ISO/IEC JTC1/SC22/WG5 N1174

International Standards Organization

Interoperability between Fortran and C

Technical Report defining extensions to
ISO/IEC 1539-1 : 1996

{Draft PDTR produced 08-Jan-96}

THIS PAGE TO BE REPLACED BY ISO CS

Contents

Foreword

[This page to be provided by ISO CS]

Introduction

This Technical Report defines extensions to the programming language Fortran to permit Fortran procedures to call C procedures. The current Fortran language is defined by the International Standard ISO/IEC 1539-1:1996, and the current C language is defined by the International Standard ISO/IEC 9899:1990.

This Technical Report has been prepared by ISO/IEC JTC1/SC22/WG5, the technical Working Group for the Fortran language. It is the intention of ISO/IEC JTC1/SC22/WG5 that the semantics and syntax described in this Technical Report shall be incorporated in the next revision of IS 1539-1 (Fortran) exactly as they are specified here, unless experience in the implementation and use of this feature has identified any errors which need to be corrected, or changes are required in order to achieve proper integration, in which case every reasonable effort will be made to minimise the impact of such integration changes on existing commercial implementations.

These extensions are being defined by means of a Type 2 Technical Report in the first instance to allow early publication of the proposed specification. This is to encourage early implementations of important extended functionalities in a consistent manner, and will allow extensive testing of the design of the extended functionality prior to its incorporation into the Fortran language by way of the revision of IS 1539-1 (Fortran).

Information Technology – Programming Languages – Fortran

Technical Report: Interoperability between Fortran and C

1 General

1.1 Scope

This Technical Report defines extensions to the programming language Fortran to permit Fortran procedures to call C procedures. The current Fortran language is defined by the International Standard ISO/IEC 1539-1:1996, and the current C language is defined by the International Standard ISO/IEC 9899:1990. The enhancements defined in this Technical Report cover three main areas. The first area addresses the mapping of external names of C to Fortran names. The second area provides mechanisms to map data types of C to Fortran data types, and the third area addresses the calling conventions of inter-language procedure calls.

1.2 Organization of this Technical Report

This document is organized in four sections, covering general issues and the three main areas mentioned above. Section 2 provides a rationale, which explains the need to define the features contained in this Technical Report in advance of the next revision of IS 1539-1 (Fortran) and motivates the specific implementation of these features. Section 3 contains a full description of the syntax and semantics of the features defined in this Technical Report, and section 4 contains a complete set of edits to ISO/IEC 1539-1:1996 that whould be necessary to incorporate these features in the Fortran standard. The non-normative annex A outlines possible extensions to the features specified in this Technical Report.

1.3 Inclusions

This Technical Report specifies:

1. The form that a Fortran interface to an external procedure defined by means of C may take
2. The rules for interpreting the meaning of a call to an external procedure defined by means of C

1.4 Exclusions

This Technical Report does not specify:

1. Mixed-Language Input and Output
2. Methods to access global C data objects from Fortran
3. Methods to automatically convert C header-files to Fortran
4. Methods to access Fortran program units from C

1.5 Conformance

The language extensions defined in this Technical Report are implemented by defining a number of first-class language constructs, and some intrinsic modules which make various named constants accessible to the Fortran program.

A program is conforming to this Technical Report if it uses only those forms and relationships described in IS 1539-1 or in this Technical Report, and if the program has an interpretation according to these two documents.

Note 1.1

Because this Technical Report defines extensions to the base Fortran language, a program conforming to this Technical Report is, in general, not a standard-conforming Fortran 95 program.
However, since it is the intention of WG5 to incorporate the semantics and syntax described in this document into the next revision of IS 1539-1, it is likely that a program conforming to this Technical Report will be a standard-conforming Fortran 2000 program.

A processor is conforming to this Technical Report if it is a standard-conforming processor as defined in section 1.5 of IS 1539-1, and makes all first-class language constructs and all intrinsic modules defined in this Technical Report intrinsically available. Additionally, a USE statement for an intrinsic module ISO_C shall be supported, that module shall be interpreted as containing one USE statement (without *rename* or *only* clauses) for each of the intrinsic modules defined in this Technical Report.

Editor's Note 1

See the edit for subclause 2.5.7 for accessibility of entities defined in intrinsic modules.

1.6 Notation used in this Technical Report

The notation used in this Technical Report is the notation defined in section 1.6 of IS 1539-1 (Fortran). However, deviations from these conventions are possible in descriptions of C language elements. In such cases, the syntactic conventions of IS 9899 (C) [actually, of ANSI X3.159-1989] are followed.

Editor's Note 2

During the drafting process, this Technical Report also contains non-normative "Editor's Notes" to spot out places in the document that need further processing.

1.7 Normative References

The following standards contain provisions which, through reference in this text, constitute provisions of this Technical Report. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this Technical Report are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of ISO and IEC maintain registers of currently valid International Standards.

ISO/IEC 646 : 1991	*Information Technology – ISO 7-bit coded character set for information interchange*
ISO/IEC 1539-1 : 1996	*Information Technology – Programming Languages – Fortran*
ISO/IEC 9899 : 1990	*Information Technology – Programming Languages – C*

Editor's Note 3

Currently, ISO/IEC 1539-1:1996 means the Fortran 95 CD (WG5 document N1122), and references to ISO/IEC 9899:1990 are actually references to ANSI X3.159-1989.
Non-normative reference is made to the HPF Language Specification v1.1 and the HPF calling C Interoperability Proposal v1.3.

2 Rationale

2.1 Justification of the Technical Report

From WG5 document N1131 (Request for subdivision):

> "A significant fraction of the standard (de-facto or de-jure) computing environment comes with a C API. Examples include X-windows libraries, Motif, TCP/IP socket calls and interfaces to system routines. The Fortran programmer is currently unable to exploit this wealth of software in a portable manner. This causes many problems for those who, for example, wish to front-end a powerful scientific visualisation package, written in Fortran, with a sophisticated graphical user interface (GUI). Due to the difficulties of providing such an interface in a standard fashion, many users are turning to alternative languages for such applications, even if Fortran is ideally suited to the "computational" component of the task.
>
> It is therefore very important that a standard mechanism by which C procedures can be called from Fortran procedures is defined as soon as possible."

2.2 Rationale for name binding

2.3 Rationale for datatype mapping

2.4 Rationale for procedure calling conventions

Editor's Note 4

Rational material will be added when the technical specification is complete. Background material of features from Fortran and C can be found in the "Notes" and "Rationale" sections of WG5 document N1147, available from `ftp://ftp.nag.co.uk/sc22wg5/`. Email sc22wg5.920 contains comments on N1131 (the request for subdivision), it is archived at `ftp://dkuug.dk/JTC1/SC22/WG5/920`.

3 Technical Specification

This section describes the extensions to the base Fortran language that this Technical Report defines to facilitate interoperability with the ISO C language, or more precisely to allow a Fortran program to call C procedures.

3.1 Name binding

Fortran provides two methods to declare an external procedure: the EXTERNAL statement and attribute which declare the name of the external procedure, and the *interface-body* in an *interface-block* which declares the name and interface of the external procedure.

If the external procedure is defined by means other than Fortran, there are two interoperability issues related to the name of the external procedure: The other language may have different rules for *name*s, and the other processor may apply different transformations to generate a "binder name" from a *name*.

Note 3.1

Interoperability issues related to the interface of the external procedure are discussed in section 3.3 of this Technical Report.

Note 3.2

Even if the external procedure is defined by means of Fortran, there may be a portability problem: It is not uncommon for Fortran processors to support different transformation rules to generate a "binder name", like appending or not appending an underscore character.

This section introduces a mechanism to specify a **binding name** (often abbreviated to **binding**) of a Fortran name, including features to addresses both of the above problems.

3.1.1 The binding attribute

The binding name for a Fortran name can be established by a *bind-spec* specification. This binding attribute may be used in a *bind-stmt* (3.1.2), in a type declaration statement for an external function, and in a function or subroutine statement within an *interface-body*.

Editor's Note 5

The extension of name binding to COMMON, to procedure definition, and to all other Fortran global entities have been moved to the appendix to keep the Technical Report small.

This section defines the form and semantics of the *bind-spec* binding specification, which may appear as an *attr-spec* (R503) or *prefix-spec* (R1219). The BIND statement is introduced in section 3.1.2. It is a *specification-stmt* (R214), the statement form of the *bind-spec*.

Riop?	*bind-spec*	**is**	BIND ([NAME=] *name-string* □ □ [, [LANG=] *lang-keyword*])
Riop?	*name-string*	**is**	*scalar-default-char-init-expr*
Riop?	*lang-keyword*	**is**	FORTRAN
		or	C

The support of other *lang-keyword*s than those specified in Riop? is processor-dependent. When no LANG clause is specified, the binding names generated are processor-dependent, as are those for processor-dependent *lang-keyword*s. The processor shall report the use of unsupported *lang-keyword*s.

If *lang-keyword* is FORTRAN, the value of *name-string* shall follow the rules for Fortran names. If *lang-keyword* is C, the value of *name-string* shall follow the rules for C *external names*. In all other cases, the set of characters allowed in the *name-string* is processor dependent.

Note 3.3

Note that although names of C entities are normally case-sensitive, a C processor may ignore the distinction of alphabetic case of *external names*. This limitation is implementation-defined.

A strictly conforming C program shall not rely on implementation-defined behavior. For this reason, a Fortran processor that does not support lowercase letters may still claim conformance to this Technical Report because it will be able to generate bindings to all external names that are allowed in a strictly conforming C program.

Interpretation of the *bind-spec* specification:

Editor's Note 6

To be added. Basically, generate binder name from *name-string* using Fortran transformation rules if LANG=FORTRAN and C transformation rules when LANG=C. Mention as a Note that compiler switches may alter these transformation rules. Blame the user if name conflicts occur.
Decide if the binding is part of the characteristic of the procedure.
Give LANG=FORTRAN interpretation: NAME="*xyz-name*" is redundant if the Fortran name is *xyz-name*, and a rename to *xyz-name* if the Fortran name is *abc-name*.

3.1.2 The BIND statement

A **BIND statement** specifies the binding attribute for the name of an external procedure.

Riop?	*bind-stmt*	**is**	*bind-spec* [::] *external-name*

Constraint: The *external-name* shall be the name of an external procedure.

The interpretation of the *bind-spec* appearing in the BIND statement is given in section 3.1.1. The BIND statement shall not be used to establish a name binding for an external subprogram in whose *specification-part* it appears.

Editor's Note 7

There is a similar ambiguity for the EXTERNAL statement in the current Fortran standard, and in the Fortran 95 CD. I have submitted a defect item to ensure that the *external-name* in an EXTERNAL statement shall not be the name of an external subprogram in whose *specification-part* it appears.

Note 3.4

Examples of name bindings to the C routine `double MPI_Wtime(void)` that show the *bind-spec* in various places are:

```
    USE iso_c, ONLY: c_dbl_kr

    REAL(c_dbl_kr), EXTERNAL, BIND("MPI_Wtime",C) :: MPI_WTIME

    REAL(c_dbl_kr) MPI_WTime
    EXTERNAL MPI_Wtime
    BIND("MPI_Wtime",C) MPI_WtImE

    INTERFACE
     FUNCTION MPI_WTIME ( )
      USE iso_c, ONLY: c_dbl_kr
      REAL(c_dbl_kr), BIND("MPI_Wtime",C) :: MPI_WTIME
     END FUNCTION MPI_WTIME
    END INTERFACE

    INTERFACE
     BIND("MPI_Wtime",C) FUNCTION MPI_WTIME ( )
      USE iso_c, ONLY: c_dbl_kr
      REAL(c_dbl_kr) MPI_WTIME
     END FUNCTION MPI_WTIME
    END INTERFACE
```

(The kind value `c_dbl_kr` is defined in section 3.2.)

Editor's Note 8

Relation to the HPFF proposal:
HPFF couples interoperability with C to the EXTRINSIC mechanism and defines a new *extrinsic-kind-keyword* "C" for the *extrinsic-prefix* (H601, H602). This implicitly takes care of the transformation rules to generate "binder names". To address the case-sensitivity of C names, an EXTERNAL_NAME clause is proposed, which takes a character string argument like the *name-string*. There are two ways to merge the two proposals:

- HPFF may adopt the solution defined here, by defining that in the absence of a LANG clause, the *extrinsic-kind-keyword* from the *extrinsic-prefix* is used as the *lang-keyword*.
- WG5 may adopt the HPFF solution. This implies that an explicit interface is always required when calling C procedures, and that the *extrinsic-prefix* is included in the language. In this case, it is always visible for which language the binding is to be generated and the LANG clause in the *bind-spec* is not needed.

In both cases, I prefer *bind-spec* to HPFF's EXTERNAL_NAME. (The names of the keywords are irrelevant, but the detailed specification of where they are allowed to appear and how they are interpreted is not. I have not seen specifications for EXTERNAL_NAME yet...)

3.2 Datatype mapping

When a Fortran program accesses C code there are three interoperability issues caused by the fact that the two languages have different datatypes:

1. the argument association of data objects defined in Fortran with a C procedure's dummy arguments,
2. the use of a result value of a C function in a Fortran expression, and
3. the access of global C data objects from within the Fortran program.

This section defines facilities to map C datatypes to Fortran datatypes, which is a necessary prerequisite to address these issues.

Note 3.5

To specify an inter-language procedure call, the last item is irrelevant (except for the possibility of side-effects of the C procedure), but a complete interoperability facility should include it.

Editor's Note 9

The third area, access of global C data, has been moved to the annex because a processor's implementation of COMMON may cause problems that are too time-consuming (for the development body, not the processor) to deal with in this TR. However, WG5 may wish to include this area in the TR, provided that the main goal (to facilitate portable procedure calling) is achieved in time...

Both languages define types that are intrinsically available, these are called *intrinsic types* in Fortran and *basic types* in C. Different sorts of *derived types* can be constructed from them. Section 3.2.1 specifies a complete mapping of C *basic types* to Fortran types, the remaining sections deal with C's *derived types*.

3.2.1 Matching C basic types with Fortran intrinsic types

The *basic types* of C are the *character types*, *integer types* and *floating types*.

Note 3.6

The C `enum` type is not specified to be a basic type in the C standard (it is an *integral type*, but not an *integer type*), but neither is it specified to be a derived type. This Technical Report treats `enum` as a basic type.

This Technical Report utilizes the kind type parameters of Fortran's intrinsic types to establish a one-to-one matching of C's *basic types* to Fortran character, integer and real types: An intrinsic module ISO_C_KINDS defines Fortran kind type parameters for all C basic types. The processor shall provide access to the named constants used in the model implementation below for all scoping units that contain a module reference to ISO_C_KINDS, subject to the rules of use association.

Editor's Note 10

Relation to the HPFF proposal:
HPFF uses the C_TYPE clause of a MAP_TO attribute (which is only defined within an EXTRINSIC(C) interface) to do datatype mapping. Apart from keywords for (some of) the basic types, support of `char *` and `void *` is also required. The semantics of C_TYPE is not specified exactly, but compared to a C cast which means a real conversion takes place (except if C_TYPE=NO_CHANGE, but this is the same as not specifying a mapping). There is no restriction that forbids, for example, a use of the form `CHARACTER, MAP_TO(FLOAT) :: DUMMY_ARG`, but neither does the proposal specify the semantics for such uses.
Implementation of MAP_TO is much more difficult than defining kind type parameters. Unrestricted pointers are controversial. The caller may do LAYOUT using the TRANSPOSE intrinsic (for rank-2 arrays).

```
MODULE iso_c_kinds   !  F95 module for C89 <basic types>
  IMPLICIT NONE

  ! KIND values for CHARACTER datatype (C <character types>):
  !
  INTEGER, PARAMETER ::  c_char_kc = <c-kind-param>
  INTEGER, PARAMETER :: c_schar_kc = <c-kind-param>
  INTEGER, PARAMETER :: c_uchar_kc = <c-kind-param>

  ! KIND values for INTEGER datatype (C <integer types>, enum):
  !
  INTEGER, PARAMETER :: c_schar_ki = <c-kind-param>
  INTEGER, PARAMETER :: c_uchar_ki = <c-kind-param>
  INTEGER, PARAMETER ::  c_shrt_ki = <c-kind-param>
  INTEGER, PARAMETER :: c_usort_ki = <c-kind-param>
  INTEGER, PARAMETER ::   c_int_ki = <c-kind-param>
  INTEGER, PARAMETER ::  c_uint_ki = <c-kind-param>
  INTEGER, PARAMETER ::  c_long_ki = <c-kind-param>
  INTEGER, PARAMETER :: c_ulong_ki = <c-kind-param>
  !
  INTEGER, PARAMETER ::  c_enum_ki = <c-kind-param>

  ! KIND values for REAL datatype (C <floating types>):
  !
  INTEGER, PARAMETER ::   c_flt_kr = <c-kind-param>
  INTEGER, PARAMETER ::   c_dbl_kr = <c-kind-param>
  INTEGER, PARAMETER ::  c_ldbl_kr = <c-kind-param>
END MODULE iso_c_kinds
```

If the processor supports a C datatype, the corresponding *c-kind-param* shall be a *kind-param* supported by the processor, otherwise it shall be a negative default integer constant. The value returned by C_CHAR_KC shall be the value of C_SCHAR_KC or the value of C_UCHAR_KC. Which of these two is returned is processor-dependent.

Note 3.7

In C, the question if `char` is implemented as `signed char` or `unsigned char` is implementation-defined. Only the so-qualified types are also *integer types*, the type `char` is not.

Editor's Note 11

If `enum`s are not implemented as integers, return a negative *c-kind-param*. If `unsigned` integers are too complicated, return a negative *c-kind-param*. If they are allowed to pe passed through procedure interfaces but not allowed to be defined by Fortran, return the *c-kind-param* of the corresponding `signed` type and impose that restriction. If Fortran operations on `unsigned` can be well-defined, do not impose that restriction.

3.2.2 Numerical limits of the C environment

The ISO C standard requires that a conforming C implementation shall document all its numerical limits in the headers `<limits.h>` and `<float.h>`. This Technical Report specifies two intrinsic modules that make these limits available in Fortran through constants having the same names as those defined in these headers. Except for the `unsigned` integer types, the values returned by a Fortran processor shall conform to the requirements of the C standard if that C type is supported by the Fortran processor.

Note 3.8

Fortran probably cannot represent the `unsigned` integer values.

```
MODULE iso_c_float_h  !  F95 module for C89 <float.h>
  USE iso_c_kinds
  IMPLICIT NONE

  INTEGER, PARAMETER ::      FLT_ROUNDS =  -1  !  indeterminable

  INTEGER, PARAMETER ::       FLT_RADIX =   2
  INTEGER, PARAMETER ::    FLT_MANT_DIG = <scalar-int-init-expr>
  INTEGER, PARAMETER ::    DBL_MANT_DIG = <scalar-int-init-expr>
  INTEGER, PARAMETER ::   LDBL_MANT_DIG = <scalar-int-init-expr>
  INTEGER, PARAMETER ::         FLT_DIG =   6
  INTEGER, PARAMETER ::         DBL_DIG =  10
  INTEGER, PARAMETER ::        LDBL_DIG =  10
  INTEGER, PARAMETER ::     FLT_MIN_EXP = <scalar-int-init-expr>
  INTEGER, PARAMETER ::     DBL_MIN_EXP = <scalar-int-init-expr>
  INTEGER, PARAMETER ::    LDBL_MIN_EXP = <scalar-int-init-expr>
  INTEGER, PARAMETER ::  FLT_MIN_10_EXP = -37
  INTEGER, PARAMETER ::  DBL_MIN_10_EXP = -37
  INTEGER, PARAMETER :: LDBL_MIN_10_EXP = -37
  INTEGER, PARAMETER ::     FLT_MAX_EXP = <scalar-int-init-expr>
  INTEGER, PARAMETER ::     DBL_MAX_EXP = <scalar-int-init-expr>
  INTEGER, PARAMETER ::    LDBL_MAX_EXP = <scalar-int-init-expr>
  INTEGER, PARAMETER ::  FLT_MAX_10_EXP =  37
  INTEGER, PARAMETER ::  DBL_MAX_10_EXP =  37
  INTEGER, PARAMETER :: LDBL_MAX_10_EXP =  37

  REAL(c_flt_kr),  PARAMETER ::   FLT_MAX     = 1.0E37_c_flt_kr
  REAL(c_dbl_kr),  PARAMETER ::   DBL_MAX     = 1.0E37_c_dbl_kr
  REAL(c_ldbl_kr), PARAMETER ::  LDBL_MAX     = 1.0E37_c_ldbl_kr
  REAL(c_flt_kr),  PARAMETER ::   FLT_EPSILON = 1.0E-5_c_flt_kr
  REAL(c_dbl_kr),  PARAMETER ::   DBL_EPSILON = 1.0E-9_c_dbl_kr
  REAL(c_ldbl_kr), PARAMETER ::  LDBL_EPSILON = 1.0E-9_c_ldbl_kr
  REAL(c_flt_kr),  PARAMETER ::   FLT_MIN     = 1.0E-37_c_flt_kr
  REAL(c_dbl_kr),  PARAMETER ::   DBL_MIN     = 1.0E-37_c_dbl_kr
  REAL(c_ldbl_kr), PARAMETER ::  LDBL_MIN     = 1.0E-37_c_ldbl_kr
END MODULE iso_c_float_h
```

Editor's Note 12

A processor may choose to define other values of `FLT_ROUNDS`. However, this may interfere with the rounding mode used by the C processor, and also with the Technical Report on Floating Point Exceptions.

```
MODULE iso_c_limits_h  !  F95 module for C89 <limits.h>
  USE iso_c_kinds
  IMPLICIT NONE

  INTEGER,             PARAMETER ::   CHAR_BIT =          8
  INTEGER(c_schar_ki), PARAMETER ::  SCHAR_MIN =       -127_c_schar_ki
  INTEGER(c_schar_ki), PARAMETER ::  SCHAR_MAX =        127_c_schar_ki
  INTEGER(c_uchar_ki), PARAMETER ::  UCHAR_MAX =          0
  INTEGER,             PARAMETER ::   CHAR_MIN = <scalar-int-init-expr>
  INTEGER,             PARAMETER ::   CHAR_MAX = <scalar-int-init-expr>
  INTEGER,             PARAMETER :: MB_LEN_MAX =          1
  INTEGER(c_shrt_ki),  PARAMETER ::   SHRT_MIN =     -32767_c_shrt_ki
  INTEGER(c_shrt_ki),  PARAMETER ::   SHRT_MAX =      32767_c_shrt_ki
  INTEGER(c_ushrt_ki), PARAMETER :   USHRT_MAX =          0
  INTEGER(c_int_ki),   PARAMETER ::    INT_MIN =     -32767_c_int_ki
  INTEGER(c_int_ki),   PARAMETER ::    INT_MAX =      32767_c_int_ki
  INTEGER(c_uint_ki),  PARAMETER ::   UINT_MAX =          0
  INTEGER(c_long_ki),  PARAMETER ::   LONG_MIN = -2147483647_c_long_ki
  INTEGER(c_long_ki),  PARAMETER ::   LONG_MAX =  2147483647_c_long_ki
  INTEGER(c_ulong_ki), PARAMETER ::  ULONG_MAX =          0
END MODULE iso_c_limits_h
```

If a processor returns a negative value for a *c-kind-param* defined in ISO_C_KINDS, it need not provide constants defined in ISO_C_LIMITS_H and ISO_C_FLOAT_H that use that *c-kind-param* as a *kind-param*. In this case, it is processor-dependent if the names of these constants are defined (with another kind type parameter supported by the processor) or not.

3.2.3 Mapping the C array type to Fortran

Editor's Note 13

To be provided. Different array element orders should not be converted automatically. C assumes a linear memory model, Fortran does not. Does the TR need to address this problem, or should it be in the responsibility of the user that the Fortran array is contiguous?

3.2.4 Mapping the C structure type to Fortran

Editor's Note 14

To be provided. A CSTRUCT clause as proposed in the X3J3 Liaison Report [X3J3/95-293] is probably the best solution. This could appear as an *access-spec* in the *derived-type-stmt*, or as an extension to the *private-sequence-stmt* within the type definition body.

3.2.5 Mapping the C union type to Fortran

This Technical Report does not provide features to map C `union` types to Fortran.

Editor's Note 15

See annex A. The user may possibly specify such mappings "manually", if a portable way to specify `struct` mappings is provided by the TR.

3.2.6 Handling of C pointer declarators

This Technical Report does not provide features to map general C pointers to Fortran. For support of pointer declarators of procedure arguments, see section 3.3.

3.2.7 No support of `typedef`

The C standard allows a *typedef-name*, which must have been previously defined, as a *type-specifier*. This declaration looks similar to a TYPE(*type-name*) in a Fortran *type-spec*, but the functionalities of these two specifications differ markedly:

- A Fortran *type-name* is always the name of a Fortran derived type (corresponding to a C `struct`), because it can only be defined in a *derived-type-def*.
- The definition of a C *typedef-name* by means of the *storage-class-specifyer* `typedef` can be used to establish a new name for any C type, not only for `struct` names.

Therefore, `typedef` cannot be supported without Fortran language extensions that allow such aliasing of type names. Such extensions may have a large impact on compilers, and are therfore not proposed in this Technical report.

Editor's Note 16

However, this feature is very important for the portability of calls to C procedures because many system and application packages make use of `typedef`s and do not document the ultimate types that are used. It should become a Fortran 2000 requirement.

3.2.8 No support of `<wchar.h>` and `<wctype.h>`

This Technical Report does not specify mappings for the types defined in `<wchar.h>` and `<wctype.h>`, which are standardized in Normative Addendum 1 to IS 9899.

3.3 Procedure calling conventions

Editor's Note 17

Still not developed.

Using something like the EXTRINSIC(C) mechanism of HPF seems a good idea to deal with some of the calling conventions by a single "switch". But instead of using LOC (or the operator form) on each call, it should be tried to specify this in the interface (which is always explicit), the compiler can then pass the address automatically. As a side effect, one gets rid of the dangers of unrestricted pointer features on the Fortran side. Some standard form of the "existing practice" %BYVAL and %BYREF specifications may be established for this purpose, but restricted to within interface blocks.

If the MAP_TO specification is not used (for datatype mapping, kind type parameters seem to be more convenient, they also have the advantage that they can be used for derived-type definitions that are needed for `struct`s), some similar method can be used to indicate that an argument is a null-terminated C character string (in fact, actual implementations use %BYREF this way for character strings).

Difficulties may arise because probably too many assumptions on how a C procedure passes its arguments are made. Does the C standard really specify that `*a`, `a[]`, `a[42]`, `a[9][6]`, ... are all passed the same way? Pointers to different C types may all be different, so passing a pointer to a `struct` like the often-quoted `Display` is **`not`** guaranteed to be the same as passing a pointer to `void`; although such a cast must be *possible*, it need not be actually *done* in the course of the procedure call...

Restrictions on external procedures written in C can be quickly summarized: they shall behave as if they were external subprograms. Every violation of this principle is nonconforming.

Restrictions on the interface specifications allowed for external C procedures depend on the details of proposed extensions, which are not worked out yet. If the processor is aware of being in an interface for a C procedure, the implementation of specifications like INTENT may be automatically chosen to conform to the C conventions, still allowing the Fortran processor to perform checks according to Fortran semantics. Which kinds of *array-spec*s are allowed is probably processor-dependent (of both the Fortran and C processor), and mainly depends on how much automatic conversion is put into the interface.

4 Editorial changes to ISO/IEC 1539-1 : 1996

The following subsections contain the editorial changes to ISO/IEC 1539-1:1996 required to include the extensions defined in this Technical Report in a revised version of the International Standard for the Fortran language.

Editor's Note 18

These are mostly edits for name binding, as in WG5 document N1147.

Page xiv
Line 23
Update the "Organization of this International Standard" subclause.

Page 1
Subclause 1.4
Lines 22 and 23: Exclusions (1) and (2) may be affected.

Page 2
Subclause 1.5
Conformance paragraph at line 37 may be affected.

Page 7
Subclause 1.9
At the end of the references, add

> ISO/IEC 9899:1990, Information technology – Programming languages – C (also ANSI X3.159-1989, American National Standard for Information Systems – Programming Language – C)

Page 10
Subclause 2.1
In

R214	*specification-stmt*	**is**	*access-stmt*
		or	*allocatable-stmt*
		...	

add after line 32:

		or	*bind-stmt*

Page 18
Subclause 2.5.7
In line 40, change "procedures" to "procedures, modules". After line 41, add

> Entities defined in an intrinsic module may be used without further definition or specification in those scoping units that contain a module reference for that intrinsic module, subject to the rules of use association (11.3.2).

Page 47
Subclause 5.1
In

R503	*attr-spec*	**is**	PARAMETER
		or	*access-spec*
		. . .	

add after line 27:

		or	*bind-spec*

Page 48
Subclause 5.1
In the Constraints list, add after line 20:

Constraint: If a *bind-spec* is specified, the *entity-decl-list* shall be a single *entity-decl*, and that entity shall be an external function.

Editor's Note 19

The two valid occurences are in conjunction with the EXTERNAL attribute, and inside an *interface-body*. Perhaps "shall be an external function" is not restrictive enough: does it allow a *bind-spec* on external procedure definition?

Page 52
Subclause 5.1.2
After section 5.1.2.2, insert a new section after line 37:

> **5.1.2.2a Binding attribute**
>
> The **binding attribute** specifies the name binding of an external function name. Name binding and related interoperability issues are described in section ?. This attribute may also be declared via the BIND statement (?.?.?).

Page 57
Subclause 5.2
At line 17, change

> This also applies to EXTERNAL and INTRINSIC statements.

to

> This also applies to BIND, EXTERNAL and INTRINSIC statements.

Page 191
Subclause 12.3.2.1
"Procedure interface block" may be affected.

Page 205
Subclause 12.5.2.2
In

```
R1219   prefix-spec        is   type-spec
                           or   RECURSIVE
                           or   PURE
                           or   ELEMENTAL
```

add after line 33:

```
                           or   bind-spec
```

Page 205
Subclause 12.5.2.2
In the Constraints list following R1219, add after line 37:

Constraint: A *bind-spec* shall only be specified within an *interface-body*.

Page 210
Subclause 12.5.3
"Procedures defined by means other than Fortran" may be affected.

Page 271
Subclause 14.1
"Scope of names" may be affected.

Page 288
New clause 16
Introduce a new section 16 (Interoperability with ISO C)

Editor's Note 20

This is a big edit. The final form of section 3 of this TR should be that this edit reads "take section 3, replace section heading with 'Interoperability with ISO C', replace all 'TR' by 'IS', renumber sectioning, rules and notes, and include the result as section 16 into IS 1539-1."

Page 289
Annex A
Update the Glossary:
After 289:37, add the term **binding** with a definition.
After 294:3, add a line 294:3a "**name binding** (?.?): See *binding*."

Page 305
Annex C
C.9.2 "Procedures defined by means other than Fortran (12.5.3) " and C.9.3 "Procedure interfaces (12.3)" on pages 329+ may be affected.

Page 343
Annex D
Update the Index :-)

A Possible extensions

There are two areas in which extensions to the basic features defined in the normative part of this Technical Report may be desirable:

- Some extensions might be desirable, but their addition to compilers will possibly have a large impact on the overall compiler maintenance process. These features do not satisfy the criteria established in WG5 document N1152. However, where their implementation in the Technical Report is easy, this annex contains the corresponding specifications.

- The second area is the revision process of the C standard. The next revision of ISO C, informally known as C9X, is scheduled to register the CD in December 1996, with CD and DIS ballots in December 1997 and 1998. This means there is no time to include features from C9X in this Technical report. However, where drafts of important extensions to ISO C are sufficiently well developed, this annex establishes corresponding Fortran extensions.

WG5 may wish to include some or all of the features specified in this annex in a revision of either IS 1539-1 or this Technical Report.

A.1 Features not selected for the Technical Report

A.1.1 Access to global C data objects via COMMON

To access `extern` data objects of C, two things are needed: the name binding facility must be extended to Fortran common blocks, and the layout of the common blocks must match the C datatype of the `extern` data object.

Editor's Note 21

Additionally, the Fortran processor is assumed to allocate common storage statically, and without any "header" information before the Fortran data objects in the common block.

The edit to implement name binding for common blocks is:

Page 68
Subclause 5.5.2
Change lines 12 to 14

R549 *common-stmt* **is** COMMON [/ [*common-block-name*] /] □
□ *common-block-object-list* □
□ [[,] / [*common-block-name*] / *common-block-object-list*] ...

to

R549 *common-stmt* **is** COMMON [/ [*bind-spec*] [*common-block-name*] /] □
□ *common-block-object-list* □
□ [[,] / [*bind-spec*] [*common-block-name*] / □
□ *common-block-object-list*]...

Global data objects of C having a type for which this Technical report defines a mapping to a Fortran datatype can then be accessed from Fortran as follows: A common block is declared with a *bind-spec* matching the C name of the C data object, and this common block holds a single Fortran data object with arbitrary local name and the suitable Fortran datatype. There shall be no definition for that common block in the Fortran program.

Editor's Note 22

Examples of C and Fortran data layout to be added...

A.1.2 Name binding for Fortran global entities

To facilitate interoperability of other languages with Fortran, the proposed name binding mechanism could be provided for all Fortran names that are *global entities*, not only for the declarations of *external procedure*s.

Name binding for *common block*s is specified above. Edits to implement name binding for *main program*, *module*, and *block data program unit* names are given below. They have no explicit use within a Fortran program, but may be used by other language processors to access these program units.

Editor's Note 23

Edits to allow name binding for the declaration and definition of all *external procedure*s and *external subprogram*s are to be provided.

Page 183
Subclause 11.1
Change line 14

R1102 *program-stmt* **is** PROGRAM *program-name*

to

R1102 *program-stmt* **is** [*bind-spec*] PROGRAM *program-name*

Page 184
Subclause 11.3
Change line 21

R1105	*module-stmt*	**is**	MODULE *module-name*

to

R1105	*module-stmt*	**is**	[*bind-spec*] MODULE *module-name*

Page 187
Subclause 11.4
Change line 3

R1111	*block-data-stmt*	**is**	BLOCK DATA *block-data-name*

to

R1111	*block-data-stmt*	**is**	[*bind-spec*] BLOCK DATA *block-data-name*

A.1.3 Support of unrestricted C pointers

Editor's Note 24

There were strong objections in WG5 against fully supporting the C pointer mechanisms. More work is needed to define what will be done here.

A.1.4 Support of the C `union` datatype

Editor's Note 25

There is currently no support for C `union` objects. One possible workaround would be to define a mapping of each component of a `union` to its Fortran equivalent, and then equivalencing these. I don't know if this will work. X3J3 proposed a CUNION statement for derived types, but this seems to have a too big impact on compilers.

A.2 Features planned for C9X

There are at least two mayor extensions planned for C9X which have an immediate impact on interoperability of Fortran and C. These are the introduction of complex numbers, and a new method to parametrize integer datatypes.

A.2.1 Complex C Extensions

Editor's Note 26

The "Complex C Extensions" are defined in an X3J11 Technical Report with that title, dated March 26, 1995. Document WG14/N470 X3J11/95-071 gives the C9X edits. Document WG14/N471 X3J11/95-072 defines `<complex.h>`. These documents are available at `ftp://ftp.dmk.com/dmk/sc22wg14/c9x/complex/`.

A.2.2 Big integers in C

Editor's Note 27

There are several documents on the subject in the directory `ftp://ftp.dmk.com/dmk/sc22wg14/c9x/extended-integers/`.