After all the discussion at the last few meetings about the details of this report, it occurs to me that there is a strategic flaw in the approach employed. Specifically, the report is based on the strategy of providing Fortran names for all the semantic representations one might need in communicating with a C procedure. This appears to me to have the following drawbacks:

- To the extent that this provision is incomplete, there may be C procedures that cannot be described. Further, because the Fortran names chosen are evocative of the C names but not systematically derived from them, vendors who might choose to fill in such gaps have no reasonable expectation that they will choose the same names as each other or as would be chosen in the next revision of this facility.

- On the other hand, the more complete this naming is, the more C semantic entities become "first class citizens" in Fortran, for use anywhere, not just in interfaces to C procedures:

  * If, for example, the C compiler to which the Fortran compiler is interfacing supports a 4-bit integer type or an 80-bit integer type, the Fortran compiler must provide a corresponding Fortran INTEGER kind which would be usable not only in C interfaces, but in the generic INTEGER intrinsic functions, as I/O units, iostat variables, etc.

  * If the C compiler supports a pragma that packs structures without concern for natural alignment, then the Fortran compiler must support something equivalent for derived types and support all those misaligned components <u>everywhere</u>, not just in the context of that type.

  * Since Fortran arrays are used to describe C arrays, Fortran arrays must be laid out in memory like C arrays — no distributing arrays across memories as in HPF!

  * Since C functions can have significant side effects, the Fortran optimizer may have to know this and deal with them as a special case rather than treating them the same way it treats Fortran functions.

  Further, if the same strategy is applied in interfacing to additional languages, Fortran could end up containing all the data types of all the languages with which it interoperates. (Remember the "success" of PL/1.)

- It seems to be me that this approach also makes the interoperability interfaces harder to write. The C programmer may be able to look at the documentation for a C procedure and at the report to identify the corresponding Fortran words but then not know enough Fortran to know what to do with them. The Fortran programmer may not know enough C to be able to recognize how the C syntax in the documentation is corresponds to the syntax in the report in order to be able to identify the corresponding Fortran words. Success may require substantial knowledge of Fortran, C, and the interoperability report.

How would I do things differently? I would describe Fortran's view of the C procedure in Fortran, the actual C interface in C, and let the interoperability facility provide the necessary mapping to connect them. For example,

```
      INTERFACE
        INTEGER FUNCTION fortran_name(i)
          LINK('C','int CName(int i)')
          INTEGER, INTENT(IN) :: I
        END FUNCTION fortran_name
      END INTERFACE
```

would implement Fortran calls to the procedure `fortran_name` by accessing the value of the Fortran `INTEGER` argument, converting it to the C `int` type, passing that result by value to the C procedure `CName`, take the C int value returned by the procedure by C linkage conventions, convert that value to Fortran `INTEGER` type, and return it as a function result by Fortran's linkage conventions. On many machine, `INTEGER` and `int` may have the same representation, and C and Fortran may return function results in the same way, so these conceptual conversions and changes may have no cost, but this process will work even if Fortran is running on one machine representing integers in decimal, C is running on another machine representing integers in binary, and the interoperation is via remote procedure calls.

```
      INTERFACE
        SUBROUTINE action(in, out)
          LINK('C','int ProcedureWithSideEffects(int arg)')
          INTEGER, INTENT(IN) :: in
          INTEGER, INTENT(OUT) :: out
        END SUBROUTINE action
      END INTERFACE
```

In this version, we have essentially the same C interface but a different Fortran interface in order to alert the compiler that it needs to deal with the fact that the C procedure has side effects.

```
      INTERFACE
        SUBROUTINE action2(in, out)
          LINK('C','void action2(int *in, *out)')
          INTEGER, INTENT(IN) :: in
          INTEGER, INTENT(OUT) :: out
        END SUBROUTINE action
      END INTERFACE
```

Now we have essentially same Fortran interface corresponding to a different C interface. If minimizing the conversion costs is important, a function such as `C_INT_KIND('long int')` could be used to identify the Fortran type that is closest to a C type (in this case, `long int`).

To cover the case where C pointers establish relationships that cannot readily preserved through the conceptual copying, I would add a `TYPE(C_OBJ)` to allow the Fortran program to allocate and keep track of objects having C types. The values of such objects would be accessed and set through intrinsic procedures that would perform the "conversion" between the C type and appropriate Fortran types. This could also be used as the method for accessing C external variables.

In the interests of keeping this on 2 pages, I will not elaborate further here.

Ω