Date: 1998/05/28
To: J3
From: interop
Subject: Interoperability syntax (Part 1)
References: J3/98-132r1, J3/98-139

Describing pre-defined C data types

An **ISO_C_TYPES** module shall be provided that makes accessible the following named constants of type default integer: **C_INT**, **C_SHORT**, **C_LONG, C_LONG_LONG, C_SIGNED_CHAR, C_FLOAT, C_DOUBLE, C_LONG_DOUBLE, C_COMPLEX, C_DOUBLE_COMPLEX, C_LONG_DOUBLE_COMPLEX** and **C_CHAR**.

C_INT, C_SHORT, C_LONG, C_LONG_LONG and **C_SIGNED_CHAR** shall have values that are representation methods for integers that exist on the processor or shall have the value –1.

Because the C standard specifies that the representations for positive signed integers are the same as the representations for corresponding values of unsigned integers, and because Fortran will not provide any real support for unsigned kinds of integers, we have decided not to provide **C_UNSIGNED_INT, C_UNSIGNED_SHORT, C_UNSIGNED_LONG, C_UNSIGNED_LONG_LONG** or **C_UNSIGNED_CHAR** constants in the **ISO_C_TYPES** module. Instead a user can use the constants for the signed kinds of integers to access the unsigned kinds as well. Note that this has the potentially surprising side-effect that **unsigned char** would be declared as **INTEGER(C_SIGNED_CHAR)** in Fortran.

C_FLOAT, C_DOUBLE and **C_LONG_DOUBLE** shall have values that specify approximation methods for the real type that exist on the processor or shall have the value -1. The values of **C_COMPLEX, C_DOUBLE_COMPLEX,** and **C_LONG_DOUBLE_COMPLEX** may be the same as the values of **C_FLOAT, C_DOUBLE,** and **C_LONG_DOUBLE**, respectively, or –1 if not supported.

(***TBD - It's not clear whether C requires the layout of complex data to be similar to that used in Fortran. If so, the "may" in the above paragraph will become "shall".)***

The value of **C_CHAR** shall specify a representation method for characters that exists on the processor or shall have the value -1.

A scalar entity or derived type component in the "**Fortran type**" column of the following table, that has a kind type parameter that has the same value as the named constant made accessible from the **ISO_C_TYPE** modules specified in the "**Type kind**" column, are said to **interoperate** with scalars or structure components of C types that are *compatible (ref. C standard*) with the C types in the corresponding row of the "**C type**" column.

Fortran type	Type kind	C type
INTEGER	C_INT	**int** **unsigned int**
	C_SHORT	**short int** **unsigned short int**
	C_LONG	**long int** **unsigned long int**
	C_LONG_LONG	**long long int** **unsigned long long int**
	C_SIGNED_CHAR	**signed char** **unsigned char**
REAL	C_FLOAT	**float**
	C_DOUBLE	**double**
	C_LONG_DOUBLE	**long double**
COMPLEX	C_COMPLEX	**complex**
	C_DOUBLE_COMPLEX	**double complex**
	C_LONG_DOUBLE_COMPLEX	**long double complex**
CHARACTER	C_CHAR	**char**

So, for example, a scalar object of type integer, with a kind parameter equal to the value of **C_SHORT**, interoperates with a scalar object of the C type **short** or of any C type derived (via **typedef**) from **short**.

No other entities than those mentioned in this document shall be made accessible from the **ISO_C_TYPES** module. This prevents a program that is conforming with respect to one processor from being made non-conforming with respect to another due to names made accessible from the **ISO_C_TYPES** module.

C pointer types

The **ISO_C_TYPES** module shall make accessible an entity with the name **C_PTR**. This entity shall be a derived type or a type alias name. A Fortran scalar entity or derived type component of type **C_PTR interoperates** with C scalars or structure components that are of any C pointer type.

Note that this requires the representation method for all C pointer types to be the same for the C processor, if it is to be the "target" of interoperability of a Fortran processor. The C standard does not impose this requirement, so this may limit the ability of some processors to conform to Fortran 2000. Whether any C processors of interest actually take advantage of this needs to be determined.

Handling of structures

A new "attribute" (that's not the right term, but. . .) is introduced for Fortran derived type declarations. This is the BIND(C) attribute. For example,

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

Such a derived type definition shall not specify the SEQUENCE statement.

A Fortran scalar object or derived type component of derived type is said to **interoperate** with a C scalar object or derived type component of a **struct** type, if the derived type definition includes the **BIND(C)** attribute, the derived type and the **struct** type have the same number of components, and components of the derived type interoperate with the corresponding components of the **struct** type, which shall not be bit fields or arrays whose first bound is unspecified (***need correct C terminology again)***.

A Fortran derived type that specifies the BIND(C) attribute shall satisfy the following.

- It shall not be a parameterized derived type.
- It shall not specify either the EXTENDS extends or the EXTENSIBLE attribute.
- Any component that is of derived type shall be of a type that specifies the BIND(C) attribute as well.
- Any component shall not specify the POINTER nor the ALLOCATABLE attribute.

For example, a C scalar object of type **myctype** interoperates with a Fortran scalar object of type **myftype**.

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

Note that C9x requires the names and component names of two **struct** types to be the same in order for the types to be considered to be the same. This is similar to Fortran's rule describing when sequence derived types are considered to be the same type. However, because of the problem of mixed-case names in C, we have decided to be more forgiving.

Note that **union**s and bit fields are not supported. In addition, C **struct**s like the following cannot interoperate with any Fortran structure (this is a new feature of C9x):

```
struct {
   int m[];    /* Last component has an unspecified bound – like assumed-size */
}
```

Straw vote: Should we use BIND(C) or add an optional "(C)" to the SEQUENCE statement?

Handling of arrays

Because Fortran arrays are stored in column-major order, whereas C arrays are stored in row-major order, the *n*th dimension of a Fortran array corresponds to the *(r-n+1)*th dimension of the C array, where both arrays are of rank *r*.

A Fortran explicit-shape or assumed-size array object or structure component interoperates with C objects or structure components of array types if the elements interoperate, the ranks of the arrays are the same, and

1. if the Fortran array is explicit-shape, the extent in each dimension is the same as the extent of the corresponding dimension of the C array (***need C terminology here***); or
2. if the Fortran array is assumed-size, the extent in each dimension but the last must be the same as the extent of the corresponding dimension of the C array, and the extent of the first dimension the C array shall be unspecified.

- ***Need to deal with arrays of arrays of. . . . (This is mainly to ensure that we're using the correct terminology throughout to map Fortran's multi-dimensional arrays to C's arrays of arrays.) This is to be clarified in a future paper.***

Handling of characters

This is TBD in a future paper.

Type aliases

In order to facilitate portable use of C functions that use data types defined with C's **typedef** facility, a *type aliasing* statement is introduced to Fortran. The syntax is:

type-alias-stmt **is** TYPEALIAS :: *type-alias-list*
type-alias **is** *type-alias-name* => *type-spec*

A type alias name can then appear as the *type-spec* (R502) in a *type-declaration-stmt* (R501), a *component-def-stmt* (R425) or an *implicit-spec* (R542). Explicit or implicit declaration of an entity or component using a type alias name is identical to declaration using the *type-spec* for which it is an alias. The keyword TYPE is used in declarations.

For example,

```
  TYPEALIAS :: DOUBLECOMPLEX => COMPLEX(KIND(1.0D0)),     &
&                NEWTYPE => TYPE(DERIVED)
```

```
TYPE(DOUBLECOMPLEX) :: X, Y
TYPE(NEWTYPE) :: S
```

The type alias name can also be used as a structure constructor name, if it is an alias for a derived type.

A type alias name shall not be the same as the name of an intrinsic type. The type alias name declared is a local entity of class (1) (14.1.2), and can be made accessible via use or host association.

Note the => in the type alias statement syntax precludes making the :: optional. Otherwise, there is an ambiguity with pointer assignment in fixed source form. Also, the TYPE keyword is required when the type alias name is used in *type-spec*s, as there would be a potential for ambiguity in fixed source form were it omitted:

```
TYPEALIAS :: REWIND => LOGICAL
REWIND I
```

Suggestions that allow the "::" to be optional and suggested alternatives to using TYPE in declarations will be gladly entertained.

Note that the TYPEALIAS is not as flexible as C's **typedef** facility because certain type modifiers of C (such as array bounds) are attributes in Fortran, rather than being a part of the type.

Attributes of procedures and dummy arguments

A new VALUE attribute (along with a VALUE declaration statement) is defined for dummy data objects, and a BIND attribute is introduced for function and subroutine statement. A Fortran procedure interoperates with a C function if:

- the procedure is declared with the BIND(C) attribute;
- the results of the procedure and function interoperate, if the Fortran procedure is a function, or the result type of the C function is **void,** if the Fortran procedure is a subroutine;
- the number of dummy arguments of the Fortran procedure is equal to the number of formal parameters of the C function;
- dummy arguments with the VALUE attribute interoperate with corresponding formal parameters, and dummy arguments without the VALUE attribute correspond to formal parameters that are pointers whose reference types (***need correct C terminology here!)*** interoperate with the types of the corresponding dummy argument;

The BIND(C) attribute shall not be specified for a subroutine or function if it requires an explicit interface or has asterisk dummy arguments.

Note that the requirement that the Fortran procedure not require an explicit interface prohibits dummy arguments from having the POINTER attribute, having the ALLOCATABLE attribute, being assumed-shape arrays, having an array result, etc.

Here's an example:

```
BIND(C) INTEGER(C_SHORT) FUNCTION FUNC(I, J, K, L, M)
  INTEGER(C_INT), VALUE :: I
  REAL(C_DOUBLE) :: J
  INTEGER(C_INT) :: K, L(10)
  TYPE(C_PTR), VALUE :: M
END FUNCTION FUNC

short func(int i; double *j; int *k; int l[10], void *m);
```

This ties together some of the syntax specified in previous sections. Note that a C pointer may correspond to a Fortran dummy argument of type **C_PTR**, or to a Fortran scalar that does not have the VALUE attribute.

If an object is not a dummy argument, it shall not have the VALUE attribute. An object shall not have the VALUE attribute if it is an array or has INTENT(OUT) or INTENT(INOUT). A dummy argument of type character with a length parameter whose value is not one, shall not have the VALUE attribute.

If a dummy argument in a subprogram has the VALUE attribute, it implicitly has the INTENT(IN) attribute. (That is, VALUE can be used in a Fortran subprogram definition.)

The intent for BIND(C) is to be able to specify both procedures defined in C that can be called from Fortran, and subprograms defined in Fortran that can be called from C. Thus far, we have dealt with the former. The latter is to be dealt with in a future paper

Note also that, as an extension, a processor could provide additional BIND subattributes to specify particular C processors with the Fortran processor.

Name mangling

This is a facility to handle name changes that both allows the Fortran processor to specify with what name a C function or global data object was declared (since the rules for the two languages are different, particularly with regard to treatment of case), and to explicitly override the algorithm used to map a name to a "binder" name. More than one such name may be associated with a procedure. (See 98-139 for more details on direction.)

TBD – this will be completed in a future paper. The intended direction is to add additional optional subattributes to the BIND attribute.

Access to global data

This will be a facility to associate Fortran objects with global C data objects. Additional rules on association (similar to those specified for COMMON) will be required. It may be possible to extend such association to COMMON as well. Details TBD by a future paper.

The LOC intrinsic

This will be a function that returns the address of an object as a C_PTR. The precise details and restrictions are TBD in a future paper.

stdargs

A facility to specify a varying number of arguments to a procedure. The details are TBD by a future paper.

Conditions under which a C function may maintain a pointer to an object

Currently, Fortran's rules that enable optimization are such that, a reference to a procedure may cause a local object to be modified or referenced, if that object is accessible via host association, dummy argument association, has the TARGET attribute or the POINTER attribute. C's use of pointers needs to be taken into account here. Details are TBD by a future paper.