ISO/IEC JTC1/SC22/WG5 N1178
X3J3 / 96-069

International Standards Organization

Interoperability of Fortran and C

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

{Draft PDTR produced 24-Apr-96}

THIS PAGE TO BE REPLACED BY ISO CS

The most recent version of this working document may be obtained from
`http://www.uni-karlsruhe.de/~SC22WG5/TR-C/`

Comments may be sent to the Project Editor, `hennecke@rz.uni-karlsruhe.de`,
or to the email list `sc22wg5-interop@ncsa.uiuc.edu`.

Contents

Foreword

[This page to be provided by ISO CS]

Introduction

This Technical Report defines extensions to the programming language Fortran to permit Fortran programs to call C procedures and access C data objects with external linkage. 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 of Fortran and C

1 General

1.1 Scope

This Technical Report defines extensions to the programming language Fortran to permit Fortran programs to call C procedures and access C data objects with external linkage. 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 provides general mechanisms to map data types of C to Fortran. The second area addresses the calling conventions for a C procedure referenced in a Fortran program, and the third area provides access to global C data objects from within Fortran.

1.2 Organization of this Technical Report

This document is organized in four sections, covering general issues and the 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 would be necessary to incorporate these features in the Fortran standard.

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 form that a Fortran specification for a data object defined by means of C may take
3. The rules for interpreting the meaning of a reference to an external procedure or data object defined by means of C

1.4 Exclusions

This Technical Report does not specify:

1. Mixed-Language Input and Output
2. Methods to automatically convert C headers to Fortran
3. 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 entities 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.

Note 1.2

> 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 1

> 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 2

> Currently, ISO/IEC 1539-1:1996 means the proposed Fortran 95 DIS (WG5 document N1176), and references to ISO/IEC 9899:1990 are actually references to ANSI X3.159-1989.
> Non-normative reference is made to the draft C++ standard (WG21 document N0687), 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."

Editor's Note 3

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, more precisely to allow a Fortran program to reference C procedures and data objects that have external linkage.

3.1 The BIND attribute

The Fortran standard does not specify the mechanisms by which programs are transformed for use on computing systems (1.4). Additionally, a reference in a Fortran program to a procedure defined by means other than Fortran is normally made as though it were defined by an external subprogram (12.5.3).
This Technical Report defines a **BIND attribute**, which may be employed to adapt the behavior of the Fortran processor to the behavior of another processor, possibly for another language, in a portable way. The corresponding *bind-spec* specification may be used in all places where it is necessary to inform the Fortran processor that a change of processor dependent and language dependent conventions is required for the interoperability of Fortran and C. This section specifies the general form of a *bind-spec* specification.

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

Constraint: If *name-string* is present and *lang-keyword* is FORTRAN, the value of *name-string* shall be a valid Fortran name.

Constraint: If *name-string* is present and *lang-keyword* is C or C_STDARG, the value of *name-string* shall be a valid C *external name*.

The processor shall support at least those *lang-keywords* listed in Riop?, support of other *lang-keywords* is processor dependent. The processor shall report the use of unsupported *lang-keywords*.
BIND(FORTRAN) specifies the default behavior of the Fortran processor. The behavior for *lang-keywords* C and C_STDARG is defined in this Technical Report. The behavior for *lang-keywords* other than those listed in Riop? is processor dependent.

Note 3.1

Selecting the programming language C with the *lang-keyword* alone does not specify the implementation-defined and implementation-dependent behavior of the C processor, and specifying such information would in fact make the program unportable. The Fortran processor should be accompanied with documentation that states which C processor's conventions are followed.
If multiple C processors are supported, selection of a specific C processor should occur outside the Fortran program (e.g. by command-line arguments) rather than by introducing new *lang-keyword*s for nondefault C processors.

Note 3.2

Note that although names of C entities are normally case-sensitive, a C processor may ignore the distinction of alphabetic case of *external name*s. This limitation is implementation-defined.
A strictly conforming C program shall not rely on implementation-defined behavior, and a Fortran processor that does not support lowercase letters still conforms 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.

Editor's Note 4

C++ has a *linkage-specification* (7.5) which is very similar to the *bind-spec*, and requires the processor to support `"C"` and `"C++"`. However, C++ does not need a NAME= clause because C and C++ have the same (case-sensitive) rules for names.

The *bind-spec* may appear in a *derived-type-def*, as a *prefix-spec* or *attr-spec* within an interface block for an external procedure, or as an *attr-spec* in the specification of a data object in the *specification-part* of a module Since Fortran also provides specification statements for attributes, the *bind-attr* for external procedures and data entities may alternatively be specified by a **BIND statement**.

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

Constraint: A *bind-stmt* may only be specified in an *interface-body* or in the *specification-part* of a module.

The following sections describe the specific applications of the BIND attribute.

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 in sections 3.3 and 3.4.

Note 3.3

> 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.

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, access to the corresponding environmental limits is specified in section 3.2.2.

The remaining sections deal with some of C's *derived types*. The mechanisms defined in this Technical Report do not specify mappings for all possible C datatypes. Derived type generation in C can be recursively applied, the resulting types do not necessarily have a general approximation in Fortran 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 3.4

> 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.

```
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_ushrt_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 of C_CHAR_KC shall be the value of C_SCHAR_KC or the value of C_UCHAR_KC, this is processor-dependent.

Note 3.5

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 5

If `enums` 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 be 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 3.6

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 = RADIX      (0.0_c_flt_kr)
  INTEGER, PARAMETER ::    FLT_MANT_DIG = DIGITS     (0.0_c_flt_kr)
  INTEGER, PARAMETER ::    DBL_MANT_DIG = DIGITS     (0.0_c_dbl_kr)
  INTEGER, PARAMETER ::   LDBL_MANT_DIG = DIGITS     (0.0_c_ldbl_kr)
  INTEGER, PARAMETER ::         FLT_DIG = PRECISION  (0.0_c_flt_kr)
  INTEGER, PARAMETER ::         DBL_DIG = PRECISION  (0.0_c_dbl_kr)
  INTEGER, PARAMETER ::        LDBL_DIG = PRECISION  (0.0_c_ldbl_kr)
  INTEGER, PARAMETER ::     FLT_MIN_EXP = MINEXPONENT(0.0_c_flt_kr)
  INTEGER, PARAMETER ::     DBL_MIN_EXP = MINEXPONENT(0.0_c_dbl_kr)
  INTEGER, PARAMETER ::    LDBL_MIN_EXP = MINEXPONENT(0.0_c_ldbl_kr)
  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 = MAXEXPONENT(0.0_c_flt_kr)
  INTEGER, PARAMETER ::     DBL_MAX_EXP = MAXEXPONENT(0.0_c_dbl_kr)
  INTEGER, PARAMETER ::    LDBL_MAX_EXP = MAXEXPONENT(0.0_c_ldbl_kr)
  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     = HUGE   (0.0_c_flt_kr)
  REAL(c_dbl_kr),  PARAMETER ::  DBL_MAX     = HUGE   (0.0_c_dbl_kr)
  REAL(c_ldbl_kr), PARAMETER :: LDBL_MAX     = HUGE   (0.0_c_ldbl_kr)
  REAL(c_flt_kr),  PARAMETER ::  FLT_EPSILON = EPSILON(0.0_c_flt_kr)
  REAL(c_dbl_kr),  PARAMETER ::  DBL_EPSILON = EPSILON(0.0_c_dbl_kr)
  REAL(c_ldbl_kr), PARAMETER :: LDBL_EPSILON = EPSILON(0.0_c_ldbl_kr)
  REAL(c_flt_kr),  PARAMETER ::  FLT_MIN     = TINY   (0.0_c_flt_kr)
  REAL(c_dbl_kr),  PARAMETER ::  DBL_MIN     = TINY   (0.0_c_dbl_kr)
  REAL(c_ldbl_kr), PARAMETER :: LDBL_MIN     = TINY   (0.0_c_ldbl_kr)
END MODULE iso_c_float_h
```

Editor's Note 6

C's and Fortran's floating point number models are identical. I have not yet tracked down the relation of RANGE and `*_MIN_10_EXP` / `*_MAX_10_EXP`.

```
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 *c-kind-param* defined in ISO_C_KINDS has a negative value, the processor need not provide constants defined in ISO_C_LIMITS_H and ISO_C_FLOAT_H which use that *c-kind-param* as a *kind-param*. In this case, it is processor-dependent whether the names of such constants are accessible (with another kind type parameter supported by the processor) or not.

3.2.3 Mapping C array types to Fortran

An *array type* in C with an *element type* for which this Technical Report establishes a corresponding Fortran type can be mapped to Fortran by specifying the DIMENSION attribute for that type. If the entity concerned is a dummy argument, the *array-spec* shall be an *explicit-shape-spec-list* or an *assumed-size-spec*. Otherwise, it shall be an *explicit-shape-spec-list*.

Note 3.7

This rule includes the common case of a C array of unknown size which is initialized: the declaration

```
int x[] = { 1, 3, 5 };
```

defines `x` as a one-dimensional array of initially incomplete type, but at the end of the *initializer-list* it has no longer incomplete type but a size of three elements.

Note 3.8

C guarantees a minimum of 12 array (or pointer or function) declarators, whereas Fortran only supports 7 array dimensions. However, this limit will be seldom reached for actual C code. For dummy arguments it can be circumvented by the use of an *assumed-size-spec*.

Because the array element ordering (6.2.2.2) of Fortran arrays is reverse to the array subscripting of C arrays, the extents entering the Fortran *array-spec* shall be specified in the reverse order of the corresponding C array declarators.

Note 3.9

For one-dimensional arrays there is no difference between Fortran and C. If required, conversion of two-dimensional arrays can be performed by the intrinsic procedure TRANSPOSE (13.14.111). For higher-dimensional arrays this transposition must be done by the user.

C and Fortran have different concepts of character strings, so C character strings shall not be mapped to a CHARACTER array using the DIMENSION attribute. Section 3.2.7 defines the mapping of C character strings to Fortran.

3.2.4 Mapping C structure types to Fortran

A *structure type* in C with member objects which all have a type for which this Technical Report establishes a corresponding Fortran type can be mapped to Fortran by using a derived type definition. To ensure that the memory layout of the Fortran derived type matches the layout of the C `struct`, the BIND(C) attribute shall be specified in the *derived-type-def*.

Note 3.10

The *type-name* need not correspond to the tag of the C `struct` because both are local to their respective scoping units. Consequently, a NAME= clause in a BIND(C) specification within a derived type definition is not allowed.

The order of the *component-def-stmt*s shall be identical to the order of the corresponding *struct-declaration-list*. A *component-initialization* shall not be specified for derived types that have the BIND(C) attribute.

Note 3.11

> The POINTER *component-attr-spec* is not allowed because there is no C type whose corresponding Fortran type has the POINTER attribute.
> Similarly, C `struct`s that include *bit-field*s cannot be mapped to Fortran because this Technical Report does not specify mappings for bit-fields. The behavior for a Fortran derived type in which bit-field member objects are mapped to objects of integer type is processor dependent, because the memory layout (alignment, padding) of such derived types may differ from the layout of the original C `struct`.

3.2.5 Mapping C union types to Fortran

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

Editor's Note 7

> The user may specify such mappings "manually" by specifying separate derived types for each member object (as if that member object were the only member of a `struct`), declaring the data object with the largest of these types, and using TRANSFER to convert between the member object types. Another "hack" would be to declare different data objects with these separate types, but bind them all to the same C data object by using identical NAME= clauses of their BIND(C) specs. This does not work for dummy arguments.

3.2.6 Handling of C pointer declarators

This Technical Report does not provide features to map general C pointers to Fortran. However, several special cases are supported: Within an explicit interface that has the BIND(C) or BIND(C_STDARG) attribute,

- a dummy argument with C type "array of T" is equivalent to type "pointer to T". This case is supported by specifying the DIMENSION attribute for the dummy argument.

- a dummy argument which has C type "pointer to T" because the procedure modifies the scalar argument of type T is supported by specifying the BYREFERENCE attribute for the dummy argument.

- a dummy argument may be a dummy procedure that has an explicit interface and the BIND(C) or BIND(C_STDARG) attribute. This case shall be mapped by the Fortran processor to a C type "pointer to function", with a C return type and C arguments derived from the dummy procedure's interface body specifications.

Editor's Note 8

A BYREFERENCE attribute is not yet defined in this TR, but this functionality is necessary to support "call by reference".
This may be implemented by redefining INTENT semantics within a BIND(C) interface, so that no INTENT or INTENT(IN) implies call by value, whereas INTENT(OUT) or INTENT(INOUT) implies call by reference. INTENT(IN) for arrays should not imply call by value but rather mimic the `const` qualifier of a C array argument.

Additionally, a derived type with *type-name* C_VOID_PTR shall be supported, this type shall have the BIND(C) attribute and PRIVATE components.This type shall be mapped by the Fortran processor to the C type "pointer to `void`".
Pointers to `void` and all other C pointer types which have the same representation can be mapped to this type, this applies to function results and dummy arguments as well as to `struct` member objects. The behavior for cases where the C pointer type has a representation different from "pointer to `void`" is processor dependent.

3.2.7 Mapping C character strings to Fortran

The processor shall provide an intrinsic module ISO_C_STRINGS, which shall provide access to three derived type definitions with *type-names* C_CHAR_STRING, C_SCHAR_STRING, and C_UCHAR_STRING. They shall have the BIND(C) attribute and PRIVATE components.
These types shall be used to map C character strings which are dummy arguments of a procedure with the BIND(C) or BIND(C_STDARG) attribute to Fortran, as specified in section 3.3 of this Technical Report. They may also be used to access C character strings which are data objects with external linkage defined in a C translation unit, as specified in section 3.4 of this Technical Report.

The module ISO_C_STRINGS shall also provide the following:

- ASSIGNMENT(=) for the following combinations of *variable* and *expr*:

Type of *variable*	Type of *expr*
TYPE(c_char_string)	TYPE(c_char_string)
TYPE(c_char_string)	CHARACTER(KIND=c_char_kc)
CHARACTER(KIND=c_char_kc)	TYPE(c_char_string
TYPE(c_schar_string)	TYPE(c_schar_string)
TYPE(c_schar_string)	CHARACTER(KIND=c_schar_kc)
CHARACTER(KIND=c_schar_kc)	TYPE(c_schar_string
CHARACTER(KIND=c_uchar_kc)	TYPE(c_uchar_string
TYPE(c_uchar_string)	TYPE(c_uchar_string)
TYPE(c_uchar_string)	CHARACTER(KIND=c_uchar_kc)

  Assignments among objects of the same TYPE shall copy the contents of *expr* to *variable*, up to and including the first ASCII `NUL` character.
  Assignment of a CHARACTER *expr* to its corresponding TYPE shall copy

the value of *expr* and append an ASCII `NUL` character.
Assignment to a CHARACTER *variable* from its corresponding TYPE shall copy the contents of *expr* to *variable*, up to and excluding the first ASCII `NUL` character; if the lengths of *expr* and *variable* do not agree, the rules for intrinsic character assignment apply.

- Extension of the generic interface for the LEN intrinsic function (13.14.54): LEN shall accept a scalar argument of TYPE(c_char_string), TYPE(c_schar_string) or TYPE(c_uchar_string); the result value for these arguments shall be the number of characters in the internal representation of these types, up to and excluding the first ASCII `NUL` character.

Editor's Note 9

> The concatenation and comparison operators may also be extended, as well as some intrinsic procedures for character processing (and perhaps type conversion procedures from CHARACTER to these types).

3.2.8 Mapping of C `typedef` names

In C, a declaration whose storage-class specifyer is `typedef` can be used to define identifiers that name types. These *typedef-names* do not introduce new types, only synonyms for types that could be specified in another way. They may be used as *type-specifiers*. This Technical Report introduces a *type-alias-stmt*, which is a *declaration-construct*, to allow similar type name aliasing in Fortran.

Riop? *type-alias-stmt* **is** TYPE [[, *access-spec*] ::] □
□ *type-alias-name* => *type-spec*

Constraint: An *access-spec* is only allowed if the *type-alias-stmt* is within the *specification-part* of a module.

Constraint: A *type-name* shall not be the same as the name of any intrinsic type defined in IS 1539 nor the same as any accessible *type-name* or *type-alias-name*.

The *type-alias-name* declared in a *type-alias-stmt* can be used interchangeable with the corresponding *type-spec*: entities declared with TYPE(*type-alias-name*) have the same type as if they were declared with the corresponding *type-spec*.

Note 3.12

> For derived type *type-names*, this is similar to a *rename* of the name in a USE statement. The *type-alias-stmt* is more general in that it also allows aliasing intrinsic types, and is not limited to the USE statement.

Note 3.13

Example:
The Xlib application programming interface includes a type `Window`. It is defined in `<X11/Xlib.h>`, by the following `typedef`s:

```
typedef unsigned long XID;
typedef XID Window;
```

Rather than directly using an INTEGER(C_ULONG_KI) *type-spec* in the application program, these details may be hidded by declaring type aliases

```
TYPE XID => INTEGER(c_ulong_ki)
TYPE Window => TYPE(XID)
```

for the above `typedef`s and using TYPE(Window) as the *type-spec*.

3.2.9 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

This section defines mechanisms to instruct the Fortran processor to follow the calling conventions of the processor designated by the *lang-keyword*s C and C_STDARG when an external procedure defined by means of C is referenced. An explicit interface for that procedure shall be accessible in all scoping units containing a procedure reference that should follow these modified calling conventions. The corresponding *interface-body* shall contain a *bind-spec* specification with *lang-keyword* C or C_STDARG.

If a C function's return type is `void`, the Fortran interface for such a function shall specify a subroutine. If a C function returns a type other than `void` for which this Technical Report establishes a corresponding Fortran type, the Fortran interface shall specify a function with that type. In all other cases, the Fortran interface may specify a function but the behavior is processor dependent.

If the *bind-spec* does not specify a *name-string*, the *function-name* or *subroutine-name* is used to generate an external entry for the procedure, using the Fortran processor's conventions (this implies ignoring alphabetic case of the name). If a *name-string* is specified, the external entry is generated using the C processor's conventions, as if the value of the *name-string* were a C identifier with external linkage.

3.3.1 Procedure interface for BIND(C) binding

The *interface-body* that specifies a Fortran interface to a C procedure shall specify dummy arguments that correspond by position with the arguments of the C procedure, and have a Fortran type that corresponds to the type of the C procedure argument as specified in section 3.2 of this Technical Report. If the argument of a C procedure has type "function returning T" (or "pointer to function returning T" ?), the Fortran interface shall specify a dummy procedure. There shall be an explicit interface for the dummy procedure in the *specification-part* of the *interface-body*, that interface shall specify the BIND(C) attribute.

The following additional rules apply for the specification of the procedure interface:

- The POINTER and TARGET *attr-spec*s shall not appear
- INTENT other than IN shall not be specified for dummy arguments which are passed according to the C default conventions (call by value)
- If a dummy argument is of derived type, that type shall have the BIND(C) attribute
- A dummy argument shall not have type COMPLEX or LOGICAL
- OPTIONAL shall not be specified
- A dummy procedure shall have an explicit interface, and that interface shall specify the BIND(C) attribute

In a reference to a procedure with the BIND(C) attribute, all scalar dummy arguments that do not have the BYREFERENCE attribute imply that the actual argument is passed by value. All actual arguments that have the DIMENSION or BYVALUE attribute are passed by reference. The processor shall generate a C function type for actual arguments that are associated with dummy procedures, using the specifications of the dummy procedure's explicit interface.

Editor's Note 10

See the Editor's note in section 3.2.6 concerning BYREFERENCE. Keyword arguments should be allowed. PURE should be allowed, if the C procedure is pure. ELEMENTAL reference may be allowed, this may need some wording. RECURSIVE is allowed and has no effect.

Note 3.14

Examples of bindings to the C routine `double MPI_Wtime(void)`:

```
INTERFACE
 FUNCTION MPI_WTIME1 ( )
  USE iso_c, ONLY: c_dbl_kr
  REAL(c_dbl_kr), BIND(C,"MPI_Wtime") :: MPI_WTIME1
 END FUNCTION MPI_WTIME1

 BIND(C,"MPI_Wtime") FUNCTION MPI_WTIME2 ( )
  USE iso_c, ONLY: c_dbl_kr
  REAL(c_dbl_kr) MPI_WTIME2
 END FUNCTION MPI_WTIME2

 REAL(c_dbl_kr) FUNCTION MPI_WTIME3 ( )
  USE iso_c, ONLY: c_dbl_kr
  BIND(C,"MPI_Wtime") :: MPI_WTIME3
 END FUNCTION MPI_WTIME3
END INTERFACE
```

The kind value `c_dbl_kr` is defined in section 3.2. Note that in the last *interface-body*, it is also accessible in the *function-stmt*.

3.3.2 Procedure interface for BIND(C_STDARG) binding

The C_STDARG *language-keyword* can be used to specify the binding to a C procedure that utilizes C variable argument lists, as defined in the ISO C header `<stdarg.h>`.

If C_STDARG binding is specified, the behavior is as if C binding were specified, except for the following rules for the specification of the procedure interface:

- The interface shall specify at least one non-optional dummy argument

- The OPTIONAL attribute on dummy arguments that are not dummy procedures is allowed
- In the *dummy-arg-name-list*, all OPTIONAL dummy arguments shall be specified after all non-optional dummy arguments

If a procedure interface specifies BIND(C_STDARG) binding, the semantics of a call to this procedure are changed so that all non-optional arguments are passed according to the conventions of the C processor (like for BIND(C) binding specified above), all PRESENT optional paramenters are passed in a way the `<stdarg.h>` macros can handle, and nothing is passed for those optional parameters that are not PRESENT. The last non-optional parameter specifies the offset for the `va_start` macro.

3.4 Access to global C data objects

This section defines mechanisms to reference global data objects that are defined in C translation units from within a Fortran program.

To access a C data object of type T with external linkage from within Fortran, a Fortran variable with the Fortran type corresponding to T (as specified in section 3.2 of this Technical Report) shall be declared in a module, and may then be accessed within the module and all other scoping units that contain a module reference for that module.
To specify that the storage for the Fortran variable is reserved by the C translation unit, the BIND(C) attribute shall be specified, with a *name-string* whose value is the identifier of the C data object. The following additonal restrictions apply:

- The BIND(C) attribute for a module variable implies the SAVE attribute
- No *initialization* shall appear in the *entity-decl*
- PARAMETER, POINTER or TARGET shall not be specified
- Appearance of a data entity having the BIND(C) attribute as a *common-block-object* is prohibited.
- For a given *name-string*, there shall be at most one Fortran variable with the BIND(C,*name-string*) attribute in the program.
- The *name-string* is a global name. The rules of section 14.1 apply.

Editor's Note 11

This is a very preliminary specification. Some more work is needed, especially to avoid unintended storage association.

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

This section contains 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.

Page xiv
Line 24
Update the "Organization of this International Standard" subclause.

Page 2
Subclause 1.5
Conformance paragraph at line 39 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

R207	*declaration-construct*	**is**	*derived-type-def*
		or	*interface-block*
		...	

add after line 7:

		or	*type-alias-stmt*

Page 10
Subclause 2.1
In

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

add after line 32:

		or	*bind-stmt*

Page 19
Subclause 2.5.7
In line 6, change "procedures" to "procedures, modules".
After line 8, 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 38
Subclause 4.4.1
In

R424	*private-sequence-stmt*	**is**	PRIVATE
		or	SEQUENCE

add after line 31:

		or	*bind-spec*

Editor's Note 12

> Also do a global rename of *private-sequence-stmt* to *derived-type-body-stmt*.

Page 38
Subclause 4.4.1
In the Constraints list, add after line 37:

Constraint: If a *bind-spec* is present, it shall not contain a *name-string* and *lang-keyword* shall not be C_STADRG.

Constraint: If a *bind-spec* is present, all derived types specified in component definitions shall have that BIND attribute.

Page 39
Subclause 4.4.1
In the Constraints list, add after line 29:

Constraint: *component-initialization* shall not appear if a *bind-spec* is present in the derived type definition.

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 18:

Constraint: A *bind-spec* may only be specified in an *interface-body* or in the *specification-part* of a module.

Page 48
Subclause 5.1
After the Constraints list, add after line 41:

> If a *bind-spec* is present, the additional constraints of section 16.x apply.

Page 53
Subclause 5.1.2
After section 5.1.2.2, insert a new section after line 5:

> **5.1.2.2a BIND attribute**
>
> The **BIND attribute** specifies that mechanisms for interoperability with other languages are used. Binding to entities that are defined by means of ISO C and have external linkage is described in section 16. This attribute may also be declared via the BIND statement (16.x.y).

Editor's Note 13

> Maybe dd a note explaining that there is another usage of BIND: in *derived-type-def* s.

Page 57
Subclause 5.2
At line 41, change

> This also applies to EXTERNAL and INTRINSIC statements.

to

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

Editor's Note 14

> Maybe something in 6.3 (dynamic association) for dealing with C dynamic memory in BIND(C) structures, like the C string datatypes...

Editor's Note 15

> Maybe something in section 7 for the C string operations...

Page 193
Subclause 12.3.1.1
After line 15, add a new clause to the list (1):

> (f) That should follow other than the processor's default calling conventions (16.x).

Page 207
Subclause 12.5.2.2
In

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

add after line 1:

		or	*bind-spec*

Page 207
Subclause 12.5.2.2
In the Constraints list following R1219, add after line 5:

Constraint: A *bind-spec* may only be specified in an *interface-body*.

Page 208
Subclause 12.5.2.2
Add after line 4:

> If a *bind-spec* is present in the *prefix* or *specification-part* of the function, the additional constraints of section 16.x apply.

Page 208
Subclause 12.5.2.3
After R1123 at line 34, add:

Constraint: If a *bind-spec* is present in the *specification-part* of the subroutine, * shall not appear as *dummy-arg*.

Page 209
Subclause 12.5.2.3
Add after line 4:

> If a *bind-spec* is present in the *specification-part* of the subroutine, the additional constraints of section 16.x apply.

Page 211
Subclause 12.5.3
After "external subprogram on line 14, add ", except when the binding mechanisms described in section 16 are used".

Page 275
Subclause 14.1
"Scope of names" may be affected.

Page 292
New clause 16
Introduce a new section 16 (Interoperability with ISO C).

Editor's Note 16

> 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 293
Annex A
Update the Glossary:
After 293:39, add the term **binding** with a definition.
After 294:6, add the term **calling conventions** with a definition.

Page 309
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 334+ may be affected.

Page 347
Annex D
Update the Index :-)