that is defined in a module is not accessible in an interface body within the module. The `IMPORT` statement has therefore been introduced. It has the syntax

```
IMPORT :: new_type, compute
```

and is allowed only in an interface body. Without an import-name list, it specifies that all entities in the host scoping unit are accessible by host association. With a list, those named are accessible.

3.11 Intrinsic modules

The concept of an intrinsic module was introduced in TR 15580 (Reid 2001). It has been made clear that a non-intrinsic module may have the same name and both may be accessed within a program but not within a single scoping unit.

3.12 Access to the computing environment

A new intrinsic module is `ISO_FORTRAN_ENV`. It contains the following constants

- `INPUT_UNIT, OUTPUT_UNIT,` and `ERROR_UNIT` are default integer scalars holding the unit identified by an asterisk in a `READ` statement, an asterisk in a `WRITE` statement, and used for the purpose of error reporting, respectively.
- `IOSTAT_END` and `IOSTAT_EOR` are default integer scalars holding the values that are assigned to the IOSTAT= variable if an end-of-file or end-of-record condition occurs, respectively.
- `NUMERIC_STORAGE_SIZE, CHARACTER_STORAGE_SIZE,` and `FILE_STORAGE_SIZE` are default integer scalars holding the sizes in bits of a numeric, character, and file storage unit, respectively.

In addition, the following intrinsic procedures have been added. Note that they are ordinary intrinsics and are not part of the module `ISO_FORTRAN_ENV`.

`COMMAND_ARGUMENT_COUNT ()` is an inquiry function that returns the number of command arguments as a default integer scalar.

`CALL GET_COMMAND ([COMMAND,LENGTH,STATUS])` returns the entire command by which the program was invoked in the following `INTENT(OUT)` arguments:

- `COMMAND` (optional) is a default character scalar that is assigned the entire command.
- `LENGTH (optional)` is a default integer scalar that is assigned the significant length (number of characters) of the command.
- `STATUS (optional)` is a default integer scalar that indicates success or failure.

CALL GET_COMMAND_ARGUMENT (NUMBER[,VALUE,LENGTH,STATUS]) returns a command argument.

- NUMBER is a default integer INTENT(IN) scalar that identifies the required command argument. Useful values are those between 0 and COMMAND_ARGUMENT_COUNT().
- VALUE (optional) is a default character INTENT(OUT) scalar that is assigned the value of the command argument.
- LENGTH (optional) is a default integer INTENT(OUT) scalar that is assigned the significant length (number of characters) of the command argument.
- STATUS (optional) is a default integer INTENT(OUT) scalar that indicates success or failure.

CALL GET_ENVIRONMENT_VARIABLE (NAME[,VALUE,LENGTH,STATUS,TRIM_NAME]) obtains the value of an environment variable.

- NAME is a default character INTENT(IN) scalar that identifies the required environment variable. The interpretation of case is processor dependent.
- VALUE (optional) is a default character INTENT(OUT) scalar that is assigned the value of the environment variable.
- LENGTH (optional) is a default integer INTENT(OUT) scalar. If the specified environment variable exists and has a value, LENGTH is set to the length (number of characters) of that value. Otherwise, LENGTH is set to 0.
- STATUS (optional) is a default integer INTENT(OUT) scalar that indicates success or failure.
- TRIM_NAME (optional) is a logical INTENT(IN) scalar that indicates whether trailing blanks in NAME are considered significant.

3.13 Support for international character sets

Fortran 90 introduced the possibility of multi-byte character sets, which provides a foundation for supporting ISO 10646 (2000). This is a standard for 4-byte characters, which is wide enough to support all the world's languages.

A new intrinsic function has been introduced to provide the kind value for a specified character set:

SELECTED_CHAR_KIND(NAME) returns the kind value as a default INTEGER.

- NAME is a scalar of type default character. If it has one of the values DEFAULT, ASCII, and ISO_10646, it specifies the corresponding character set.

The Fortran character set now includes both upper-case and lower-case letters and further printable ASCII characters have been added as special characters:

```
~     Tilde                  \     Backslash
[     Left square bracket    ]     Right square bracket
'     Grave accent           ^     Circumflex accent
{     Left curly bracket     }     Right curly bracket
|     Vertical bar           #     Number sign
@     Commercial at
```

Only square brackets are actually used in the syntax (for array constructors, Section 3.16).

There is a standardized method (UTF-8) of representing 4-byte characters as strings of 1-byte characters in a file. Support for this is provided by

- permitting the specifier ENCODING='UTF-8' on the OPEN statement for a formatted file,
- adding an ENCODING= inquiry to the INQUIRE statement
- allowing any data written to a UFT-8 file to be read back into ISO 10646 character variables,
- allowing any numeric or logical data to be written to a UFT-8 file and read back into variables of the same type and kind,
- allowing any default, ASCII, or ISO 10646 character data to be written to a UFT-8 file and read back into character variables of the same kind,
- permitting an internal file to be of ISO 10646 kind, with the same freedoms for reading and writing as for a UFT-8 file,

- permitting default or ASCII character data to be assigned to ISO 10646 character variables,
- optionally, allow ISO 10646 character data to be assigned to ASCII character variables (OK up to `CHAR(127,SELECTED_CHAR_KIND(ISO_10646))`, otherwise data is lost), and
- adding a `KIND` argument to `ACHAR` and permitting `IACHAR` to accept a character of any kind.

3.14 Lengths of names and statements

Names of length up to 63 characters and statements of up to 256 lines are allowed. The main reason for the longer names and statements is to support the requirements of codes generated automatically.

3.15 Binary, octal and hex constants

A binary, octal or hex constant is permitted as a principal argument in a call of the intrinsic function `INT, REAL, CMPLX,` or `DBLE` (not for an optional argument that specifies the kind).

Examples are

```
INT(O'345'), REAL(Z'1234ABCD')
```

For `INT,` the 'boz' constant is treated as if it were an integer constant of the kind with the largest range supported by the processor.

For the others, it is treated as having the value that a variable of the same type and kind type parameters as the result would have if its value were the bit pattern specified. The interpretation of the value of the bit pattern is processor dependent. If the kind is not specified, it is the default kind.

The advantage of limiting boz constants in this way is that there is no ambiguity in the way they are interpreted. There are vendor extensions that allow them directly in expressions, but the ways that values are interpreted differ.

3.16 Array constructor syntax

Square brackets are permitted as an alternative to `(/` and `/)` for delimiters for array constructors.

An array constructor may include a type specification such as

```
[ CHARACTER(LEN=7) :: 'Takata', 'Tanaka', 'Hayashi' ]
[ matrix(KIND(0.0),m=10,n=20) :: a,b,c]
```

which allows values of length type parameters to be specified. The element values are obtained by the rules of intrinsic assignment, which means that this example is valid despite the varying character lengths. For a derived type with kind type parameters, these must agree. For a derived type with length type parameters, these must agree unless the parameters are specified in the array constructor and the element values can be obtained by the rules of intrinsic assignment.

3.17 Specification and initialization expressions

The rules on what may appear in an initialization expression have been relaxed. Any standard intrinsic procedure is permitted. For details of the new rules, see 7.1.7 of the draft standard (J3 2003). The rules on what may appear in a specification expression have been relaxed, too. For details, see 7.1.6 of the draft standard.

3.18 Complex constants

A named constant may be used to specify the real or imaginary part of a complex constant:

```
REAL, PARAMETER :: pi = 3.1415926535897932384
COMPLEX :: C = (0.0,pi)
```

3.19 Changes to intrinsic functions

To allow for the speed of modern processors, the argument `COUNT_RATE` of the subroutine `SYSTEM_CLOCK` is permitted to be of type `REAL` as well as `INTEGER`.

The intrinsics MAX, MAXLOC, MAXVAL, MIN, MINLOC, and MINVAL have been extended to apply to type CHARACTER as well as REAL and INTEGER.

Minor changes have been made to ATAN2, LOG, and SQRT to take proper account of signed zeros (so that the value is the limit of corresponding values for small arguments of the same sign). If the processor can distinguish between positive and negative real zero and X is negative, ATAN2(-0.0,X) and LOG(CMPLX(X,-0.0)) have the value -π and SQRT(CMPLX(X,-0.0)) has negative imaginary part.

3.20 Controlling IEEE underflow

The IEEE standard specifies that underflow be 'gradual', that is, that under flowed values be represented with the most negative exponent and reduced precision whenever this is possible. Some processors represent all such values as exact zeros ('abrupt' underflow), which may permit faster execution. A facility has been added for finding out which mode is in effect and changing it on systems that permit this.

The following procedures have been added to the intrinsic module IEEE_ARITHMETIC (of TR 15580, Reid 2001):

IEEE_SUPPORT_UNDERFLOW_CONTROL(X) has the value true if the processor supports control of the underflow mode for floating-point calculations with the same type as X, and false otherwise. The argument X may be omitted, in which case the test is for all floating-point calculations.

IEEE_GET_UNDERFLOW_MODE(GRADUAL) is a subroutine that sets GRADUAL, of type default logical, to true if gradual underflow is in effect and false otherwise.

IEEE_SET_UNDERFLOW_MODE(GRADUAL) is a subroutine that sets the underflow mode to gradual underflow if GRADUAL, of type default logical, is true and to abrupt underflow otherwise. It is permitted to be invoked only if IEEE_SUPPORT_UNDERFLOW_CONTROL(X) is true for some X.

3.21 Another IEEE class value

The value IEEE_OTHER_VALUE has been added for the type IEEE_CLASS_TYPE (of TR 15580, Reid 2001), which is used to identify whether a value is a signalling NaN, quiet NaN, -infinity, negative denormalized, etc. IEEE_OTHER_VALUE is needed for implementing the module on systems which basically conform to the IEEE Standard, but do not implement all of it. It might be needed, for example, if an unformatted file were written in a program executing with gradual underflow enabled and read with it disabled.

4 Input/output enhancements

4.1 Derived type input/output

It may be arranged that when a derived-type object is encountered in an input/output list, a Fortran subroutine is called. This reads some data from the file and constructs a value of the derived type or accepts a value of the derived type and writes some data to the file.

For formatted input/output, the DT edit descriptor passes a character string and an integer array to control the action. An example is

```
DT 'linked-list' (10, -4, 2)
```

The character string may be omitted; this case is treated as if a string of length zero had been given. The bracketed list of integers may be omitted, in which case an array of length zero is passed.

Such subroutines may be bound to the type as generic bindings (see Sections 2.4 and 2.6) of the forms

```
GENERIC :: READ(FORMATTED) => r1, r2
GENERIC :: READ(UNFORMATTED) => r3, r4, r5
GENERIC :: WRITE(FORMATTED) => w1
GENERIC :: WRITE(UNFORMATTED) => w2, w3
```

which makes them accessible wherever an object of the type is accessible. An alternative is an interface block such as

```
INTERFACE READ(FORMATTED)
MODULE PROCEDURE r1, r2
END INTERFACE
```

The form of such a subroutine depends on whether it is for formatted or unformatted input or output:

```
SUBROUTINE formatted_io (dtv,unit,iotype,v_list,iostat,iomsg)
SUBROUTINE unformatted_io(dtv,unit, iostat,iomsg)
```

- `dtv` is a scalar of the derived type. It must be polymorphic (so that it can be called for the type or any extension of it). Any length type parameters must be assumed. For output, it is of intent(in) and holds the value to be written. For input, it is of `intent(inout)` and must be altered in accord with the values read.
- `unit` is a scalar of `intent(in)` and type default `integer`. Its value is the unit on which input/output is taking place or negative if on an internal file.
- `iotype` is a scalar of intent(in) and type character(*). Its value is '`LISTDIRECTED`', '`NAMELIST`', or '`DT//string` where string is the character string from the `DT` edit descriptor.
- `v_list` is a rank-one assumed-shape array of `intent(in)` and type default integer. Its value comes from the parenthetical list of the edit descriptor.
- `iostat` is a scalar of `intent(out)` and type default `integer.` If an error condition occurs, it must be given a positive value. Otherwise, if an end-of-file or end-of-record condition occurs it must be given the value `IOSTAT_END` or `IOSTAT_EOR` (see Section 3.12), respectively. Otherwise, it must be given the value zero.
- `iomsg` is a scalar of `intent(inout)` and type `character(*).` If `iostat` is given a nonzero value, iomsg must be set to an explanatory message. Otherwise, it must not be altered.

Input/output within the subroutine to external files is limited to the specified unit and in the specified direction. However, input/output to an internal file is permitted. An input/output list may include a `DT` edit descriptor for a component of the `dtv` argument, with the obvious meaning.

The file position on entry is treated as a left tab limit and there is no record termination on return.

This feature is not available in combination with asynchronous input/output (next section).

4.2 Asynchronous input/output

Input/output may be asynchronous, that is, other statements may execute while an input/output statement is in execution. It is permitted only for external files opened with `ASYNCHRONOUS='YES'` in the `OPEN` statement and is indicated by `ASYNCHRONOUS='YES'` in the `READ` or `WRITE` statement. Execution of an asynchronous input/output statement initiates a 'pending' input/output operation, which is terminated by a wait operation for the file. This may be performed by an explicit wait statement

```
WAIT(10)
```

or implicitly by an `INQUIRE`, a `CLOSE`, or a file positioning statement for the file. The compiler is permitted to treat each asynchronous input/output statement as an ordinary input/output statement; this, after all, is just the limiting case of the input/output being fast. The compiler is, of course, required to recognize all the new syntax.

Further asynchronous input/output statements may be executed for the file before the wait statement is reached. The input/output statements for each file are performed in the same order as if they were synchronous.

An execution of an asynchronous input/output statement may be identified by a scalar integer variable in an `ID=` specifier. Successful execution of the statement causes the variable to be given a processor-dependent value which can be passed to a subsequent `WAIT` or `INQUIRE` statement as a scalar integer variable in an ID= specifier.

A wait statement may have `END=, EOR=, ERR=` and `IOSTAT=` specifiers. These have the same meanings as for an input/output statement and refer to situations that occur while the input/output operation is pending. If there is an `ID=` specifier, too, only the identified pending operation is terminated and the other specifiers refer to this; otherwise, all pending operations for the file are terminated in turn.

An INQUIRE statement is permitted to have a PENDING= specifier for a scalar default logical variable. If an ID= specifier is present, the variable is given the value true if the particular input/output operation is still pending and false otherwise. If no ID= specifier is present, the variable is given the value true if any input/output operations for the unit are still pending and false otherwise. In the 'false' case, wait operations are performed for the file or files. Wait operations are not performed in the 'true' case, even if some of the input/output operations are complete.

A file positioning statement (BACKSPACE, ENDFILE, REWIND) performs wait operations for all pending input/output operations for the file.

Asynchronous input/output is not permitted in conjunction with user-defined derived type input/output (Section 4.1) because it is anticipated that the number of characters actually written is likely to depend on the values of the variables.

A variable in a scoping unit is said to be an 'affector' of a pending input/output operation if any part of it is associated with any part of an item in the input/output list, namelist, or SIZE= specifier. While an input/output operation is pending, an affector is not permitted to be redefined, become undefined, or have its pointer association status changed. While an input operation is pending, an affector is also not permitted to be referenced or associated with a dummy argument with the VALUE attribute (Section 5.6).

The ASYNCHRONOUS attribute has been introduced to warn the compiler that some code motions across wait statements (or other statements that can cause wait operations) might lead to incorrect results. If a variable appears in an executable statement or a specification expression in a scoping unit and any statement of the scoping unit is executed while the variable is an affector, it must have the ASYNCHRONOUS attribute in the scoping unit.

A variable is automatically given this attribute if it or a subobject of it is an item in the input/output list, namelist, or SIZE= specifier of an asynchronous input/output statement. A named variable may be declared with this attribute:

```
INTEGER, ASYNCHRONOUS :: int_array(10)
```

or given it by the ASYNCHRONOUS statement

```
ASYNCHRONOUS :: int_array, another
```

This statement may be used to give the attribute to a variable that is accessed by use or host association.

Like the VOLATILE (Section 3.9) attribute, whether an object has the ASYNCHRONOUS attribute may vary between scoping units. All subobjects of a variable with the ASYNCHRONOUS attribute have the attribute.

There are restrictions that avoid any copying of an actual argument when the corresponding dummy argument has the ASYNCHRONOUS attribute.

4.3 FLUSH statement

Execution of a FLUSH statement for an external file causes data written to it to be available to other processes, or causes data placed in it by means other than Fortran to be available to a READ statement. The syntax is just like that of the file positioning statements.

4.4 IOMSG= specifier

Any input/output statement is permitted to have an IOMSG= specifier. This identifies a scalar variable of type default character into which the processor places a message if an error, end-of-file, or end-of-record condition occurs during execution of the statement. If no such condition occurs, the value of the variable is not changed.

4.5 Stream access input/output

Stream access is a new method of accessing an external file. It is established by specifying ACCESS='STREAM' on the OPEN statement and may be formatted or unformatted.

The file is positioned by 'file storage units', normally bytes, starting at position 1. The current position may be determined from a scalar integer variable in a POS= specifier of an INQUIRE statement for the unit. A required position may be indicated in a READ or WRITE statement by the POS= specifier which accepts a scalar integer

expression. For formatted input/output, the value must be 1 or a value previously returned in an INQUIRE statement for the file. In the absence of a POS= specifier, the file position is left unchanged.

The standard permits a processor to prohibit the use of POS= for particular files that do not have the properties necessary to support random positioning or the use of POS= for forward positioning.

4.6 ROUND= specifier

Rounding during formatted input/output may be controlled by the ROUND= specifier on the OPEN statement, which takes one of the values UP, DOWN, ZERO, NEAREST, COMPATIBLE, or PROCESSOR_DEFINED. It may be overridden by a ROUND= specifier in a READ or WRITE statement with one of these values. The meanings are obvious except for the difference between NEAREST and COMPATIBLE. Both refer to a closest representable value. If two are equidistant, which is taken is processor dependent for NEAREST and the value away from zero for COMPATIBLE.

The rounding mode may also be temporarily changed within a READ or WRITE statement by the RU, RD, RZ, RN, RC, and RP edit descriptors.

4.7 DECIMAL= specifier

The character that separates the parts of a decimal number in formatted input/output may be controlled by the DECIMAL= specifier on the OPEN statement, which takes one of the values COMMA or POINT. It may be overridden by a DECIMAL= specifier in a READ or WRITE statement with one of these values. If the mode is COMMA in list-directed input/output, values are separated by semicolons instead of commas.

The mode may also be temporarily changed within a READ or WRITE statement by the DC and DP edit descriptors.

This feature is intended for use in those countries in which decimal numbers are usually written with a comma rather than a decimal point: 469,23.

4.8 SIGN= specifier

The SIGN= specifier has been added to the OPEN statement. It can take the value SUPPRESS, PLUS, or PROCESSOR_DEFINED and controls the optional plus characters in formatted numeric output. It may be overridden by a SIGN= specifier in a WRITE statement with one of these values. The mode may also be temporarily changed within a WRITE statement by the SS, SP, and S edit descriptors, which are part of Fortran 95.

4.9 Kind type parameters of integer specifiers

Some of the integer specifiers (e.g. NEXTREC) were limited to default kind in Fortran 95. Any kind of integer is permitted in Fortran 2003.

4.10 Recursive input/output

A recursive input/output statement is one that is executed while another input/output statement is in execution. We met this in connection with derived-type input/output (Section 4.1). The only other situation in which it is allowed, and this is an extension from Fortran 95, is for input/output to/from an internal file where the statement does not modify any internal file other than its own.

4.11 Intrinsic function for newline character

The intrinsic function NEW_LINE(A) has been introduced to return the newline character. A must be of type character and specifies the kind of the result.

4.12 Input and output of IEEE exceptional values

Input and output of IEEE infinities and NaNs, now done in a variety of ways as extensions of Fortran 95, is specified. All the edit descriptors for reals treat these values in the same way and only the field width w is taken into account.

The output forms are

- `-Inf` or `-Infinity` for minus infinity
- `Inf, +Inf, Infinity`, or `+Infinity` for plus infinity
- `NaN`, optionally followed by non-blank characters in parentheses (to hold additional information).

Each is right justified in its field.

On input, upper- and lower-case letters are treated as equivalent. The forms are

- `-INF` or `-INFINITY` for minus infinity
- `INF, +INF, INFINITY`, or `+INFINITY` for plus infinity
- `NAN`, optionally followed by non-blank characters in parentheses for a `NaN`. With no such non-blank characters it is a quiet `NaN`.

4.13 Comma after a P edit descriptor

The comma after a `P` edit descriptor becomes optional when followed by a repeat specifier. For example, `1P2E12.4` is permitted (as it was in Fortran 66).

5 Interoperability with C

5.1 Introduction

Fortran 2003 provides a standardized mechanism for interoperating with C. Clearly, any entity involved must be such that equivalent declarations of it may be made in the two languages. This is enforced within the Fortran program by requiring all such entities to be ‘interoperable’. We will explain in turn what this requires for types, variables, and procedures. They are all requirements on the syntax so that the compiler knows at compile time whether an entity is interoperable. We finish with two examples.

5.2 Interoperability of intrinsic types

There is an intrinsic module named `ISO_C_BINDING` that contains named constants holding kind type parameter values for intrinsic types. Their names are shown in Table 1, together with the corresponding C types. The processor is not required to support all of them. Lack of support is indicated with a negative value.

Table 1. Interoperability between Fortran and C types

Type	Named constant	C type or types
INTEGER	C_INT	int
	C_SHORT	short int
	C_LONG	long int
	C_LONG_LONG	long long int
	C_SIGNED_CHAR	signed char, unsigned char
	C_SIZE_T	size_t
	C_INT8_T	int8_t
	C_INT16_T	int16_t
	C_INT32_T	int32_t
	C_INT64_T	int64_t
	C_INT_LEAST8_T	int_least8_t
	C_INT_LEAST16_T	int_least16_t
	C_INT_LEAST32_T	int_least32_t
	C_INT_LEAST64_T	int_least64_t
	C_INT_FAST8_T	int_fast8_t
	C_INT_FAST16_T	int_fast16_t
	C_INT_FAST32_T	int_fast32_t
	C_INT_FAST64_T	int_fast64_t
	C_INTMAX_T	intmax_t
	C_INTPTR_T	intptr_t
REAL	C_FLOAT	float
	C_DOUBLE	double
	C_LONG_DOUBLE	long double
COMPLEX	C_FLOAT_COMPLEX	float _Complex
	C_DOUBLE_COMPLEX	double _Complex
	C_LONG_DOUBLE_COMPLEX	long double _Complex
LOGICAL	C_BOOL	_Bool
CHARACTER	C_CHAR	char

For character type, interoperability also requires that the length type parameter be omitted or be specified by an initialization expression whose value is one. The following named constants (with the obvious meanings) are provided: C_NULL_CHAR, C_ALERT, C_BACKSPACE, C_FORM_FEED, C_NEW_LINE, C_CARRIAGE_RETURN, C_HORIZONTAL_TAB, C_VERTICAL_TAB.

5.3 Interoperability with C pointers

For interoperating with C pointers (which are just addresses), the module contains the derived types C_PTR and C_FUNPTR that are interoperable with C object and function pointer types, respectively. There are named constants C_NULL_PTR and C_NULL_FUNPTR for the corresponding null values of C.

The module also contains the following procedures:

C_LOC(X) is an inquiry function that returns the C address of an object. X is permitted to be

- a variable with interoperable type and type parameters that has the TARGET attribute and is either interoperable, an allocated allocatable variable, or a scalar pointer with a target; or
- a nonpolymorpic scalar without length parameters that has the TARGET attribute and is either an allocated allocatable variable, or a scalar pointer with a target.

C_FUNLOC(X) is an inquiry function that returns the C address of a procedure. X is permitted to be a procedure that is interoperable (see Section 5.6) or a pointer associated with such a procedure;

C_ASSOCIATED (C_PTR1[, C_PTR2]) is an inquiry function for object or function pointers. It returns a default logical scalar. It has the value false if C_PTR1 is a C null pointer or if C_PTR2 is present with a different value; otherwise, it has the value true.

C_F_POINTER (CPTR, FPTR [, SHAPE]) is a subroutine with arguments

- CPTR is a scalar of type C_PTR with INTENT(IN). Its value is the C address of an entity that is interoperable with variables of the type and type parameters of FPTR or was returned by a call of C_LOC for a variable of the type and type parameters of FPTR. It must not be the C address of a Fortran variable that does not have the TARGET attribute.
- FPTR is a pointer that becomes pointer associated with the target of CPTR. If it is an array, its shape is specified by SHAPE.
- SHAPE (optional) is a rank-one array of type integer with INTENT(IN). If present, its size is equal to the rank of FPTR. If FPTR is an array, it must be present.

C_F_PROCPOINTER (CPTR, FPTR)) is a subroutine with arguments

- CPTR is a scalar of type C_FUNPTR with INTENT(IN). Its value is the C address of a procedure that is interoperable.
- FPTR is a procedure pointer that becomes pointer associated with the target of CPTR.

This is the mechanism for passing dynamic arrays between the languages. A Fortran pointer target or assumed-shape array cannot be passed to C since its elements need not be contiguous in memory. However, an allocated allocatable array may be passed to C and an array allocated in C may be associated with a Fortran pointer.

Case (ii) of C_LOC allows the C program to receive a pointer to a Fortran scalar that is not interoperable. It is not intended that any use of it be made within C except to pass it back to Fortran, where C_F_POINTER is available to reconstruct the Fortran pointer.

5.4 Interoperability of derived types

For a derived type to be interoperable, it must be given the BIND attribute explicitly:

```
TYPE, BIND(C) :: MYTYPE
:
END TYPE MYTYPE
```

Each component must have interoperable type and type parameters, must not be a pointer, and must not be allocatable. This allows Fortran and C types to correspond, for example

```
typedef struct {
      int m, n;
      float r;
} myctype
```

is interoperable with

```
USE ISO_C_BINDING
TYPE, BIND(C) :: MYFTYPE
      INTEGER(C_INT) :: I, J
      REAL(C_FLOAT) :: S
END TYPE MYFTYPE
```

The name of the type and the names of the components are not significant for interoperability.

No Fortran type is interoperable with a C union type, struct type that contains a bit field, or struct type that contains a flexible array member.

5.5 Interoperability of variables

A scalar Fortran variable is interoperable if it is of interoperable type and type parameters, and is neither a pointer nor allocatable.

An array Fortran variable is interoperable if it is of interoperable type and type parameters, and is of explicit shape or assumed size. It interoperates with a C array of the same type, type parameters and shape, but with reversal of subscripts. For example, a Fortran array declared as

```
INTEGER :: A(18, 3:7, *)
```

is interoperable with a C array declared as

```
int b[][5][18]
```

5.6 Interoperability of procedures

For the sake of interoperability, a new attribute, VALUE, has been introduced for scalar dummy arguments. When the procedure is called, a copy of the actual argument is made. The dummy argument is a variable that may be altered during execution of the procedure, but on return no copy back takes place. If the type is character, the character length must be one.

A Fortran procedure is interoperable if it has an explicit interface and is declared with the BIND attribute:

```
FUNCTION FUNC(I, J, K, L, M), BIND(C)
```

All the dummy arguments must be interoperable. For a function, the result must be scalar and interoperable. The procedure has a 'binding label', which has global scope and is the name by which it is known to the C processor. By default, it is the lower-case version of the Fortran name. For example, the above function has the binding label func. An alternative binding label may be specified:

```
FUNCTION FUNC(I, J, K, L, M), BIND(C, NAME='C_Func')
```

Such a procedure corresponds to a C function prototype with the same binding label. For a function, the result must be interoperable with the prototype result. For a subroutine, the prototype must have a void result. A dummy argument with the VALUE attribute and of type other than C_PTR must correspond to a formal parameter of the prototype that is not of a pointer type. A dummy argument without the VALUE attribute or with the VALUE attribute and of type C_PTR must correspond to a formal parameter of the prototype that is of a pointer type.

5.7 Interoperability of global data

An interoperable module variable or a common block with interoperable members may be given the BIND attribute:

```
USE ISO_C_BINDING
INTEGER(C_INT), BIND(C) :: C_EXTERN
INTEGER(C_LONG) :: C2
BIND(C, NAME='myVariable') :: C2
COMMON /COM/ R, S
REAL(C_FLOAT) :: R, S
BIND(C) :: /COM/
```

It has a binding label defined by the same rules as for procedures and interoperates with a C variable of a corresponding type.

5.8 Example of Fortran calling C

C Function Prototype:

```
int C_Library_Function(void* sendbuf, int sendcount, int *recvcounts)
```

Fortran Module:

```
MODULE FTN_C
      INTERFACE
            INTEGER (C_INT) FUNCTION C_LIBRARY_FUNCTION &
            (SENDBUF, SENDCOUNT, RECVCOUNTS), &
```

```
            BIND(C, NAME='C_Library_Function')
            USE ISO_C_BINDING
            IMPLICIT NONE
            TYPE (C_PTR), VALUE :: SENDBUF
            INTEGER (C_INT), VALUE :: SENDCOUNT
            TYPE (C_PTR), VALUE :: RECVCOUNTS
            END FUNCTION C_LIBRARY_FUNCTION
      END INTERFACE
END MODULE FTN_C
```

Fortran Calling Sequence:

```
USE ISO_C_BINDING, ONLY: C_INT, C_FLOAT, C_LOC
USE FTN_C
...
REAL (C_FLOAT), TARGET :: SEND(100)
INTEGER (C_INT) :: SENDCOUNT
INTEGER (C_INT), ALLOCATABLE, TARGET :: RECVCOUNTS(:)
...
ALLOCATE( RECVCOUNTS(100) )
...
CALL C_LIBRARY_FUNCTION(C_LOC(SEND), SENDCOUNT, &
C_LOC(RECVCOUNTS))
...
```

5.9 Example of C calling Fortran

Fortran Code:

```
SUBROUTINE SIMULATION(ALPHA, BETA, GAMMA, DELTA, ARRAYS), BIND(C)
      USE ISO_C_BINDING
      IMPLICIT NONE
      INTEGER (C_LONG), VALUE :: ALPHA
      REAL (C_DOUBLE), INTENT(INOUT) :: BETA
      INTEGER (C_LONG), INTENT(OUT) :: GAMMA
      REAL (C_DOUBLE),DIMENSION(*),INTENT(IN) :: DELTA
      TYPE, BIND(C) :: PASS
            INTEGER (C_INT) :: LENC, LENF
            TYPE (C_PTR) :: C, F
      END TYPE PASS
      TYPE (PASS), INTENT(INOUT) :: ARRAYS
      REAL (C_FLOAT), ALLOCATABLE, TARGET, SAVE :: ETA(:)
      REAL (C_FLOAT), POINTER :: C_ARRAY(:)
      ...
      ! Associate C_ARRAY with an array allocated in C
      CALL C_F_POINTER (ARRAYS%C, C_ARRAY, (/ARRAYS%LENC/) )
      ...
      ! Allocate an array and make it available in C
      ARRAYS%LENF = 100
      ALLOCATE (ETA(ARRAYS%LENF))
      ARRAYS%F = C_LOC(ETA)

      ...
END SUBROUTINE SIMULATION
```

C Struct Declaration:

```
struct pass {int lenc, lenf; float*c, *f}
```

C Function Prototype:

```
void simulation(long alpha, double *beta, long *gamma,
double delta[], struct pass *arrays)
```

C Calling Sequence:

```
simulation(alpha, &beta, &gamma, delta, &arrays);
```

6 References

ASCII (1991) ISO/IEC 646:1991, Information technology – ISO 7-bit coded character set for information interchange. ISO, Geneva.

Cohen, Malcolm (ed.) (2001) ISO/IEC TR 15581(E) Technical Report: Information technology – Programming languages – Fortran – Enhanced data type facilities (second edition). ISO, Geneva.

IEEE (1989) IEC 60559: 1989, Binary floating-point arithmetic for microprocessor Systems. Originally IEEE 754-1985.

ISO 10646 (2000) ISO/IEC 10646-1:2000, Information technology – Universal multiple-octet coded character set (UCS) – Part 1: Architecture and basic multilingual plane. ISO, Geneva.

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

Reid, John (ed.) (2001) ISO/IEC TR 15580(E) Technical Report: Information technology – Programming languages – Fortran – Floating-point exception handling (second edition). ISO, Geneva.

WG5 (2004) ISO/IEC 1539-1:2004(E) Information technology – Programming languages – Fortran – Part 1: Base language. ISO, Geneva.