This report is the collection of all of the comments from members of J3, the US Fortran standards committee, on the PDTR on Interoperability with C. The latest version of this document is paper number ISO/IEC JTC1/SC22/WG5 N1277 (a.k.a. NCITS/J3/97-154). There are many open issues, editorial flaws, and serious defects.

We have broken the issues into three categories:

1) those that we absolutely cannot accept without change;
2) those we feel need serious attention, and another round of review; and
3) editorial issues.

Category 1

Comment 1.
1.5
Is a processor permitted to make accessible entities in the ISO_C intrinsic module or other modules that are not defined by this PDTR? If so, the names may conflict with the name of a user-defined entity. This is similar to the issue Fortran has with permitting a standard-conforming processor to define intrinsic procedures that are not specified by the standard.

Comment 2.
1.5, paragraph after Note 1.1
Section 1.5 of DIS 1539-1 states that a processor conforms if it "contains the capability to detect and report the use within a submitted program unit of an additional form or relationship that is not permitted by the numbered syntax rules or their associated constraints". How should a conforming processor handle the additional forms and relationships specified by this PDTR?

Comment 3.
3, the whole section
If this section is intended to be the technical specification, then it should be written in the same form and with the same precision of terminology as the Fortran 95 standard, particularly since the edit (in Section 4) for page 292 says that section 3 is to simply be inserted into the Fortran standard as the new section 16.

Comment 4.
3.1, 1st paragraph after Note 3.3
It says that an implementation may support all or parts of the contents of the corresponding C standard header. This seems to be a large hole in portability if vendors can not only choose which headers they support but can also determine the contents of the headers.

Comment 5.
3.2, Note 3.6
Although the C standard requires that a C program not use two external names that are distinguished only by case, this TR needs to require a Fortran processor that does not support lower case letters to have some facility to enable the mapping to the C external name. For example,

```
 int MyCFunc(void)
 {}

 INTERFACE
   BIND(C,NAME='MYCFUNC') INTEGER(C_INT) FUNCTION F()
```

```
    USE ISO_C
  END FUNCTION F
 END INTERFACE
```

If the C processor preserves the case in the bind name for MyCFunc, the Fortran processor needs some way of getting to that name. It should probably be a method that's not specified in the TR, but still required.

Comment 6.
3.3.1, Note 3.10
Support for unsigned integers is still confusing. The paragraph after Note 3.10 notes that unsigned C types have the same size and alignment as their signed counterparts.

Given that the unsigned C types have the same representation as their associated signed types, there shouldn't be a need for the unsigned kinds, since there's no real support in Fortran for unsigned values.

Comment 7.
3.3.2
The first paragraph of this section states that ISO_C_FLOAT_H module shall provide a module with *constants* for the numeric limits provided by the <float.h> header. But most of the macros in <float.h> do not have to be constants. They can expand into function calls for example. So, a module could not reliably provide constants if the C implementation decided to delay floating-point C characteristics until runtime.

Comment 8.
3.3.4, Note 3.15
In the sentence

"Consequently, a NAME= clause in a BIND(C) specification within a derived type definition is not allowed."

should either be a constraint or a rule in prose in normative text. Notes are not normative.

Comment 9.
3.3.4, 3.3.6, pp. 15, 17
As currently defined, the PDTR only supports the concepts of arrays of characters and pointers to type `char`. There really is no easy, straightforward, and to a Fortran programmer, intuitive way of handling CHARACTER data. This is especially true when a Fortran programmer is trying to pass a CHARACTER variable, array element, or substring to a C procedure which expects a C-style null-terminated string. Several commercial compilers already offer a transformational function, usually called CSTRING, which takes a Fortran CHARACTER scalar data object and transforms it into a C-style null-terminated string. The PDTR should include such a capability.

Comment 10.
3.3.6, Note 3.20
Note 3.20 suggests that sequence association could be used to circumvent the problem that C permits 12 array-specs, while Fortran supports 7 array dimensions. We assume that this is suggesting that the rank specified for the dummy argument in the interface block would be seven or less, while the C array had 8 or more array-specs. If this is correct, it conflicts with the normative text in the paragraph that follows the note, which states the extents in the Fortran <array-spec> are those specified in the corresponding C array declarators (in reverse order). We read that as requiring the ranks to be the same. If they are not required to be the same, sequence association needs to be explicitly permitted, and the rules must be spelled out.

Comment 11.
3.3.7, C_ADDRESS, C_DEREFERENCE, C_INCREMENT
The argument to C_ADDRESS should be required to have the TARGET attribute; failing to require this severely hinders a processor's ability to perform optimization.

Comment 12.
3.3.7, C_DEREFERENCE and C_INCREMENT
Is the type of the MOLD argument specified permitted to be different from the type of the object from which the pointer was derived? Is C_INCREMENT permitted to specify an increment value that causes the dereference to exceed the bounds of an array? A user might expect to be able to do this in a case like the following:

```
    INTEGER, TARGET :: T1, T2
    COMMON /COM/ T1(1), T2(1)
    PRINT *, C_DEREFERENCE(C_INCREMENT(C_ADDRESS(T1), T1, 1), T1)
    END
```

Comment 13.
3.3.7, Note 3.24
The situations in which a pointer becomes "stale" need to be specified. This should be similar to the list of events that cause variables to become undefined (14.7.6 of 1539-1). That is, it is true of many more instances than just automatic objects. Why are automatic objects the only ones mentioned? The additional text added to clarify this point should be normative rather than informative.

Comment 14.
3.3.7, C_DEREFERENCE description
The description of case (iii) says that a dereference of a C_CHAR_PTR returns the whole string. Generally in C, a dereference of a character pointer only references a single character. If C_DEREFERENCE by definition always returns the whole string (like a C char pointer referenced in, say, strcpy()), then how does one use a C char pointer (from the Fortran side) to mimic the usual reference to only a single character?

Comment 15.
3.3.7, C_DEREFERENCE - Result value Case (i)
For the following example:

```
  integer(c_int) :: type(20)
  print *, C_DEREFERENCE(PTR, TYPE)
```

is MOLD_T considered to be "int" or "int[20]"? This makes a significant difference in the meaning.

Comment 16.
3.3.7, after C_DEREFERENCE
A C_SET_DEREFERENCE (say) subroutine is desirable. This would provide a method of setting a value through a pointer.

Comment 17.
3.3.9, R1606
The syntax of the <type-alias-stmt> leads to an ambiguity in fixed source form. One may name a pointer "TYPEXID" and one may have an array named "INTEGER" that has the TARGET attribute. Even in the presence of the ISO_C module,

```
        TYPE XID => INTEGER(c_ulong)
```

could easily be a pointer assignment statement.

One possibility is to require the "::" in fixed source form but this doesn't seem to be the optimal solution since the Fortran standard has no similar rule.

Also, it might be a good idea to permit a list of entities to be declared in a single <type-alias-stmt>. For example,

```
  type :: t1 => integer, t2 => real
```

And finally, the proposed syntax might cause people to confuse the objects declared with structures. We would like a keyword other than TYPE to be considered to call attention to the fact that the <type-alias-stmt> has different semantics.

<u>Comment 18.</u>
3.3.9, second constraint
Can the <type-alias-name> be the same as a variable name? a common block name? a procedure name? the name of a named constant? The second constraint isn't sufficient. The <type-alias-name> needs to be added to the list of local entities of class (1) in 1539-1 (14.1.2).

<u>Comment 19.</u>
3.3.9, paragraph after Note 3.28
The second sentence states:

> If the aliased <type-spec> is an intrinsic type, a <structure-constructor> for <type-alias-name> shall contain a single <expr>, which shall be assignment compatible with that intrinsic type.

The use of the term <structure-constructor> is misleading, since the value is not necessarily a structure. Why should the <type-alias-name> become a derived type if the <type-spec> is an intrinsic type? This doesn't make any sense for either Fortran or C (and is not like C; a typedef that names an intrinsic type does not suddenly create a struct). Perhaps a new non-terminal symbol (<alias-value-constructor>, for example) is needed. In addition, the meaning of this constructor is unclear, especially for intrinsic types; what is the value of the expression, what is its type?

Creating a new name for an intrinsic type would be a generally useful feature. We don't understand why the type alias needs to be a derived type if the type spec is for an intrinsic type and we think this seriously limits the usefulness and generality of this statement.

Note also that the first sentence of section 3.4.1 says "shall be a type alias for the implementation-defined integer type". We think a Fortran user would be very surprised that a type alias for an integer type is a derived type.

<u>Comment 20.</u>
3.4.3
If a NULL constant is being defined, why is C_ISNULL needed? And where does the value C_NULL come from? If this is defined later, there should be a forward reference here. If C_ISNULL is also going to be kept, why does the description of C_ISNULL compare the PTR argument to zero instead of to NULL?

<u>Comment 21.</u>
3.5.1.1, second paragraph
This states that if no <name-string> is present, the Fortran processor's rules are used to generate the external entry. Is this helpful? This means the user can't do anything portably if they don't specify the <name-string>. Why not specify that it's treated as if <name-string> was present with the value equal to the <function-name> or <subroutine-name>, with any lower-case letters converted to upper-case?

<u>Comment 22.</u>
3.5.1.2
This section appears to allow a "pointer to double" (and others, like "int *") to be passed as an argument to a C function, but the function itself cannot (portably) return a "pointer to double" value. This seems like a pretty limiting restriction. If the Fortran translator must somehow 'know' about pointers to basic types and pointers to structure types, then it seems like there is no technical reason why a C function cannot return a pointer to all of these types.

<u>Comment 23.</u>
3.5.1.5, After 4th bullet
Add a new bullet
"A function result shall not be an array."

<u>Comment 24.</u>
3.6
Some edits are needed to tie these objects in with the other global entities in 14.1.1 of 1539-1. One difference between these and other global entities is that it's not the name that is global, but the value specified by the <name-string>. Is the value of the <name-string> permitted to be the same as the name of any other global entity?

<u>Comment 25.</u>
4, New clause 16
This indicates that section 3 could be placed into IS 1539-1 almost unchanged. However, section 3 is not currently in a state that that could be done. For example, it would not be appropriate for Note 3.1 to appear in IS 1539-1. Also, the section contains rules and explanatory material that doesn't necessarily belong in a new section 16. For example, in 3.3.4, the first paragraph after Note 3.15 states "The POINTER <component-attr-spec> is not allowed because there is not C type whose corresponding Fortran type has the POINTER attribute." This more properly belongs in the edits to section 4.4.1 of IS 1539-1 as a constraint. Another example is the definition of the BIND statement in 3.2 rather than adding it as an edit to section 4 of IS 1539-1.

<u>Comment 26.</u>
General comment
Let's get a little philosophical. How can a vendor know if they conform to this TR and whose "fault" is it when a mixed language program fails? The intent seems to be one of giving Fortran programmers access to the operating system, graphics libraries, C language libraries, etc. However, there's no guarantee that these routines are written in C! They could easily be written in assembly. It's not uncommon for parts of the Standard C Library to be written in assembly. The C standard doesn't require that they actually be written in C.

Consider this scenario:

Vendor A provides a "standard conforming" C compiler for architecture X
Vendor B provides a "standard conforming" Fortran compiler for architecture X
A mixed-language program fails

Now, is it the Fortran vendor's "fault" if the C compiler is validated? That is, the C function works fine when called from C so it must be the Fortran compiler's problem.

This seems like dangerous logic to get into. The best that can be done here may be to make this an informative annex because there's no way to seriously check conformance or to arbitrate resolution when mixed language programs fail. Unless the Fortran committee wants to say if the function works when called from C then it must be a Fortran problem.

Now, if it can be the C compiler's fault then we recommend communication be made with WG14/J11 to make sure the C committee agrees with the responsibility that has just placed upon their shoulders.

Since the document chooses not to address mixed language I/O, every Fortran programmer wishing to call a C function cannot put a "printf" or "fprintf" statement into their C function as a means of debugging their code. It is quite common in C to use "fprintf(stderr, ...)" to issue diagnostics when a function has been called incorrectly. It sounds like any function a user might want to call from Fortran had better not do that sort of thing. The same is true with the "assert" macro. This seems to me to be a serious limitation for any mixed language program, especially if you want to call an existing library routine.

Exactly what routines are I/O routines? Is it some or all of the routines defined in the standard header <stdio.h>? For example, calling the "tmpnam" function seems totally innocuous. What about "sprintf" which is defined in <stdio.h> but doesn't really do any output to a file? What about the "assert" macro? How about the "system" function? If an X-Windows routine pops up a window on the screen and asks the user to enter their name, is that considered to be I/O?

Category 2

Comment 27.
3.1, Note 3.2
The note says that not all entities contained in <stddef.h> are required to be supported in ISO_C_STDDEF_H. Who chooses what is supported and what is not? Is this implementation-dependent?

Comment 28.
3.1, Paragraphs following Note 3.2
Should this document mandate the names of these modules without any specification of the contents? It's potentially confusing.

Comment 29.
3.2, R1601
Why are the LANG= and NAME= specifiers specified as being optional but PRAGMA= is required for each specified pragma?

Comment 30.
3.2, first constraint after R1604
Does the value of <name-string> include leading or trailing blanks? We assume so, but we would like this clarified since blanks are ignored in determining the value of an I/O specifier.

Comment 31.
3.3, Note 3.8
It states that "enum" types are not integer types (but rather integral types).

Defect Report #067 asks the question about which category enumerated types falls into. The reply starts out by saying:

> "Signed integer type", "unsigned integer type", and plain "integer type" are used interchangeably with "signed integral type", "unsigned integral type" and "integral type" in the C Standard.

So, an enumerated type must map onto one of the integer types, but the implementation need not reveal what the underlying type really is. I don't think you want to provide a binding to enum types in the TR, but Note 3.8 should be corrected. The problem is that there is no way to tell what the underlying integral type really is.

<u>Comment 32.</u>
3.3.1, the list of C basic types and Fortran intrinsic types
The names of the named constants should be spelled out so that the names have the same spellings as the C data types. For example, C_SHRT should be C_SHORT. We understand that Note 3.9 has the rationale for choosing the names, but Fortran programmers are not going to be looking in these headers. They are going to be somewhat familiar with the C data types so the terms used to describe these C data types should use the same words.

<u>Comment 33.</u>
3.3.1, Note 3.10
It states that the type "char" is not an integer type. This is an incorrect assertion for the same reason as cited for "enum" above.

<u>Comment 34.</u>
3.3.1, Note 3.12
The TR should not give suggestions about possible extensions.

<u>Comment 35.</u>
3.3.2, third paragraph
This indicates that the values made accessible shall conform to the requirements of the C standard. What if that requires representation of values that are not model numbers in the Fortran model, e.g. -2**31?

<u>Comment 36.</u>
3.3.3
Perhaps we could provide partial support for enum types with integer kind parameters named C_SCHAR_ENUM (for enums whose value is within the signed char range), C_SHORT_ENUM (for those whose value is within the short int range), etc. This may help in almost all cases, but does not necessarily solve the problem, since the C processor may not use straightforward rules in determining the representation for the enum type. Or perhaps a SELECTED_ENUM_KIND(LOW_ENUM, HIGH_ENUM).

Of course, if the suggestion that enums not be supported is taken, this comment can be ignored.

<u>Comment 37.</u>
3.3.4, paragraph following Note 3.15
The first sentence should be a constraint.

<u>Comment 38.</u>
3.3.6, Last paragraph
Change "may build on any" to "may build on either". On the other hand, what

does it mean for the Fortran binding to "build on" the C type? How can Fortran "build on" double *?

Should this paragraph be made into a note?

<u>Comment 39.</u>
3.3.6
Should mismatched array shapes be prohibited? It’s going to be easy for the programmer to get confused between row-major and column-major subtleties.

<u>Comment 40.</u>
3.3.7, second paragraph
In which module is C_NULL defined? Also, where are the C_ISNULL, C_ADDRESS, et al. functions defined?

<u>Comment 41.</u>
3.3.7, Description of C_ISNULL
Change "Compares PTR to zero"
to "Compares PTR to the appropriate C null pointer".
(or something like that).

Should this function be elemental rather than transformational? Is the function even necessary - why not provide operator(==) and operator(/=) instead?

<u>Comment 42.</u>
3.3.7, C_ADDRESS, C_DEREFERENCE, C_INCREMENT
Some of the arguments are permitted to be of any type. These should probably be restricted to be of types that are permitted in references to C procedures.

In addition, should zero-sized objects be prohibited from appearing?

<u>Comment 43.</u>
3.3.7, C_ADDRESS
Why is the result value undefined if OBJ is of one of the pointer types and C_ISNULL(OBJ) is true? Shouldn’t this procedure be returning a pointer to OBJ rather than a pointer to the objects OBJ points to?

<u>Comment 44.</u>
3.3.7, C_DEREFERENCE - Result value Case (i)
Change "*((MOLD_T *) PTR) where PTR is ... PTR."
to "*((MOLD_T *) PTR), where PTR is ... PTR, and MOLD_T is the type of MOLD."

<u>Comment 45.</u>
3.3.7, C_DEREFERENCE - Result value Case (ii)
The description of the MOLD argument indicates that the MOLD argument shall be present if PTR is of type TYPE(C_STRUCT_PTR), but Case (ii) indicates that the result has the value of *(PTR), rather than expressing it as *((MOLD_T *)PTR). Shouldn’t cases (ii) and (iii) be combined?

<u>Comment 46.</u>
3.3.7, C_DEREFERENCE - Result and Example Case (iii)
Change "ASCII NUL" to "NUL" (or whatever term is used to describe ’\0’ in the C standard - we don’t want to require support for ASCII.) The PDTR should say something about ’\0’ being the same as CHAR (0, KIND=C_CHAR).

<u>Comment 47.</u>
3.3.7, C_DEREFERENCE - Result value Case (iii)
The dependence on the string being NUL-terminated seems unfriendly. There's no requirement that char * point to a C string. Perhaps it should return the value of (char *) ('h' in this case), making it more consistent with the other two cases. A separate set of procedures could be defined to handle the string case.

<u>Comment 48.</u>
3.3.7, C_INCREMENT - Result value
Change "(PTR *)((MOLD *)PTR+N) where PTR is . . . PTR."
to "(PTR *)((MOLD_T *)PTR+N), where PTR is . . . PTR,
and MOLD_T is the type of MOLD."

<u>Comment 49.</u>
3.3.8
This section seems to imply that a C definition like:

```
  int func(int n, float x);

  int func(n, x)
    int n; float x;
  { return n + x; }
```

is not allowed because the definition uses old-style (even though a prototype is in scope). Is this intended? Seems like this should say that the "type of the definition includes a function parameter list" and not focus on the declarator.

<u>Comment 50.</u>
3.3.9, R1606
The meaning of the <access-spec> on the <type-alias-stmt> needs to be specified. Compare to p.40, lines 39-41 of 1539-1.

<u>Comment 51.</u>
3.3.9, first paragraph after constraints after R1606
Rule R502 of IS 1539-1 needs to be extended to permit TYPE(<type-alias-name>) as a <type-spec>.

<u>Comment 52.</u>
3.3.10
If we're not specifying the meaning of volatile at all, we shouldn't permit a Fortran entity to be associated with such an object.

<u>Comment 53.</u>
3.3.10, last paragraph
Instead of specifying that if a C object of a const-qualified type is used in a way that violates the C standard, the object becomes undefined, shouldn't we specify that such an object is not permitted to become redefined?

<u>Comment 54.</u>
3.3.11
Why is there a restriction on "register" being present for parameter declarations? Some rationale is needed.

<u>Comment 55.</u>
3.4.2, Argument

As with the arguments to C_ADDRESS, et al., should the type be restricted? Should the TYPE argument be required to be of a type that has the BIND(C) attribute or of an intrinsic type that has a kind parameter defined in the ISO_C module?

After "it shall be allocated." add "It shall not be an assumed-size array.".

Comment 56.
3.4.2, the Example
The example uses the kind type parameter C_CHAR to precede a character constant. Is there any implication here that C_CHAR implies the character constant is terminated by the null character? The Argument description for C_SIZEOF states that EXPR may be of any type. Does this mean that a default character constant can be passed to it or is there some unstated expectation that it must be of type C_CHAR?

Comment 57.
3.4.3
Why are all the new intrinsic functions prefixed by "C_" except for OFFSETOF? This seems to be quite inconsistent.

Comment 58.
3.4.3, Description
What does "its structure" mean? For example, in a structure reference of the form OUTER%MIDDLE%COMP is "its structure" defined to mean OUTER or OUTER%MIDDLE? Seems like it should be OUTER%MIDDLE since COMP is a component of OUTER%MIDDLE, but we can easily imagine that a user might want to know the offset of COMP within OUTER or within OUTER%MIDDLE. Does the specification of the TYPE argument (which really should be STRUCTURE because we’re talking about the offset within the object named with a structure name, not with a derived type name) allow

```
       C_OFFSETOF(OUTER, OUTER%MIDDLE%COMP)
```

as well as

```
       C_OFFSETOF(OUTER%MIDDLE, OUTER%MIDDLE%COMP)
```

If there is no intent of allowing the second case then the first argument is superfluous.

Comment 59.
3.5, first paragraph
This states that an explicit interface is required for a procedure defined by means of C, and that it have the BIND(C) attribute. Currently 1539-1 doesn’t require this, so this requirement would cause conforming Fortran 95 programs to be non-conforming with respect to this TR.

Comment 60.
3.5.1.2, Note 3.34
If the second sentence is "implying" a rule, the sentence should be moved out of the note and turned into a rule in normative text.

Comment 61.
3.5.1.2, second bullet
Delete second bullet. It is confusing in a list of supported items and how they are supported. Also, delete sentence that reads "All other C pointer types are not supported." Instead, insert before 1st paragraph of 3.5.1.2, something like:

"The return type of the C function shall be void, a basic type, a structure type, pointer to void, pointer to struct or pointer to char. The return type is not permitted to be an enumeration type, a union type or any other C pointer type."

Comment 62.
3.5.1.3, R1607
As in the BIND attribute, why is "PRAGMA=" required?

Comment 63.
3.5.1.3, the constraint for R1607
Why force a user to write a zero-length string? If it doesn't apply then just don't specify it.

Comment 64.
3.5.1.3, Constraint after R1608
Why not permit blanks that are not significant as is done with I/O specifiers?

Comment 65.
3.5.1.3, second paragraph after R1608
Change "A "*" character in the <pass-by-string>"
to "If the value of the <pass-by-string> is "*", it"
(Of course, the suggested modification will be affected by the following comment.)

Should some more suggestive value for the <pass-by-string> be used other than '*' and ''?

Reword the sentence that reads
"A "*" character in the. . . "pointer to T""
As written, it implies that "pointer to T" is a Fortran type, whereas there are no pointer types in Fortran.

Comment 66.
3.5.1.4, second bullet of first bulleted list and fifth bullet of the second bulleted list
Reword in a way that is consistent with the suggestion for 3.5.1.2, second bullet.

Comment 67.
3.5.1.4
It is unclear how arrays are passed to C. Specifically, C is row major and Fortran is column major, but the TR states that the Fortran interface "shall declare the type corresponding to the C type T, a DIMENSION attribute corresponding to the C array declarator, ...".

If the C parameter is declared:
```
       int A[2][3]
```
then 3.3.6 states that the corresponding Fortran declaration is:
```
       INTEGER, DIMENSION(3,2) :: A
```
but what happens if the (for all practical purposes) identical declarations of:
```
       int (*A)[3]
```
or
```
       int A[][3]
```
are used? Should the corresponding Fortran declaration be:
```
       INTEGER, DIMENSION(3,*) :: A
```
It is not clear from the TR whether this is correct.

Comment 68.
3.5.1.5, 4th bullet
Change "A dummy argument or function result" to "A dummy argument".

Comment 69.
3.5.2.1
Change "If the dummy argument"
to "If a dummy argument of a procedure with the BIND(C) attribute"
in the first sentence of each paragraph.

Comment 70.
3.5.2.1, second sentence (beginning "It shall be")
What does "It" refer to? The dummy argument or actual argument? Same comment for second sentence of second paragraph following Note 3.40 and for the second sentence of the paragraph following Note 3.42. This same sentence following Note 3.42 contains the phrase of type TYPE(C_VOID_PTR) which compares equal to NULL. Can it be compared equal to NULL or must it be passed to C_ISNULL? Does the font indicate the C NULL? If so, why? Why not specify that it must be equal to the Fortran NULL constant?

The same question applies to “It” in the second sentence of paragraph 3.

Comment 71.
3.5.2.1, 3rd paragraph
The sentence that begins "ASSIGNMENT(=) for the types. . . " seems to describe something that is unnatural. The entire concept that conversions for actual arguments happen implicitly on procedure references is foreign to Fortran, but support for implicit defined assignment even when the defined assignment is not accessible is very strange. In addition, what happens when the user redefines assignment for these types?

Comment 72.
3.5.2.2
Change "If a dummy argument"
to "If a dummy argument of a procedure with the BIND(C) attribute"
in the first sentence of each paragraph.

Comment 73.
3.5.3
The handling of <stdarg.h> seems clumsy. Why is the operator specified as "OPERATOR(//)" instead of just "//"? This form is used for interface blocks and makes one think that an interface block for the operator // must be provided somewhere. A suggested alternative would be to define a descriptive procedure in ISO_C_STDARG_H that has a variable number of arguments (similar to MAX and MIN). For example,

```
  call sub(va_list(i,j,k,a(17)))
  call sub(va_list(i,r))
```

If the first item in the list must always be VA_EMPTY, why make the user write it? The compiler can just construct the VA list this way.

Comment 74.
3.5.3
Since VA_LIST must be a derived type, it seems like the corresponding C type must be a structure type. Many implementations of va_list use a pointer instead. This could be a problem if, say, the minimum size for any derived

type is 64-bits but a pointer is 32-bits.

<u>Comment 75.</u>
3.6
It should be noted when such a variable becomes defined (as is done in 14.7.5 of 1539-1 for variables which are initialized).

<u>Comment 76.</u>
3.6, first bullet after Note 3.51
Change "No initialization shall appear in the <entity-decl>."
to "initialization shall not appear in an <entity-decl> in a <type-declaration-stmt> for a variable with the BIND(C) attribute."

<u>Comment 77.</u>
3.6, second bullet after Note 3.51
Change "ALLOCATABLE, PARAMETER or POINTER shall not be specified."
to "The variable shall not
 * have the ALLOCATABLE, PARAMETER or the POINTER attribute
 * be an automatic object
 * be a function result variable."

<u>Comment 78.</u>
3.6, last paragraph before Note 3.52
Change "If two or more. . . <name-string> are accessible in a scoping unit"
to "If two or more. . . <name-string>"

<u>Comment 79.</u>
3.6, last paragraph
It's not clear what this paragraph is saying. Is it talking about things like errno? Also, what does it mean to say that "The Fortran processor is not required to guard such behavior"?

<u>Comment 80.</u>
3.7, paragraph preceding Note 3.54
The last sentence is describing a comparison to (apparently) the C NULL again instead of using the Fortran NULL or C_ISNULL.

<u>Comment 81.</u>
4, 2nd constraint in edits for page 38
Change "the same <name-string>" to "the same <lang-keyword>".
(At least, we think that was what was intended.)

<u>Comment 82.</u>
4, edits for page 38
Is BIND(FORTRAN) permitted to appear in a derived type definition? If so, what effect does it have? Should the SEQUENCE statement still be prohibited for that case?

<u>Comment 83.</u>
General comment
C9X is due to hit the streets in 1999. So, by the time this TR makes it into a Fortran standard, there will be a new (and hopefully improved) C Standard. The TR should attempt to align itself with C9X.

Things to consider:

new keywords: restrict, complex

new headers: <complex.h> <fenv.h> <inttypes.h>
<bool.h> is likely to be added soon
external names: 31 characters, mixed case
new types: long long, unsigned long long, float complex,
double complex, long double complex,
restricted pointers, variable length arrays

Category 3

Comment 84.
1.4, item 1
Change "Mixed-Language Input and Output" to "Mixed-language input and output".

Comment 85.
1.4, item 4
Change "and some pointer types" to ", some pointer types, and bit fields".

Comment 86.
1.5
Delete "first-class" (twice) {we don't use this kind of terminology in the standard}

Comment 87.
2, first paragraph
What is meant by "the standard (de-facto or de-jure) computing environment? What standard? A Macintosh in someone's home does not generally have a C compiler on it and yet it seems to be a productive computing environment.

Comment 88.
2, second paragraph after bulleted list
Change "environment: Many" to "environment - many".

Also in this paragraph in the sentence beginning "Due to the difficulties...": People are not moving to C because of the difficulties of producing a standard for communications between Fortran and C; they are moving to C because there is no such standard at all or because it is more "natural" to write the application in C.

Comment 89.
3, Note 3.1
Given the general statement on section 3 (in category 1), this note should be deleted.

Comment 90.
3.1, the bulleted list
Who is going to provide these standard modules and by what mechanism are they going to be kept current with the C standard?

Comment 91.
3.1, second bullet in list
Is "common definitions" a C term? If not, a different term should be used to avoid confusion with the Fortran meaning of the word "common".

Comment 92.
3.1, First paragraph after Note 3.2
Change "facilites" to "facilities".

Comment 93.
3.1, 1st paragraph after Note 3.3
"name" appears in italics once, but other occurrences are not italicized.

Comment 94.
3.2, the first paragraph and throughout the remainder of the document
Each section reference should be qualified so the reader knows what document the section number is relative to. For example, in the first paragraph of 3.2, section 1.4 could refer to this document or to the Fortran 95 standard.

Comment 95.
3.2, paragraph following the constraints following R1604
Change "<lang-keyword>, this" to "<lang-keyword>. This".

Comment 96.
3.2, 1st paragraph after Note 3.6
Change "which" to "that".

Comment 97.
3.2, Constraint after R1605
Change "which" to "that".

Comment 98.
3.3, Note 3.7
Change "are supported, see" to "are supported; see".

Comment 99.
3.3, paragraph following Note 3.7
Change "define object types" to "define data types".

Comment 100.
3.3, Paragraph after Note 3.8
Change "Fortran types, access" to "Fortran types; access".
Change "C datatypes: Derived" to "C data types. Derived".
Change "recursively applied," to "recursively applied, and".

Comment 101.
3.3.1, first paragraph
Change "and real types: The intrinsic" to "and real types. The intrinsic".

Comment 102.
3.3.1, the list of C basic types and Fortran intrinsic types
The names of the kind parameters should be listed before specifying which data types correspond to which C data types.

Comment 103.
3.3.1, Note 3.11
It's not clear whether this note is suggesting things that an implementation needs to do to support the unsigned types or something the user needs to do.

Comment 104.
3.3.3, before Note 3.13
Change "implementation-defined: It" to "implementation-defined. It".

Comment 105.
3.3.4, Last sentence of paragraph after Note 3.14
Replace with

"A <component-initialization> shall not be specified for any
component of a derived type that has the BIND(C) attribute."

Comment 106.
3.3.4, Note 3.15
Change "the Fortran member objects" to "the Fortran derived type components".

Change "way: The" to "way. The".

Comment 107.
3.3.4, second paragraph following Note 3.15
This is the first time TYPE(C_STRUCT_PTR) has been seen. This may confuse the reader, and cause them to search back toward the front of the document to locate the definition of this term. It turns out that this term is defined later in the document. The term should either be defined before this reference or there should be a forward reference here to where this term is defined.

Comment 108.
3.3.4, Note 3.16
Same point as above for TYPE(C_CHAR_PTR) in the example.

Comment 109.
3.3.4, Paragraph after Note 3.16
Delete the sentence that begins "In either case, ...". It's not clear why the user might have thought that the length information would be stored in the structure.

Comment 110.
3.3.5, Note 3.18
Change "union members: In" to "union members. In".

Or just delete the note entirely. We shouldn't give suggestions as to how to write non-conforming code.

Comment 111.
3.3.6, Note 3.21
Change "the transposition must be done by the user"
to "one can use the RESHAPE intrinsic with the ORDER argument present".

Comment 112.
3.3.7, C_ISNULL, C_ADDRESS, et al.
Each of these functions should probably be in a separate little section (as is done for the Fortran intrinsic procedures).

Comment 113.
3.3.7, second paragraph
Change "are supported: The" to "are supported. The".

Comment 114.
3.3.7, paragraph preceding "C_ISNULL(PTR)"
Change "All C pointers" to "In a C program, all pointers". In the next sentence, insert "in Fortran" following "this comparison".

Comment 115.
3.3.7, Result value description and Example for C_ISNULL
The Fortran standard uses "true" and "false" for logical values in the

descriptions of intrinsic procedures rather than ".TRUE." and ".FALSE.".

Comment 116.
3.3.7, Note 3.22
It's not clear what the first sentence of this note is trying to say. Shouldn't it be sufficient to say that none of the operators is defined on these types. (We don't think it's really necessary to say even that much.)

Comment 117.
3.3.7, C_DEREFERENCE - Example Case (iii)
Change "character string of length 5" to "character string of length 6". (Is the length returned by LEN or strlen that is being discussed?)

Comment 118.
3.3.7, Note 3.27, paragraph following the extern example
The first sentence of the note should remain. The remainder of the note should be replaced with an example that would be valid. There is no guarantee that the representation of void * will be the same as the representation of char **. If this functionality is actually required, a C_CHAR_PTR_PTR type should be defined.

Comment 119.
3.3.7, Last paragraph
This last paragraph should be made informative.

Comment 120.
3.3.9, first constraint after R1606
Change "1539" to "1539-1". (There are other instances as well.)

Comment 121.
3.3.9, first paragraph after constraints after R1606
Change "interchangeable" to "interchangeably".
Change "corresponding <type-spec>: entities" to "corresponding <type-spec>. Entities".

Comment 122.
3.3.9, Note 3.29
Change "hidded" to "hidden".

Comment 123.
3.3.10, last paragraph
We think the sentence about a Fortran processor not being required to diagnose violations that take place while a C subprogram is executing can be deleted.

Comment 124.
3.3.11, Note 3.30
Delete "(which is comparable to Fortran PRIVATE entities)".

Comment 125.
3.4.1, Note 3.31
Why does this note exist? The result types of C_SIZEOF and OFFSETOF are explicitly described in the descriptions of these two new intrinsic functions.

Comment 126.
3.4.3
Delete the comma in the section title.

<u>Comment 127.</u>
3.4.3, 2nd paragraph
Change "follwing" to "following".

<u>Comment 128.</u>
3.4.3, Description
Change "strucure" to "structure".

Change both the second argument and the word "member" in the first sentence to the word "component". The Fortran standard uses "component", not "member".

In Result Characteristics, "imlementation-defined" is misspelled (missing the "p").

Since the next section of the description does not capitalize the word "value", "Characteristics" should also not be capitalized. (This same capitalization change should be made in other intrinsic descriptions elsewhere in the document as well.)

In Result value, delete the comma after "C standard)".

<u>Comment 129.</u>
3.5.1, first paragraph
Change "inluding" to "including".

<u>Comment 130.</u>
3.5.1, Note 3.33
Change "parantheses" to "parentheses".

Or just delete this note. It describes how a user might do something that is expressly prohibited by the normative text preceding the note.

<u>Comment 131.</u>
3.5.1.1, first sentence
Italicize "<interface-body>".

<u>Comment 132.</u>
3.5.1.2, 1st paragraph after Note 3.34
Delete "The declaration of the function result variable shall be as follows:"

<u>Comment 133.</u>
3.5.1.3, Constraint after R1608
Remove quotes around asterisk - they are not part of the value.

<u>Comment 134.</u>
3.5.1.3, first paragraph after R1608
Change "<pass-by-string> this" to "<pass-by-string>. This".

<u>Comment 135.</u>
3.5.1.4, second paragraph
Delete "The Fortran declaration. . . as follows:".

<u>Comment 136.</u>
3.5.1.4, sentence following Note 3.37
Change "of the C function" to "of a C function" and change "type: If" to "type. If".

Comment 137.
3.5.1.4, 6th bullet of second bulleted list
Change "funtion" to "function".

Comment 138.
3.5.1.4, last paragraph
Change "All other C pointer types are not supported."
to "No other C pointer types are permitted."

Comment 139.
3.5.1.5, 8th bullet
Change "shall have an explicit interface, and that interface"
to "shall have explicit interfaces, and those interfaces"

Comment 140.
3.5.2.1, last sentence before Note 3.43
Rather than indicating that the actual argument has to obey the same set of restrictions that something else obeys, repeat the restrictions for this case.

Comment 141.
3.5.2.1, Paragraph 3, last sentence.
Change “is” to “are”.

Comment 142.
3.5.2.3
Delete this note and section. It should not be necessary to call the user’s attention to this.

Comment 143.
3.5.2.3
The first occurrence of the word "free" need not be in bold Courier since it is not referring to the free() function.

Comment 144.
3.5.2.3
The phrase "to take care about" seems awkward.

Comment 145.
3.5.3, first paragraph
Change "procedure interfaces." to "procedure interface.".

Comment 146.
3.5.3, Note 3.45
Delete this note. Each of Fortran and C is able to do things that the other cannot.

Comment 147.
3.5.3, the bullet at the top of page 35
Change "to operands x1 of type" to "to operands x1 and x2 of type".

In the last sentence of this bullet, change "x<sub>2," to "x<sub>2;".

Comment 148.
3.6, first sentence after Note 3.51
Change "additonal" to "additional".

Comment 149.

3.6, third bullet after Note 3.51
Is it necessary to mention that CHARACTER with assumed character length is not permitted? This should follow from the fact that it is not permitted to be a dummy argument or named constant.

<u>Comment 150.</u>
3.6, last paragraph before Note 3.52
Change "They all refer to the same storage."
to "All such variables are storage associated."

<u>Comment 151.</u>
3.6, paragraph after Note 3.52
Is this paragraph necessary? Shouldn't this follow from the definition of storage association? Perhaps it should be made into a note if it's felt to be necessary.

<u>Comment 152.</u>
3.7, paragraph preceding Note 3.54
Change "an MOLD" to "a MOLD".

<u>Comment 153.</u>
3.7, Note 3.54
Change "allows to load X resources from command line arguments" to "allows X resources to be loaded from command line arguments".

<u>Comment 154.</u>
4, edit for page xvi
Change "defined by Fortran code" to "defined by a Fortran module program unit".

<u>Comment 155.</u>
4, edits for page 48
Change "may only" (two occurrences) to "shall only".
Change "which" to "that".

<u>Comment 156.</u>
4, Annex D

In most instances cross-references are left unqualified, so it is sometimes unclear whether the reference is to a section within the PDTR, a section in the Fortran DIS or a section in the ISO C standard.