Subject: **HPF Version of C Interoperability**
Author: Jerry Wagener
Date: 11 Dec 1997

The following six pages comprise section 11.4 of the HPF-2 document, which describes one approach to Fortran - C interoperability.

At the Nov'97 J3 meeting, section B.3 of the Ada-95 standard was distributed as paper J3/97-254 (11 pages); it describes Ada - C interoperability.

J3/97-154 (50+ pages) is the latest version of the Fortran - C interoperability Technical Report produced for WG5 by Michael Hennecke.

A number of technical problems with J3/97-154 have been identified (J3/97-187r1, J3/97-188r1); in addition there are conceptual concerns with this approach (e.g., J3/97-189). Both J3/97-254 and the attached six pages offer additional approaches that could be considered for Fortran 2000 requirement R.9 (Interoperability with C).

```
      PROGRAM MY_TEST
        INTERFACE
          EXTRINSIC('HPF','SERIAL') SUBROUTINE GRAPH_DISPLAY(DATA)
            INTEGER, INTENT(IN) :: DATA(:, :)
          END SUBROUTINE GRAPH_DISPLAY
        END INTERFACE

        INTEGER, PARAMETER :: X_SIZE = 1024, Y_SIZE = 1024

        INTEGER DATA_ARRAY(X_SIZE, Y_SIZE)
!HPF$   DISTRIBUTE DATA_ARRAY(BLOCK, BLOCK)

!  Compute DATA_ARRAY
        ...
        CALL DISPLAY_DATA(DATA_ARRAY)
      END PROGRAM MY_TEST

! The definition of a graphical display subroutine.
! In some implementation-dependent fashion,
! this will plot a graph of the data in DATA.

      EXTRINSIC('HPF','SERIAL') SUBROUTINE GRAPH_DISPLAY(DATA)
        INTEGER, INTENT(IN) :: DATA(:, :)
        INTEGER :: X_IDX, Y_IDX

        DO Y_IDX = LBOUND(DATA, 2), UBOUND(DATA, 2)
          DO X_IDX = LBOUND(DATA, 1), UBOUND(DATA, 1)
          ...
          END DO
        END DO
      END SUBROUTINE GRAPH_DISPLAY
```

11.4 C Language Bindings

A common problem faced by Fortran users is the need to call procedures written in other languages, particularly those written in C or ones that have interfaces that can be described by C prototypes. Although many Fortran implementations provide methods that solve this problem, these solutions are rarely portable.

This section defines a method of specifying interfaces to procedures defined in C that removes most of the common obstacles to interoperability, while retaining portability.

11.4.1 Specification of Interfaces to Procedures Defined in C

If a user wishes to specify that a procedure is defined by a C procedure, this is specified with an *extrinsic-spec-arg* of `LANGUAGE = 'C'`, or an *extrinsic-kind-keyword* of `C`, as specified in Section 6.

For C subprograms for which `EXTRINSIC (LANGUAGE = 'C')` has been specified, the constraints associated with the syntax for *attr-spec-extended* (H1102) are extended as fol-

lows:

Constraint: A `LANGUAGE = 'C'` function shall have a scalar result of type integer, real or double precision.

Constraint: A dummy argument of a `LANGUAGE = 'C'` procedure shall not be an assumed-shape array, shall not have the `POINTER` attribute, shall not have the `TARGET` attribute, nor shall it have a subobject that has the `POINTER` attribute.

Constraint: The bounds of a dummy argument shall not be specified by specification expressions that are not constant specification expressions, nor shall the character length parameter of a dummy argument of such a procedure be specified by a specification expression that is not a constant specification expression.

Constraint: A *dummy-arg-list* of a `LANGUAGE = 'C'` subroutine shall not have a *dummy-arg* that is * or a dummy procedure.

The value of the *scalar-char-initialization-expr* in the `EXTERNAL_NAME` specifier gives the name of the procedure as defined in C. This value need not be the same as the procedure name specified by the *function-stmt* or *subroutine-stmt*. If `EXTERNAL_NAME` is not specified, it is as if it were specified with a value that is the same as the procedure name in lower case letters.

> *Advice to users.* Note that the `EXTERNAL_NAME` specifier does not necessarily specify the name by which a binder knows the procedure. It specifies the name by which the procedure would be known if it were referenced by a C program, and the HPF compiler is required to perform any transformations of that name that the C compiler would perform.
>
> The `EXTERNAL_NAME` specifier also allows the user to specify a name that might not be permitted by an HPF compiler, such as a name beginning with an underscore, or as a way of enforcing the distinction between upper and lower case characters in the name. (*End of advice to users.*)

The *extrinsic-spec-arg* of `LANGUAGE = 'C'` helps a compiler identify a procedure that is defined in C so that it can take appropriate steps to ensure that the procedure is invoked in the manner required by the C compiler.

> *Advice to implementors.* A vendor may feel compelled to provide support for more than one C compiler, if different C compilers available for a system provide different procedure calling conventions or different data type sizes. For instance, a vendor's compiler may provide support for a value of `GNU_C` in the `LANGUAGE=` specifier, or it may provide support through the use of compiler switches. (*End of advice to implementors.*)

11.4.2 Specification of Data Type Mappings for C

The extrinsic dummy argument feature, consisting of the `MAP_TO`, `LAYOUT`, and `PASS_BY` attributes, is the principal feature that facilitates referencing procedures defined in C from within Fortran programs. Together, these attributes allow the user to specify conversions required to associate the actual arguments specified in the procedure reference with the

formal arguments defined by the referenced procedure. In particular, the MAP_TO attribute indicates the type of the C data to which the HPF data shall be converted by the compiler; the PASS_BY attribute indicates whether a C pointer to the dummy argument needs to be passed; the LAYOUT attribute indicates for an array whether the array element order needs to be changed from Fortran's array element ordering to C's.

For C, the constraints associated with *attr-spec-extended*, *map-to-spec*, *layout-spec*, and *pass-by-spec* (H1102–H1105) are further extended as follows.

Constraint: The MAP_TO attribute shall be specified for all dummy arguments and function result variables of a LANGUAGE = 'C' explicit interface.

Constraint: The *map-to-spec* associated with a dummy argument shall be compatible with the type of the dummy argument. (See below for compatibility rules.)

Constraint: A LAYOUT attribute shall only be specified for a dummy argument that is an array.

Constraint: A LAYOUT attribute shall not be specified for an assumed-size array.

If the compiler is capable of representing letters in both upper and lower case, the value specified for a *map-to-spec*, *layout-spec* or *pass-by-spec* is without regard to case. Any blanks specified for a *map-to-spec*, *layout-spec* or *pass-by-spec* shall be ignored by the compiler for the purposes of determining its value.

An implementation shall provide a module, ISO_C, that shall define a derived type, C_VOID_POINTER. The components of the C_VOID_POINTER type shall be private.

> *Advice to users.* The C_VOID_POINTER type provides a method of using void * pointers in a program, but does not give the user any way of manipulating such a pointer in the Fortran part of the program, since I/O cannot be performed on an object with private components outside the module that defines the type, neither can the components or structure constructor of such a structure be used outside of the module that defines the type. (*End of advice to users.*)

The values permitted for a *map-to-spec* for LANGUAGE = 'C' are 'INT', 'LONG', 'SHORT', 'SIGNED_CHAR', 'FLOAT', 'DOUBLE', 'LONG_DOUBLE', 'CHAR', 'CHAR_PTR', 'VOID_PTR', or a comma-separated list, delimited by parentheses, of any of these values. The HPF types with which these are compatible are shown in the table below.

A *map-to-spec* that is a parenthesized list of values is compatible with a dummy argument of derived type if each value in the list is compatible with the corresponding component of the derived type.

When the PASS_BY attribute is used, the values permitted for a *pass-by-spec* are 'VAL', '*', or '**'. If no PASS_BY attribute is specified, then PASS_BY ('VAL') is assumed. If a *pass-by-spec* of VAL is specified, the dummy argument shall not have the INTENT(OUT) or INTENT(INOUT) attribute specified. If a value of '*' or '**' is specified for the *pass-by-spec*, an associated actual argument shall be a variable.

The value of the *map-to-spec* specified for a dummy argument in the interface body of a procedure for which a LANGUAGE= specifier whose value is C appears shall be such that at least one of the permitted mapped-to types is the same as the C data type of the corresponding formal argument in the C definition of the procedure (or a type that is compatible with one of the permitted mapped-to types). The C data type of a function in the C definition

of a procedure shall be one of the permitted mapped-to types (or a type that is equivalent to the permitted mapped-to types) specified for the function result variable in the interface body of a function with the LANGUAGE= specifier whose value is C. If a subroutine has been specified with a LANGUAGE= specifier whose value is C, the C definition of the procedure shall be specified with a data type of void.

The permitted mapped-to types for scalar dummy arguments of intrinsic type or of the derived type C_VOID_POINTER, are shown in the following table.

MAP_TO	Compatible With	C Type if PASS_BY		
		'VAL'	'*'	'**'
'INT'	INTEGER	int	int*	int**
'LONG'	INTEGER	long	long*	long**
'SHORT'	INTEGER	short	short*	short**
'SIGNED_CHAR'	INTEGER	signed char	signed char*	signed char**
'FLOAT'	REAL	float	float*	float**
'DOUBLE'	REAL	double	double*	double**
'LONG_DOUBLE'	REAL	double	double*	double**
'CHAR'	CHARACTER(1)	char	char*	char**
'CHAR_PTR'	CHARACTER	char*	char**	char***
'VOID_PTR'	C_VOID_POINTER	void*	void**	void***

The permitted mapped-to types of an array are the same as the permitted mapped-to types of a scalar variable of that type followed by a left bracket ([), followed by the extent of the corresponding dimension of the dummy argument, followed by a right bracket (]), for each dimension of the array. If no value is specified for the LAYOUT attribute, the corresponding dimensions of the dummy argument are determined from right to left; if the value C_ARRAY is specified for the LAYOUT attribute, the corresponding dimensions of the dummy argument are determined from left to right.

The value permitted for a LANGUAGE = 'C' *layout-spec* is C_ARRAY.

The permitted mapped-to types of a scalar variable of derived type are the structures whose corresponding members are of one of the permitted mapped-to types of the components of the derived type.

If there is a mismatch between the precision, representation method, range of permitted values or storage sequence between the type of the dummy argument and the permitted mapped-to type of the dummy argument, the compiler shall ensure that, for the duration of the reference to a procedure defined with a LANGUAGE= specifier whose value is C, the dummy argument is represented in a manner that is compatible with the expectations of the C processor for an object of the permitted mapped-to type. Upon return from the procedure, the compiler shall ensure that the value of an actual argument that is a variable is restored to the specified type and kind.

If the range of permitted values of the type and mapped-to type differ and the value of the actual argument or some subobject of the actual argument is not within the permitted range of the mapped-to type, the value of the associated dummy argument or subobject becomes undefined. Conversely, if the value of the dummy argument or some subobject of the dummy is not within the permitted range of values of the associated dummy argument, and the associated actual argument is a variable, the value of the associated actual argument or subobject of the actual becomes undefined.

Advice to users. These rules were created to ensure the portability of interoperability. However, it should be noted that for large objects, a significant overhead may be incurred if there is a mismatch between the representation method used for the data type versus the representation method used for the permitted mapped-to type. (*End of advice to users.*)

Advice to users. In some cases, this may cause the value of the actual argument to change without the value being modified by the procedure referenced. For example,

```
PROGRAM P
  INTERFACE
    EXTRINSIC(LANGUAGE='C') SUBROUTINE C_SUB(R,I)
      REAL(KIND(1.0D0)), MAP_TO('FLOAT'), PASS_BY('*') :: R
      INTEGER, MAP_TO('INT'), PASS_BY('*') :: I
    END SUBROUTINE C_SUB
  END INTERFACE
  REAL(KIND(0.0D0)) RR

  RR = 1.0D0 + 1.0D-10
  I = 123456789
  PRINT *, RR
  CALL C_SUB(RR, I)
  PRINT *, RR
END PROGRAM P

void c_sub(float *r, int *i)
{
}
```

might print

```
1.00000000010000000
1.00000000000000000
```

although the value of `*r` is not modified in `c_sub`. Similarly, the value of `I` might become undefined after the reference to `c_sub`, although `*i` is not modified.

Although it is good practice to avoid specifying a mapped-to type of `float` for a dummy argument of any type other than default real, or a mapped-to type of `double` for a dummy argument of any type other than double precision real, selecting an appropriate dummy argument type for objects requiring a mapped-to type `int` or `long` might not be so simple. (*End of advice to users.*)

If no *layout-spec* is specified for a dummy array argument, the array element order shall be the same as that specified by Fortran. If the value of *layout-spec* specified is `C_ARRAY`, the array element order of the array shall be transposed for the duration of the reference to the procedure.

11.4.2.1 Examples of Data Type Mappings

Some examples should help to clarify what sorts of C procedure definitions would be permitted given an interface body in a Fortran program. For example, the following interface body

```
INTERFACE
  EXTRINSIC('C') SUBROUTINE C_SUB(I, R, DARR, STRUCT)
    INTEGER, MAP_TO('INT') :: I
    REAL, MAP_TO('FLOAT'), PASS_BY('*') :: R
    REAL(KIND(1.0D0)), MAP_TO('DOUBLE') :: DARR(10)
    TYPE DT
      SEQUENCE
      INTEGER :: I, J
    END TYPE DT
    TYPE(DT), MAP_TO('(INT, LONG)'), PASS_BY('*') :: STRUCT
  END SUBROUTINE C_SUB
END INTERFACE
```

could correspond to a C procedure that has the prototype

```
void c_sub(int i, float r*, double darr[10], struct {int i, long j} *)
```

In the following example of the `LAYOUT` attribute,

```
PROGRAM P
  INTERFACE
    EXTRINSIC('C') SUBROUTINE C_SUB(A, B)
      INTEGER, MAP_TO('INT') :: A(2,2)
      INTEGER, MAP_TO('INT'), LAYOUT('C_ARRAY') :: B(2,2)
    END SUBROUTINE C_SUB
  END INTERFACE

  INTEGER :: AA(2,2), BB(2,2)
  CALL C_SUB(AA, BB)
END PROGRAM P

void c_sub(int a[2][2], b[2][2])
```

the correspondence between elements of `AA` and `a`, and elements of `BB` and `b` is

```
AA(1,1)   a[0][0]          BB(1,1)   b[0][0]
AA(2,1)   a[0][1]          BB(2,1)   b[1][0]
AA(1,2)   a[1][0]          BB(1,2)   b[0][1]
AA(2,2)   a[1][1]          BB(2,2)   b[1][1]
```

11.5 Fortran Language Bindings

When the language specified in an extrinsic definition is `Fortran` the rules are basically the same as those for HPF because HPF is based on the Fortran standard. There are a few issues to consider in this case: