ISO/IEC JTC1/SC22/WG5 N1185
X3J3 / 96-119
05 June 1996

To: WG5, X3J3
From: Michael Hennecke (personal submission)
Subject: Discussion of MAP_TO and BIND interoperability
References: ISO/IEC JTC1/SC22/WG5 N1184 (X3J3/96-118),
ISO/IEC JTC1/SC22/WG5 N1178 (X3J3/96-069), X3J3/96-106R1,
X3J3/95-295 (HPF Calling C Interoperability Proposal v1.3)

Document X3J3/96-106R1 is the X3J3 Liaison Report on Interoperability of Fortran and C, based on the draft Technical Report WG5/N1178. This report favours a MAP_TO approach to interoperability, as outlined in the HPFF proposal X3J3/95-295. This is the most important conflict between the current opinion of X3J3 and the current WG5 draft TR, N1178.

Being the project editor of this TR, I will, of course, try to produce a document following whatever direction WG5 specifies to be followed to address interoperability with ISO C.
But personally, I would strongly recommend to stay with the lines of N1178 and _not_ to implement a MAP_TO mechanism. This document is an attempt to explain my arguments for this position. It is _not_ a statement of the development body for this TR.

In the discussion I tracked, there were mainly two arguments in favour of the MAP_TO approach:

(i) A vendor does not need to support intrinsic KINDs for all C types. For example, a vendor does not support an INTEGER kind for 8-bit integers, but a C signed char must be mapped to a Fortran INTEGER. With MAP_TO, this can be done by converting to default integer, e.g. by writing "INTEGER, MAP_TO(signed_char) :: VARIABLE". With the BIND approach, the vendor would be required to provide an INTEGER(C_SCHAR_KI) type, and possibly also the whole set of intrinsic procedures and operations for this type. (Returning C_SCHAR_KI==-1 is also possible, but clearly not desirable :-)

(ii) The MAP_TO approach does not require the user to deal with type kind parameters, since mappings can always be specified to Fortran types of default kind.

Obviously, (i) makes it harder to implement the BIND approach for those Fortran compilers that do not support all C basic types. Disregarding all other differences of the two approaches, this would favor MAP_TO because it is the aim of a TR to keep the impact on overall compiler maintenance small. There is, however, no technical reason that a Fortran compiler should not support all basic types that the C compiler for the same hardware supports, and users will definitely benefit from extending the number of intrinsic kinds of the Fortran compiler to those types.

Concerning (ii), it is clearly a relief if users need not carry KIND type parameters through their programs. But this is not a problem of interoperability of Fortran and C, it is caused by the Fortran rule that actual and dummy argument must match identically - see topic (V).

In contrast to these two advantages of MAP_TO, I would like to point out some problems with the MAP_TO approach, which to my understanding are of a more fundamental nature than the advantages above:

(I) MAP_TO cannot handle C structs very well.

A "recursive MAP_TO" (I'd prefer to call it a "nested" MAP_TO) has been proposed by Jerry Wagener to extend MAP_TO to C structs, see http://www.uni-karlsruhe.de/~SC22WG5/TR-C/Email/may06-01 and my replies in the same directory. I am not aware of a more detailed specification of such nested MAP_TO, and I have some concerns about that mechanism as a whole:

(1) MAP_TO replicates a struct mapping at each reference.

Suppose we have the following C specifications:

```
  struct foo_type { int i; float f; };
  void foo1 ( struct foo_type arg1 ) { ... }
  void foo2 ( struct foo_type arg2 ) { ... }
  ...
```

With the BIND approach, this can be mapped to

```
  MODULE FOO_HEADER
    USE ISO_C
    TYPE FOO_TYPE
      BIND(C)
      INTEGER(C_INT_KI) :: I ; REAL(C_FLOAT_KR) :: F
    END TYPE FOO_TYPE
  END MODULE FOO_HEADER

  INTERFACE
    BIND(C,"foo1") SUBROUTINE FOO1(ARG1)
      USE FOO_HEADER ; TYPE(FOO_TYPE) :: ARG1
    END SUBROUTINE FOO1
    BIND(C,"foo2") SUBROUTINE FOO2(ARG2)
      USE FOO_HEADER ; TYPE(FOO_TYPE) :: ARG2
    END SUBROUTINE FOO2
    ...
  END INTERFACE
```

All interfaces with a "struct foo_type" argument only reference the TYPE(FOO_TYPE), which is consistently defined in one place.
With a MAP_TO approach, each interface has to specify a Fortran TYPE, and additionally the full MAP_TO for that struct:

```
  MODULE FOO_MODULE
    TYPE FRT_TYPE
      INTEGER :: I ; REAL :: F
    END TYPE FRT_TYPE
  END MODULE FOO_MODULE

  INTERFACE
    BIND(C,"foo1") SUBROUTINE FOO2(ARG1)
      USE FOO_MODULE ; TYPE(FRT_TYPE), MAP_TO(int;float) :: ARG1
    END SUBROUTINE FOO1              ! ^^^^^^^^^^^^^^^^^
    BIND(C,"foo2") SUBROUTINE FOO2(ARG2)
      USE FOO_MODULE ; TYPE(FRT_TYPE), MAP_TO(int;float) :: ARG2
    END SUBROUTINE FOO2              ! ^^^^^^^^^^^^^^^^^
    ...
  END INTERFACE
```

This is error-prone: changing the C struct requires changing at least one Fortran TYPE (the MAP_TOs might map to different Fortran TYPEs for ARG1 and ARG2), and one MAP_TO for each such "struct foo_type" dummy.

(2) MAP_TO does not allow encapsulation of "struct" TYPEs in MODULEs

Almost every API in C that Fortran may with to interface to comes with header files that contain, among others, struct definitions for derived types. The BIND approach uses a <bind-spec> within a Fortran TYPE definition to match the layout of struct and TYPE, that TYPE definition can be placed in a "header" module. As in the original C API, the TYPE name is all an application programmer needs to know to use the Fortran API.

MAP_TO separates the definition of a Fortran TYPE (probably also in a "header" module) from the specification of its mapping (in the MAP_TO attribute for dummy arguments of that type). MAP_TO additionally requires the application programmer writing down the MAP_TO to know the layout of the struct/TYPE - not only its name.

The important point is that this breaks the portability of the Fortran binding if variables of such type also appear as dummy arguments of _user_ procedures: with BIND everything is defined in a central MODULE (which hopefully is provided by the vendor of the package/API), whereas with MAP_TO the conversion rules for a TYPE in the API-header are specified in the user's program!

Suppose two vendors of a package use two different internal layouts of an opaque struct (with a documented name but "PRIVATE" components):

```
MODULE api_header                     MODULE api_header
! Vendor ABC's header                 ! Vendor XYZ's header
  TYPE api_type                         TYPE api_type
    INTEGER i ; REAL r                    REAL r ; INTEGER i
  END TYPE api_type                     END TYPE api_type
 CONTAINS                              CONTAINS
  SUBROUTINE api_proc(s)                SUBROUTINE api_proc(s)
   TYPE(api_type), &                     TYPE(api_type), &
     MAP_TO(int,float) :: s                MAP_TO(float,int) :: s
  END SUBROUTINE api_proc               END SUBROUTINE api_proc
END MODULE api_header                 END MODULE api_header
```

Of course, both vendors specify MAP_TOs matching their own layout of the TYPE(api_type) structure for all procedures the API specifies. So the following program is portable:

```
PROGRAM my_p
  USE api_header
  TYPE(api_type) :: s
  CALL api_proc(s)
END PROGRAM my_p
```

However, if a user wants to pass an object of TYPE(api_type) through a procedure interface that is _not_ provided by the vendor, a portablility problem arises:

```
SUBROUTINE my_sub ( my_s )
  USE api_header
  TYPE(api_type), MAP_TO(int,float) :: my_s  !  tied to vendor ABC
  CALL api_proc(my_s)
END SUBROUTINE my_sub
```

will not work with vendor XYZ's implementation because the MAP_TO specified in MY_SUB is for vendor ABC's layout of TYPE(api_type). Such a design of interoperability should be avoided, since copying details from the header files into the user's code will result in unportable code, which will be difficult to write and debug since it requires knowledge of details in MODULE api_header that are not intended for the user of the API.

(3) Handling nested structs with MAP_TO is clumsy and error-prone.

Consider the following C code, where a struct has struct components:

```
struct foo_subtype { double i; double j; double k; };
struct foo_type {
  float i; struct foo_subtype a;
  float j; struct foo_subtype b;
  struct foo_subtype c;
};
void foo ( struct foo_type arg ) { ... }
```

This could be mapped transparently with the BIND approach:

```
MODULE FOO_HEADER
  USE ISO_C
  TYPE FOO_SUBTYPE
    BIND(C)
    REAL(C_DBLE_KR) :: I, J, K
  END TYPE FOO_SUBTYPE
  TYPE FOO_TYPE
    BIND(C)
    REAL(C_FLT_KR) :: I ; TYPE(FOO_SUBTYPE) :: A
```

```
    REAL(C_FLT_KR) :: J ; TYPE(FOO_SUBTYPE) :: B
    TYPE(FOO_SUBTYPE) :: C
  END TYPE FOO_TYPE
END MODULE FOO_HEADER

INTERFACE
  BIND(C,"foo") SUBROUTINE FOO(ARG)
    USE FOO_HEADER
    TYPE(FOO_TYPE) :: ARG
  END SUBROUTINE FOO
END INTERFACE
```

However, using MAP_TO to do this mapping would blow up the interface:

```
  MODULE FOO_MODULE
    TYPE FRT_SUBTYPE
      DOUBLE PRECISION :: I, J, K
    END TYPE FRT_SUBTYPE
    TYPE FRT_TYPE
      REAL :: I ; TYPE(FRT_SUBTYPE) :: A
      REAL :: J ; TYPE(FRT_SUBTYPE) :: B
      TYPE(FRT_SUBTYPE) :: C
    END TYPE FRT_TYPE
  END MODULE FOO_MODULE

  INTERFACE
    BIND(C,"foo") SUBROUTINE FOO(ARG)
      USE FOO_MODULE
      TYPE(FRT_TYPE), MAP_TO( float, (double, double, double),  &
&                             float, (double, double, double),  &
&                             (double, double, double)        ) :: ARG
    END SUBROUTINE FOO
  END INTERFACE
```

This example again shows that the mapping of a struct should be tied to the Fortran TYPE definition. Anything else creates very complicated code, and the separation of TYPE definition (most likely in a "header" module) and mapping (in _each_ MAP_TO for dummy arguments) is likely to cause bugs for complicated structs. Such structs _do_ exist in APIs that Fortran might want to access...

(II) MAP_TO cannot handle typedef.

The typedef mechanism of C is very important to ensure the portability of application programming interfaces. A Fortran binding to an API in C should be able to deal with typedef-ed names - resolving the typedef names to "intrinsic" types requires knowledge of implementation details that are normally not specified in the API, and thus is non-portable. The BIND facility for derived types together with the proposed <type-alias-stmt> can handle typedef in a transparent way. MAP_TO is unable to deal with typedef, since this would require the processor which parses the C types in the MAP_TO to be aware of all user- or API-defined type names.

(III) MAP_TO does not handle C extern data objects.

It has been requested both by X3J3 and by members of WG5 that access to global data objects defined in C is provided. The MAP_TO mechanism has only been proposed for procedure interfaces, and extending it to variables is not a good idea. Since its semantics is that of a type cast (conversion), it may introduce huge run-time costs when accessing global C variables - almost every reference to a C data object would start up the MAP_TO machinery.

(IV) MAP_TO cannot deal with self-referential structures.

At present, both the HPFF approach and N1178 only define mappings for pointers of type void*. But N1178 can, should this be necessary, be extended to deal with more general C pointers, most importantly self-referential, dynamic data structures:

```
  TYPE LIST
    BIND(C)
    INTEGER(C_INT_KI)   :: DATA
    TYPE(list), POINTER :: NEXT_NODE
  END TYPE LIST
```

would be a natural mapping for

```
  struct list {
    int data;
    struct list *next_node;
  };
```

This is impossible with a MAP_TO solution, which does not include a type name and thus cannot specify self-referencial structs (at least, not in finite time and code size :-). Extensions as the one above may be needed if a C compiler represents a "struct list *" differently from a "void *" ...

(V) Comments on MAP_TO's type/kind conversions at a procedure call

In Fortran, actual argument type and type kind parameters must match exactly those of the corresponding dummy argument. This requires some typing for ALL applications that use non-default KINDs. Consider for example a library implemented like this:

```
  MODULE SOME_LIB
    INTEGER, PARAMETER :: WP = KIND(0.0D0)  !  instead of DOUBLE PREC.
   CONTAINS
    SUBROUTINE LIB_PROC ( A, B )
      REAL(WP), INTENT(IN)  :: A
      REAL(WP), INTENT(OUT) :: B
      B = 42.0_WP * A
    END SUBROUTINE LIB_PROC
  END MODULE SOME_LIB
```

If an application program accesses SOME_LIB, care must be taken that the KIND used in it and the KINDs of the application match: For example

```
  SUBROUTINE MY_PROC ( X, Y )
    USE SOME_LIB
    REAL     :: X, Y
    REAL(WP) :: YY
    CALL LIB_PROC(REAL(X,KIND=WP), YY)
    Y = YY
  END SUBROUTINE MY_PROC
```

would result if WP is not the default real kind. Since the above use of a WP type kind is preferred over DOUBLE PRECISION, for example, similar applications with hand-coded conversions will probably become common practice as Fortran 90/95 code bases increase. A good solution to this problem would be to allow calls with non-matching arguments IF an explicit interface is visible. In this case, the compiler could do the conversion, and the above example could read

```
  SUBROUTINE MY_F2000_PROC ( X, Y )
    USE SOME_LIB         !  compiler sees that dummy args are REAL(WP)
    REAL :: X, Y
    CALL LIB_PROC(X, Y)  !  compiler converts to/from REAL(WP)
  END SUBROUTINE MY_F2000_PROC
```

(Exactly this has been allowed in C at the transition from K&R to ISO C. It may be more complicated in Fortran because of generic interfaces.) Linking the automatic conversion to the visibility of an explicit interface (function prototype in C) seem to be natural. The rules for conversion can be identical to those that are already in section 7.5 of the standard: mapping X to A is similar to an A = X assignment.

The HPFF proposal (mis-) uses MAP_TO to do such conversions at a procedure call, which I think is not the ideal way to solve this broad class of problems. Apart from the fact that the "dummy kind" may be non-existent in the current Fortran compiler (see (i) above), there is no difference between calling a Fortran or a C procedure. MAP_TO solves only a small part of the problem, and because there is nothing in the Fortran standard which describes the "dummy argument C types", the detailed rules for type conversions with MAP_TO must be explicitly specified by the interoperability proposal. I am not aware of any document which contains this specification for the MAP_TO approach, and I fear that it will be a rather big task to do so.

The fact that C has more than one representation method for (signed) integers, for example, means that KIND parameters are the natural way to distinguish them in Fortran. Fortran’s inflexibility concerning non-matching kind type parameters should not be blamed on the BIND/KIND interoperability approach. Rather than specifying conversion/conformance rules both in section 7 of the standard and additionally/separately for each possible MAP_TO conversion, I would prefer to simply tell the programmer which kind type parameter designates C’s "long double" type, for example. If C has a basic type Fortran hasn’t, it seems more natural to simply _provide_ this type in Fortran that trying to say in a Fortran document how data objects of this unknown type should be converted to a Fortran type. Providing a _general_ mechanism in Fortran to allow procedure calls with non-matching arguments (if an explicit interface is visible) may be a good idea, but this is not a interoperability subject.