ISO/IEC JTC1/SC22/WG5 N1237
X3J3 / 97-108

International Organization for Standardization

Interoperability of Fortran and C

Technical Report defining extensions to
ISO/IEC 1539-1 : 1997

{Third Draft PDTR produced 1997–01–12}

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 reference C functions and C data objects with external linkage. The current Fortran language is defined by the International Standard ISO/IEC 1539-1:1997, 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 these features 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 reference C functions and C data objects with external linkage. The current Fortran language is defined by the International Standard ISO/IEC 1539-1:1997, 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 function 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). 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 1539-1:1997 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
4. A complete mapping of all C types to Fortran types. Notably, the C enumerated types, union types, and some pointer types are not supported.

1.5 Conformance

The language extensions defined in this Technical Report are implemented by defining a small 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.

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) are followed.

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 : 1997	*Information Technology – Programming Languages – Fortran*
ISO/IEC 9899 : 1990	*Information Technology – Programming Languages – C*

Note 1.3

Currently, ISO/IEC 1539-1:1997 means ISO/IEC DIS 1539-1:1996.

2 Rationale

A significant fraction of the standard (de-facto or de-jure) computing environment comes with a programming interface for the C language, ISO/IEC 9899:1990. Examples include

- the standard library of the ISO C language itself;
- the System Application Program Interface (API) of the Portable Operating System Interface (POSIX), ISO/IEC 9945-1:1990;
- the X Window System of the X Consortium, including the X Library Xlib and X Toolkit Intrinsics Xt, and Motif;
- the Distribute Computing Environment (DCE) of the Open Software Foundation (OSF), including among others the DCE remote procedure calls;
- visualization and graphics packages;
- low-level mechanisms for interprocess communication (IPC) like semaphores, shared memory mechanisms, and TCP/IP sockets.

Many of these components do not have a Fortran programming interface, and the Fortran programmer is currently unable to exploit this wealth of software in a portable way. Recoding this existing technology in Fortran is often impossible for technical as well as economical reasons.

This causes many problems for those programmers who wish or need to interface their numerically intensive applications, written in Fortran, to the existing computing environment: Many large applications require access to operating system facilities or C-style stream I/O, rely on client/server technologies, or need Graphical User Interfaces (GUIs) and visualization capabilities. Due to the difficulties of providing a Fortran interface to C software in a standard fashion, many Fortran users are turning to alternative languages for such applications, even if Fortran is ideally suited to the "computational" component of their tasks. It is therefore very important that a standard mechanism by which such C programming interfaces can be accessed is defined as soon as possible.

Although it cannot be expected that every single feature of the C language can be made accessible to Fortran, the core requirement is that a Fortran program should be able to portably reference any external C function whose return type and parameter types have a general approximation to Fortran data types, and should also be able to portably access external C data objects of such types.

Note 2.1

> In cases where such type mapping is not easily achieved, it should be possible to portably reference a series of external C functions and pass non-Fortran-like arguments between them, or to write C stub functions that do the mapping to types supported by the Fortran interface and reference these stub functions in a portable way.

If this goal is achieved, a large portion of C code which is often separately compiled and packaged in object code libraries should be accessible from Fortran.

Another factor very important to the usability of the Fortran interface is the ability to deal with C's concept of headers: most C packages have an API which declares function prototypes, derived types, type names, macros, and external variables that can be accessed by the application through the inclusion of C headers. Although no attempt is made to automatically convert C headers to Fortran, it should be possible that either the provider of the C package or the user creates Fortran modules which contain the Fortran counterparts of the header contents (explicit procedure interfaces, type definitions, type alias names, named constants and module procedures corresponding to the C macros, and module variables which bind to the external C variables).

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 functions and data objects that have *external linkage*.

Note 3.1

The Edits of section 4 shall also be in effect.

3.1 Intrinsic modules and C standard headers

This Technical Report defines various named constants, derived type definitions, operations on these derived types, and other procedures to support some features of the ISO C language which are not an integral part of Fortran.

Access to all such entities defined in this Technical Report shall be provided by an **intrinsic module** ISO_C, with the following exceptions:

- The environmental limits specified in the headers `<limits.h>` and `<float.h>` shall be made accessible to the Fortran program through two intrinsic modules ISO_C_LIMITS_H and ISO_C_FLOAT_H, as described in section 3.3.2.

- Support for C variable argument lists `<stdarg.h>` shall be provided through an intrinsic module ISO_C_STDARG_H, as described in section 3.4.3.

Note 3.2

These three standard headers are also required by the C standard for a freestanding implementation. The fourth header required for a freestanding implementation, `<stddef.h>`, is not required by this Technical Report: Apart from the macro `NULL` for which a constant C_NULL is made accessible in the intrinsic module ISO_C, this header contains only entities which are not directly supported by this Technical Report.

Apart from these intrinsic modules, the Fortran processor is not required to support any of the C standard headers. However, if a Fortran processor does support any C standard header, it shall provide access to those facilites by additional intrinsic modules. The following module names shall be used:

C standard header	Fortran intrinsic module
<errno.h>	ISO_C_ERRNO_H
<stddef.h>	ISO_C_STDDEF_H
<assert.h>	ISO_C_ASSERT_H
<ctype.h>	ISO_C_CTYPE_H
<locale.h>	ISO_C_LOCALE_H
<math.h>	ISO_C_MATH_H
<setjmp.h>	ISO_C_SETJMP_H
<signal.h>	ISO_C_SIGNAL_H
<stdio.h>	ISO_C_STDIO_H
<stdlib.h>	ISO_C_STDLIB_H
<string.h>	ISO_C_STRING_H
<time.h>	ISO_C_TIME_H
<iso646.h>	ISO_C_ISO646_H
<wctype.h>	ISO_C_WCTYPE_H
<wchar.h>	ISO_C_WCHAR_H

In each of these modules, an implementation may support all or parts of the contents of the corresponding C standard header. These modules may also provide access to additional entities not specified in the ISO C standard.

Note 3.3

The last three standard headers are defined in Amendment 1: "C Integrity" to the C standard, ISO/IEC 9899:1990/Amd.1:1995.

3.2 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.

R1601 *bind-spec* **is** BIND ([LANG=] *lang-keyword* □
□ [, [NAME=] *name-string*] □
□ [, PRAGMA=*pragma-string*])

R1602 *lang-keyword* **is** FORTRAN
or C

R1603 *name-string* **is** *scalar-default-char-init-expr*

R1604 *pragma-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, the value of *name-string* shall be a valid C *external name*.

The processor shall support at least those *lang-keyword*s listed in R1602, support of other *lang-keyword*s is processor dependent. The processor shall report the use of unsupported *lang-keyword*s.

The term "BIND(*lang-keyword*) attribute" denotes the BIND attribute with the given *lang-keyword*, this term does not imply presence or absence of a *name-string* or *pragma-string*.

BIND(FORTRAN) specifies the default behavior of the Fortran processor. The behavior for *lang-keyword*s C is defined in this Technical Report. The behavior for *lang-keyword*s other than those listed in R1602 is processor dependent. The interpretation of *pragma-string*s is processor dependent.

Note 3.4

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 3.5

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.

Note 3.6

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, and `#pragma`s are processed by the C/C++ preprocessor.

The *bind-spec* may appear in a *derived-type-def*, as a *prefix-spec* within an interface body 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-spec* for data objects in a module may alternatively be specified by a **BIND statement**.

R1605 *bind-stmt* **is** *bind-spec* [::] *extern-name*

Constraint: A *bind-stmt* may only be specified for variables in the *specification-part* of a module.

Section 3.3.4 describes the use of a *bind-spec* to map C structure types to Fortran, section 3.4 shows how to use a *bind-spec* in explicit procedure interfaces to map C function prototypes to Fortran, and section 3.5 explains how to bind to C data objects with external linkage by using a *bind-spec*.

3.3 Datatype mapping

This section describes the mapping of C *object type*s to Fortran types. To specify C's *function type*s, a Fortran program shall use explicit procedure interfaces with a BIND(C) attribute, as described in section 3.4. The only C *incomplete type*s which are supported are C function parameters of unknown size. These are mapped to assumed size dummy arrays, see section 3.3.6.

Note 3.7

The incomplete type `void` is not supported (except when it denotes an empty *parameter-type-list*). However, the types "pointer to `void`" and "function returning `void`" derived from it are supported, see sections 3.3.7 and 3.4.

Both languages define object 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.

Note 3.8

The C enumerated types declared with the *type specifier* `enum` are not specified to be basic types in the C standard (they are *integral type*s, but not *integer type*s), but neither are they specified to be derived types. Section 3.3.3 addresses C enumerated types.

Section 3.3.1 specifies a complete mapping of C *basic types* to Fortran types, access to the corresponding environmental limits is specified in section 3.3.2. 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, not all of the resulting types have a general approximation in Fortran types.

3.3.1 Matching C basic types with Fortran intrinsic types

The *basic types* of C are the type `char`, and the *integer types* and *floating types*. 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: The intrinsic module ISO_C shall define a *c-kind-param*, which is a default integer constant, for each basic type. Their names shall be as given in the table below. If the processor supports a C basic type, the corresponding *c-kind-param* shall be the suitable *kind-param* supported by the processor. Otherwise it shall be a negative default integer constant. The following table shows the correspondence of C basic types and Fortran intrinsic types of suitable *c-kind-param*. The character length of the CHARACTER(KIND=C_CHAR) type which matches the C type `char` shall be one.

C basic type	Fortran intrinsic type
`char`	CHARACTER(KIND=C_CHAR)
`signed char`	INTEGER(C_SCHAR)
`short`	INTEGER(C_SHRT)
`int`	INTEGER(C_INT)
`long`	INTEGER(C_LONG)
`unsigned char`	INTEGER(C_UCHAR)
`unsigned short`	INTEGER(C_USHRT)
`unsigned int`	INTEGER(C_UINT)
`unsigned long`	INTEGER(C_ULONG)
`float`	REAL(C_FLT)
`double`	REAL(C_DBL)
`long double`	REAL(C_LDBL)

Equivalent combinations of C *type-specifier*s are all mapped to the same Fortran type kind, e.g. `long`, `long int`, `signed long` and `signed long int` all correspond to INTEGER(C_LONG).

Note 3.9

The mnemonic names for the Fortran type kind parameters follow the same naming conventions to identify the corresponding types that are used in the C standard headers `<limits.h>` and `<float.h>`.

Note 3.10

In C, the question if `char` is implemented as `signed char` or `unsigned char` is implementation-defined. Only the types explicitly specified as `signed` or `unsigned` are *integer types*, the type `char` is not. Although all three are *character types*, this Technical Report only maps the type `char` to a Fortran character type.

Since Fortran does not support unsigned integer types and the unsigned integer C types have the same size and alignment requirements as their signed counterparts, the kind type parameters for unsigned integer types shall have the same value as those for the corresponding signed types. The interpretation of negative values in a context which requires unsigned integer C values is processor dependent.

Note 3.11

Passing unsigned integer values returned by a C function to another C function is possible by using variables of the corresponding signed integer Fortran type. The result values of Fortran's bit manipulation intrinsic procedures can be assigned to variables which match an unsigned integer C type, since Fortran's bit model (13.5.7) matches C's unsigned integer model. The range of non-negative values of a signed integer C type is a subset of the values of the corresponding unsigned integer C type, and the representation of these values is identical. So all operations and assignments of values which stay in the non-negative range of values of a signed integer type are well-defined in a context which requires the corresponding unsigned integer type.

Note 3.12

If the Fortran processor supports the non-standard C integer type `long long`, the recommended name for the corresponding Fortran kind type parameter is C_LONG_LONG.

3.3.2 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>`. The Fortran processor shall provide two modules ISO_C_LIMITS_H and ISO_C_FLOAT_H, which shall make these limits available in Fortran through constants having the same names as those defined in these standard headers. The types and type parameters of these constants shall be as in the model implementations given below.

A model implementation of the module holding the numerical limits from the standard header `<limits.h>` could be (on a two's complement machine and with `char` being implemented as `signed char`):

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

  INTEGER,          PARAMETER ::   CHAR_BIT =           8
  INTEGER(c_schar), PARAMETER ::  SCHAR_MIN =        -128_c_schar
  INTEGER(c_schar), PARAMETER ::  SCHAR_MAX =         127_c_schar
  INTEGER(c_uchar), PARAMETER ::  UCHAR_MAX =   SCHAR_MIN
  INTEGER,          PARAMETER ::   CHAR_MIN =   SCHAR_MIN
  INTEGER,          PARAMETER ::   CHAR_MAX =   SCHAR_MAX
  INTEGER,          PARAMETER :: MB_LEN_MAX =           1
  INTEGER(c_shrt),  PARAMETER ::   SHRT_MIN =      -32768_c_shrt
  INTEGER(c_shrt),  PARAMETER ::   SHRT_MAX =       32767_c_shrt
  INTEGER(c_ushrt), PARAMETER :   USHRT_MAX =    SHRT_MIN
  INTEGER(c_int),   PARAMETER ::    INT_MIN =      -32768_c_int
  INTEGER(c_int),   PARAMETER ::    INT_MAX =       32767_c_int
  INTEGER(c_uint),  PARAMETER ::   UINT_MAX =     INT_MIN
  INTEGER(c_long),  PARAMETER ::   LONG_MIN = -2147483648_c_long
  INTEGER(c_long),  PARAMETER ::   LONG_MAX =  2147483647_c_long
  INTEGER(c_ulong), PARAMETER ::  ULONG_MAX =    LONG_MIN
END MODULE iso_c_limits_h
```

A possible implementation of the module holding the numerical limits from the standard header `<float.h>` could be:

```
MODULE iso_c_float_h  !  F95 module for C89 <float.h>
  USE iso_c, ONLY: c_flt, c_dbl, c_ldbl
  IMPLICIT NONE

  INTEGER, PARAMETER ::      FLT_ROUNDS =  -1  !  indeterminable

  INTEGER, PARAMETER ::       FLT_RADIX = RADIX      (0.0_c_flt)
  INTEGER, PARAMETER ::    FLT_MANT_DIG = DIGITS     (0.0_c_flt)
  INTEGER, PARAMETER ::    DBL_MANT_DIG = DIGITS     (0.0_c_dbl)
  INTEGER, PARAMETER ::   LDBL_MANT_DIG = DIGITS     (0.0_c_ldbl)
  INTEGER, PARAMETER ::         FLT_DIG = PRECISION  (0.0_c_flt)
  INTEGER, PARAMETER ::         DBL_DIG = PRECISION  (0.0_c_dbl)
  INTEGER, PARAMETER ::        LDBL_DIG = PRECISION  (0.0_c_ldbl)
  INTEGER, PARAMETER ::     FLT_MIN_EXP = MINEXPONENT(0.0_c_flt)
  INTEGER, PARAMETER ::     DBL_MIN_EXP = MINEXPONENT(0.0_c_dbl)
  INTEGER, PARAMETER ::    LDBL_MIN_EXP = MINEXPONENT(0.0_c_ldbl)
  INTEGER, PARAMETER ::  FLT_MIN_10_EXP = -1*RANGE   (0.0_c_flt)
  INTEGER, PARAMETER ::  DBL_MIN_10_EXP = -1*RANGE   (0.0_c_dbl)
  INTEGER, PARAMETER :: LDBL_MIN_10_EXP = -1*RANGE   (0.0_c_ldbl)
  INTEGER, PARAMETER ::     FLT_MAX_EXP = MAXEXPONENT(0.0_c_flt)
  INTEGER, PARAMETER ::     DBL_MAX_EXP = MAXEXPONENT(0.0_c_dbl)
  INTEGER, PARAMETER ::    LDBL_MAX_EXP = MAXEXPONENT(0.0_c_ldbl)
  INTEGER, PARAMETER ::  FLT_MAX_10_EXP = RANGE      (0.0_c_flt)
  INTEGER, PARAMETER ::  DBL_MAX_10_EXP = RANGE      (0.0_c_dbl)
  INTEGER, PARAMETER :: LDBL_MAX_10_EXP = RANGE      (0.0_c_ldbl)

  REAL(c_flt),  PARAMETER ::  FLT_MAX     = HUGE   (0.0_c_flt)
  REAL(c_dbl),  PARAMETER ::  DBL_MAX     = HUGE   (0.0_c_dbl)
  REAL(c_ldbl), PARAMETER :: LDBL_MAX     = HUGE   (0.0_c_ldbl)
  REAL(c_flt),  PARAMETER ::  FLT_EPSILON = EPSILON(0.0_c_flt)
  REAL(c_dbl),  PARAMETER ::  DBL_EPSILON = EPSILON(0.0_c_dbl)
  REAL(c_ldbl), PARAMETER :: LDBL_EPSILON = EPSILON(0.0_c_ldbl)
  REAL(c_flt),  PARAMETER ::  FLT_MIN     = TINY   (0.0_c_flt)
  REAL(c_dbl),  PARAMETER ::  DBL_MIN     = TINY   (0.0_c_dbl)
  REAL(c_ldbl), PARAMETER :: LDBL_MIN     = TINY   (0.0_c_ldbl)
END MODULE iso_c_float_h
```

Note 3.13

C's and Fortran's floating point number models are identical.

If a *c-kind-param* defined in ISO_C 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 and an arbitrary value) or not.

Except for the `unsigned` integer types, the values made accessible by a Fortran processor shall conform to the requirements of the C standard if that C type is

supported by the Fortran processor. For the `unsigned` integer types, the Fortran *type*`_MAX` constants shall have a value for which each binary digit w_k in the bit model of section 13.5.7 has the same value as the corresponding *bit* in the corresponding C unsigned integer *type*`_MAX` value.

3.3.3 C enumerated types

Fortran does not directly support enumerated types, and this Technical Report does not provide features to map C `enum` types to Fortran.

Although all C "enumerators" (the *enumeration constant*s) are of type `int`, the type chosen for a given enumeration type is implementation-defined: It need not be `int` but only conformable to any integer type which is capable of representing the values of this enumeration type's enumerators.

Note 3.14

The user is responsible for selecting the suitable implementation-defined integer kind for a given enumerated type. Obviously, type checking among different enumerated types is not possible when they are mapped to an integer type.

3.3.4 C structure types

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 3.15

A *pragma-string* may be used to provide additional, implementation dependent information to the Fortran processor. For example, `#pragma options` settings describing alignment of C structures may be given. However, the interpretation of *pragma-string*s is completely processor dependent.

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

Note 3.16

For example, the C structure type declaration

```
struct point {
 int x;
 int y;
};
```

can be mapped to Fortran in the following way:

```
TYPE point
 BIND(C)
 INTEGER(c_int) ::  x, y
END TYPE point
```

Note that the Fortran *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. Similarly, the Fortran member objects need not have the same names as the C structure members.
Nested structures can be treated in the same way: The C structure type declaration

```
struct rect {
 struct point pt1;
 struct point pt2;
};
```

can be mapped to a Fortran type

```
TYPE rect
 BIND(C)
 TYPE(point) ::  pt1, pt2
END TYPE rect
```

using the above mapping for `struct point` for the member objects.

The POINTER *component-attr-spec* is not allowed because there is no C type whose corresponding Fortran type has the POINTER attribute. C `struct`s that include *bit-field*s cannot be mapped to Fortran because this Technical Report does not specify mappings for bit-fields.

Note 3.17

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 of such derived types may differ from the layout of the original C `struct`.

A structure type with member objects of pointer type other than “pointer to

`void`" or "pointer to `char`" can be mapped to Fortran only if that pointer type has the same representation as "pointer to `void`", this is processor dependent. In such cases, the type of the corresponding member object shall be specified as TYPE(C_VOID_PTR) and the facilities of section 3.3.7 may be used to access that member object.

Note 3.18

For example, the self-referential C structure

```
struct tnode {
 char *word;
 int count;
 struct tnode *left;
 struct tnode *right;
};
```

represents a binary tree with two data fields and two pointers to other nodes of the tree. It can be mapped to the Fortran data type

```
TYPE tnode
 BIND(C)
 TYPE(C_CHAR_PTR) ::  word
 INTEGER(C_INT) ::  count
 TYPE(C_VOID_PTR) ::  left, right
END TYPE tnode
```

only if pointers to `struct tnode` have the same representation (size, alignment requirements) as pointers to `void`.

3.3.5 C union types

Fortran does not directly support union types, and this Technical Report does not provide features to map C `union` types to Fortran.

Note 3.19

C objects of `union` type may be accessed by specifying separate BIND(C) derived types for each union member (as if that member were the only member of a `struct`), and using TRANSFER or EQUIVALENCE to convert between these types. Note that the derived type used in the actual binding to a C `union` must be big enough to hold the "widest" member of the union and at the same time fulfil the most restrictive alignment requirement of all union members: In

```
union u_tag {
 char name[13];
 double val;
} u;
```

the member `name` probably is the widest member but the member `val` probably has more restrictive alignment requirements. This means that even if

```
TYPE u_name
 BIND(C)
 CHARACTER(LEN=1,KIND=c_char) ::  name(13)
END TYPE u_name
TYPE u_val
 BIND(C)
 REAL(c_dbl) ::  val
END TYPE u_val
```

are suitable definitions for the union members, both are probably insufficient to bind to the `union` object and it may be necessary to employ an additional derived type like

```
TYPE u_fits_all
 BIND(C)
 REAL(c_dbl) ::  alignment
 CHARACTER(LEN=1,KIND=c_char) ::  fill_up(5)
END TYPE u_fits_all
```

to fulfil both the size and alignment requirements.

3.3.6 C array types

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 C function parameter of unknown size, the *array-spec* shall be an *assumed-size-spec*. Otherwise, it shall be an *explicit-shape-spec-list*.

Note 3.20

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

```
extern 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. The Fortran declaration for that array should specifiy the DIMENSION(3) attribute. Note that the C declaration

```
extern char s[] = "I am a string";
```

results in `s` having the length 14 since a `'\0'` character gets automatically added. Fortran should specify DIMENSION(14) when binding to this C string.

Note 3.21

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* and sequence association.

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 3.22

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.

Note that arrays and pointers are closely related in C, and in most situations the C type "array of T" is implicitly converted to the C type "pointer to T". Notably, a C function parameter declared as "`double a[]`" is equivalent to a C function parameter declared as "`double *a`". The Fortran binding to such a C function parameter may build on any of these two declarations, regardless of which is actually used in the C source.

3.3.7 C pointer types

In C, the *pointer type* "pointer to T" derived from a *referenced type* T is different from the referenced type T, and is also different from pointer types derived from other types. Like all C derived type constructions, this pointer type derivation may be applied recursively.

This Technical Report does not provide features to map general C pointers to Fortran because Fortran cannot generally express C's pointer types. However, two special cases are supported: The C type "pointer to `void`" shall be mapped to a Fortran derived type with *type-name* C_VOID_PTR, and the C type "pointer to `char`" shall be mapped to a Fortran derived type with *type-name* C_CHAR_PTR. These two types shall have the BIND(C) attribute and PRIVATE components. A named constant C_NULL of type TYPE(C_VOID_PTR) shall be provided. This constant denotes a C *null pointer*, its value shall be distinct from all values denoting C pointers that compare unequal to the C macro `NULL`.

The following extensions to intrinsic operations shall be provided:

- ASSIGNMENT(=)
  C allows assignment of pointers without an explicit type cast if they have the same pointer type or if one is a pointer to `void`. Therefore, the ISO_C module shall provide ASSIGNMENT(=) for the following combinations of *variable* and *expr*:

Type of *variable*	Type of *expr*
TYPE(C_VOID_PTR)	TYPE(C_VOID_PTR)
TYPE(C_VOID_PTR)	TYPE(C_CHAR_PTR)
TYPE(C_CHAR_PTR)	TYPE(C_VOID_PTR)
TYPE(C_CHAR_PTR)	TYPE(C_CHAR_PTR)

  Execution of these assignments shall result in the definition of *variable* with the value of *expr*, as if the corresponding C assignment had taken place.

- Comparison by OPERATOR(==) and OPERATOR(/=)
  All C pointers may be compared to zero, denoted by the C macro `NULL`. To support this comparison, OPERATOR(==) and OPERATOR(/=) for the following operand types shall be provided:

Type of x_1	Type of x_2
TYPE(C_VOID_PTR)	TYPE(C_CHAR_PTR)
TYPE(C_CHAR_PTR)	TYPE(C_VOID_PTR)
TYPE(C_VOID_PTR)	TYPE(C_VOID_PTR)

  At least one of the actual arguments of type TYPE(C_VOID_PTR) shall have the value C_NULL, otherwise the result is undefined. OPERATOR(==) has result .TRUE. if the other operand represents a C null pointer and .FALSE. if it does not represent a C null pointer. OPERATOR(/=) has result .FALSE. if the other operand represents a C null pointer and .TRUE. if it does not represent a C null pointer.

Note 3.23

Comparing two pointers to members of the same array is not supported, as well as subtracting such pointers. Comparing a pointer with an integer constant zero is also not supported, the constant C_NULL must be used for this purpose.

The following two functions provide functionality similar to the C address operator `&` and C indirection (dereference) operator `*` :

C_ADDRESS (OBJ)

Description. Return the C-style address of a variable, converted to the type "pointer to `void`".

Class. Transformational function.

Arguments.

OBJ may be of any Fortran type which matches a C type, as specified in this Technical Report. It is an INTENT(IN) argument. The actual argument shall be a variable. It shall not be an array section, an assumed shape dummy argument, an allocatable array which is not allocated, or have the POINTER attribute.

Result Characteristics. The result is of type TYPE(C_VOID_PTR).

Result value. If OBJ is a scalar, the result has the value of the C expression `(void *) &`*`OBJ`*, where *OBJ* is the object associated with dummy argument OBJ. If OBJ is an array, the result value is `(void *) &`*`OBJ`*`[0]`. The result value is undefined if the type of OBJ does not match a C datatype, or if OBJ is a scalar of type TYPE(C_VOID_PTR) or TYPE(C_CHAR_PTR) and compares equal to C_NULL.

Example. If a C translation unit contains the external declaration

```
extern int cdata;
```

and the Fortran program accesses this variable by a module variable declared

```
INTEGER(c_int), BIND(C,"cdata") ::  fdata
```

the result value of C_ADDRESS(FDATA) is the C address of `cdata` cast to "pointer to `void`". This is the value of the C expression `(void *) &cdata`. If FSTRING is a rank-1 variable of type CHARACTER(KIND=C_CHAR), C_ADDRESS(FSTRING) returns a C pointer to the first character in that array, cast to "pointer to `void`". This result value may be assigned to an object of type TYPE(C_CHAR_PTR), using the extension of ASSIGNMENT(=) described above.

Note 3.24

Note that although OBJ is an INTENT(IN) argument and thus is not modified by a reference of C_ADDRESS, the value of the actual argument associated with OBJ may be changed later by C-style operations that modify the object "pointed to" by the result value of C_ADDRESS. The Fortran processor is not required to guard such behaviour.

Note 3.25

Note that if OBJ is an automatic object, the value returned by C_ADDRESS is probably meaningless when OBJ has gone out of scope. It is the responsibility of the programmer to take care of such effects.

C_DEREFERENCE (PTR [, MOLD])

Description. Dereference a C-style pointer.

Class. Transformational function.

Arguments.

PTR shall be a scalar of type TYPE(C_CHAR_PTR) or TYPE(C_VOID_PTR). It is an INTENT(IN) argument.

Optional Arguments.

MOLD shall be a scalar. It is an INTENT(IN) argument. It shall not be present if PTR is of type TYPE(C_CHAR_PTR). If PTR is of type TYPE(C_VOID_PTR), MOLD shall be present and shall have a type which matches a C type *T*, as specified in this Technical Report.

Result Characteristics. If PTR is of type TYPE(C_CHAR_PTR), the result is a rank-1 array of type CHARACTER(KIND=C_CHAR). If PTR is of type TYPE(C_VOID_PTR), the result is a scalar of the same type and type parameters as MOLD.

Result value.

Case (i): If PTR is of type TYPE(C_CHAR_PTR), the result value is the sequence of characters pointed to by PTR, up to and including the first ASCII `NUL` character.

Case (ii): If PTR is of type TYPE(C_VOID_PTR), the result has the value of the C expression `*((`*`MOLD *)PTR`*`)` where *PTR* is the object associated with dummy argument PTR. The result value is undefined if the type of MOLD does not match a C datatype.

If PTR compares equal to C_NULL, the result value is undefined.

Example.

Case (i): If a C translation unit contains the external definition

```
char *cmessage = "hello";
```

and the Fortran program accesses this pointer to a C string constant by a module variable declared as

```
TYPE(C_CHAR_PTR), BIND(C,"cmessage") ::  PTR
```

then C_DEREFERENCE(PTR) is a rank-1 array of size 5 and has the value `C_CHAR_"hello"` concatenated with an ASCII `NUL` character.

Case (ii): If a C translation unit contains the external definitions

```
double x = 1.2; void *px = &x;
```

and the Fortran program accesses the pointer by a module variable declared as

```
TYPE(C_VOID_PTR), BIND(C,"px") ::  PTR
```

the value of C_DEREFERENCE(PTR,MOLD=0.0_C_DBL) is 1.2_C_DBL.

Note 3.26

The argument MOLD is necessary for the case of generic (`void`) pointers to ensure that the `void` pointer is first converted to a pointer to the type of MOLD before it is dereferenced. Objects of type TYPE(C_VOID_PTR) and TYPE(C_CHAR_PTR) are allowed as MOLD arguments.

C pointer arithmetic is supported by the function C_INCREMENT, which supports both incrementing (positive N) and decrementing (negative N) a C pointer:

Note 3.27

This Technical Report does not support pointer arithmetic by means of adding or subtracting an integer to or from an object of type TYPE(C_VOID_PTR) using OPERATOR(+) or OPERATOR(−):
All pointer types (except pointer to `char`) are explicitly cast to "pointer to `void`" by the facilities specified in this Technical Report. However, pointer arithmetic needs the size of the original referenced type, which cannot be provided when using the operator form. The argument MOLD of the C_INCREMENT function can be used to provide this information about the referenced type.

C_INCREMENT (PTR [, MOLD] [, N])

Description. Increment a C-style pointer.

Class. Transformational function.

Arguments.

PTR shall be of type TYPE(C_VOID_PTR). It is an INTENT(IN) argument.

Optional Arguments.

MOLD shall be a scalar. It is an INTENT(IN) argument. It shall have a type which matches a C type *T*, as specified in this Technical Report.

N shall be scalar and of type integer. It is an INTENT(IN) argument.

Result Characteristics. The result type is TYPE(C_VOID_PTR).

Result value. The result has the value of the C expression

```
(void *)((MOLD *)PTR + N)
```

where *PTR* is the object associated with dummy argument PTR. The cast to "pointer to MOLD" only happens if MOLD is present, and if N is not present a default of one is used. The result value is undefined if the type of MOLD does not match a C datatype, or if PTR compares equal to C_NULL.

Example. If PTR is a pointer to the first element of an array of 10 elements of type `long double` which has been converted to "pointer to `void`", the result of the function reference C_INCREMENT(PTR,MOLD=0.0_C_LDBL,N=2) is a pointer to the third element of that array, converted to the type "pointer to `void`".

Note 3.28

Pointer arithmetic is needed to be able to deal with C's pointers to null-terminated arrays whose size is not known to the application. For example, POSIX.1 specifies that the values of environment variables are accessible through an external variable declared

```
extern char **environ;
```

This is a pointer to a null-terminated array of character strings. This type is not directly supported, but if the representation of "pointer to pointer to `char`" and "pointer to `void`" is identical, Fortran may access this external C variable by a module variable declared

```
TYPE(C_VOID_PTR), BIND(C,"environ") ::  f_environ
```

F_ENVIRON can be incremented by C_INCREMENT using an argument MOLD of type TYPE(C_CHAR_PTR), for N=1 the resulting value of type TYPE(C_VOID_PTR) points to the second "pointer to `char`" in the list. Dereferencing this value by C_DEREFERENCE using an argument MOLD of type TYPE(C_CHAR_PTR) results in a value of type TYPE(C_CHAR_PTR). This points to the second character string, or compares equal to C_NULL if there is no second character string. Dereferencing a second time (without MOLD) returns the actual string.

In addition to the C pointer types "pointer to `void`" and "pointer to `char`" described above, section 3.4.1 describes how explicit procedure interfaces that have the BIND(C) attribute can be used to specify function prototypes of the form "pointer to function returning T" (or equivalently "function returning T"), and how the PASS_BY attribute can be used to specify additional C pointer declarators for dummy arguments. Other pointer types are not directly supported by this Technical Report.

C pointer types which have the same representation as "pointer to `void`" may be mapped to type TYPE(C_VOID_PTR), this applies to function result variables and dummy arguments as well as to `extern` data objects and `struct` member objects. However, this is processor dependent and the behavior for cases where the C pointer type has a representation different from "pointer to `void`" is undefined.

3.3.8 C function types

C function types whose declarator does not contain a *parameter-type-list* are not supported by this Technical Report. The specification of a C function prototype by means of an explicit interface with the BIND(C) attribute is described in section 3.4.1.

3.3.9 Handling of C `typedef` names

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

R1606 *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-alias-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 3.29

For derived type *type-name*s, 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.

If the aliased *type-spec* is a derived type, the *expr-list* in a *structure-constructor* for *type-alias-name* shall be a valid *expr-list* for a *structure-constructor* for that derived type. 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.

Note 3.30

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) *type-spec* in the application program, these details may be hidded by declaring type aliases

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

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

3.3.10 Type qualifiers

This Technical Report does not provide facilities to specify C's *type-qualifier*s `const` or `volatile`. When mapping to a qualified type, the Fortran program shall use the corresponding unqualified type. This is possible since qualified types have the same representation as the corresponding unqualified types.

If objects of a const-qualified type are used from within the Fortran program in ways that violate the rules of the C standard for such objects, their value becomes undefined. The Fortran processor is not required to detect such violations. The value of a volatile-qualified C data object is unspecified within the Fortran program.

3.3.11 Storage class specifiers

With the exception of `typedef` which is described in section 3.3.9, this Technical Report does not provide facilities to specify C's *storage-class-specifier*s.

Note 3.31

Since only C objects and functions with *external linkage* are addressed by this Technical Report, it is assumed that all C entities accessed from Fortran have the `extern` *storage-class-specifier* (or behave as if they had). Entities which have the `static` *storage-class-specifier* are either local variables, or have file scope (which is comparable to Fortran PRIVATE entities). They are all inaccessible from Fortran, as well as local variables declared `auto` or `register`.

If a C function's parameter has the `register` *storage-class-specifier* , the behavior when referencing that function from Fortran is undefined.

3.4 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* C 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 *function-stmt* or *subroutine-stmt* in the corresponding *interface-body* shall contain a *bind-spec* specification with *lang-keyword* C.

Section 3.4.1 contains the rules for the specification of a Fortran explicit interface corresponding to a C function prototype. Section 3.4.2 describes the procedure reference to such a procedure, including the modified process of argument association. Support for C varying length argument lists is provided in section 3.4.3.

3.4.1 Procedure interface for BIND(C) binding

This Technical Report does not support old-style C function declarators which do not contain a *parameter-type-list*. An explicit interface with the BIND(C) attribute corresponds to a C function prototype inluding a *parameter-type-list*.

Note 3.32

This restriction is imposed to avoid the complicated rules of C for mixed old-style and new-style function declarations and definitions.
When binding to a C function whose definition is old-style (an optional *identifier-list* is enclosed in the function's parantheses), an explicit interface should be specified which corresponds to the C function prototype which would result from C's *default argument promotion* of the old-style C function definition.

3.4.1.1 Prefix specifications

The prefix of a *function-stmt* or *subroutine-stmt* in an interface-body that corresponds to a C function prototype shall contain a *bind-spec* specification with *lang-keyword* C. Specifying RECURSIVE in the interface body is allowed and has no effect. Since a pure procedure must be a subprogram (that is, defined by means of Fortran), the PURE or ELEMENTAL prefix shall not be present in the interface body.

If the Fortran entity is a dummy procedure, no *name-string* shall be present. If the Fortran entity is an external procedure and no *name-string* is present, 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 the Fortran entity is an external procedure and 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 external name.

3.4.1.2 Mapping the C function's return type

If the C function's return type is `void`, the Fortran interface shall specify a subroutine. Otherwise, the Fortran interface shall specify a function with a scalar result type.

Note 3.33

The ISO C standard requires that the return type of a C function shall not be a function type or array type (6.5.4.3). This implies that a Fortran interface for a C function must not specify an array-valued function result.

The declaration of the function result variable shall be as follows: If the return type of the C function is

- a basic type or a structure type, the function result variable shall have the type specified for that C type in section 3.3 of this Technical Report.

- an enumeration type or union type, this is not directly supported.
- "pointer to `char`" or "pointer to `void`", the function result variable shall have the type TYPE(C_CHAR_PTR) or TYPE(C_VOID_PTR), respectively.

All other C pointer types are not directly supported.

Note 3.34

A pointer type which has the same representation as "pointer to `void`" may be handeled by specifying the type TYPE(C_VOID_PTR) for the function result variable. This is processor dependent.

3.4.1.3 The PASS_BY attribute

The **PASS_BY attribute** specifies C *pointer* declarators for dummy data objects. It can be declared by a *pass-by-spec* inside an *interface-body* whose *function-stmt* or *subroutine-stmt* has the BIND(C) prefix.

R1607 *pass-by-spec* **is** PASS_BY (*pass-by-string* □
□ [, PRAGMA=*pragma-string*])

R1608 *pass-by-string* **is** *scalar-default-char-init-expr*

Constraint: The *pass-by-string* shall have length zero, or the value "*".

The term "PASS_BY(*pass-by-string*) attribute" denotes the PASS_BY attribute with the given *pass-by-string*, this term does not imply presence or absence of a *pragma-string*.

A "*" character in the *pass-by-string* is interpreted as a C *pointer* declarator, and changes the type T of the dummy argument as specified without that pointer declarator to the type "pointer to T". The interpretation of a *pragma-string* is processor dependent.

Note 3.35

A *pragma-string* may be used to provide additional, implementation dependent information to the Fortran processor. For example, information about near or far addressing may be given (on architectures with a segmented memory model). However, the interpretation of *pragma-string*s is completely processor dependent.

Since Fortran also provides specification statements for attributes, the *pass-by-spec* may alternatively be specified by a **PASS_BY statement**.

R1609 *pass-by-stmt* **is** *pass-by-spec* [::] *dummy-arg-name-list*

Constraint: A *pass-by-stmt* shall only occur inside an *interface-body* whose *function-stmt* or *subroutine-stmt* has the BIND(C) prefix.

Constraint: Each *dummy-arg* shall be a dummy data object.

The *pass-by-spec* is used in the following section to map C function parameters to Fortran dummy arguments.

3.4.1.4 Mapping C function parameters to dummy arguments

The *interface-body* that specifies a Fortran interface to a C function shall specify dummy arguments that correspond by position with the C function parameters in the *parameter-type-list*. If the list consists solely of `void`, no dummy argument shall be specified. Section 3.4.3 deals with the case that the list terminates with an *ellipsis*.

The Fortran declaration for these dummy arguments shall be as follows: If the type of the C function parameter is

- a basic type or a structure type, the dummy argument shall be scalar and have the type specified for that C type in section 3.3 of this Technical Report. No *pass-by-spec* with a *pointer* declarator shall be present.
- an enumeration type or union type, this is not directly supported.
- a type "function returning T", this type is adapted by the C processor to the type "pointer to function returning T" and the rules for that type shall be followed.
- a type "array of T", this type is adapted by the C processor to the type "pointer to T". The Fortran interface shall either use a declaration corresponding to "pointer to T", or shall declare the type corresponding to the C type T, a DIMENSION attribute corresponding to the C array declarator, and no *pass-by-spec* with a *pointer* declarator.

Note 3.36

Note that all information about the shape of the argument is lost when the adapted type "pointer to T" is used in the mapping.

If the type of the C function parameter is "pointer to T", the rules for the dummy argument depend on the referenced type: If the type T is

- the incomplete type `void`, the dummy argument shall be a scalar of type TYPE(C_VOID_PTR) and no *pass-by-spec* with a *pointer* declarator shall be present.
- `char`, the dummy argument shall be a scalar of type TYPE(C_CHAR_PTR) and no *pass-by-spec* with a *pointer* declarator shall be present. Alternatively, the dummy argument may be a scalar of type CHARACTER(KIND=C_CHAR) for which the PASS_BY("*") attribute is specified.
- an integer type, floating type, or structure type, the dummy argument shall be a scalar which has the type corresponding to the C type T, and the PASS_BY("*") attribute shall be present.
- an enumeration type or union type, this is not directly supported.

- a type "function returning *T1*", a dummy procedure shall be declared. There shall be an explicit interface for the dummy procedure in the *specification-part* of the *interface-body*, and that interface shall specify the BIND(C) attribute without a *name-string*. The explicit interface of the dummy procedure shall correspond to the function prototype of the C funtion parameter, as specfifed in this section.

- "pointer to `char`" or "pointer to `void`", the dummy argument shall be a scalar of type TYPE(C_CHAR_PTR) or TYPE(C_VOID_PTR), respectively, and the PASS_BY("*") attribute shall be present.

All other C pointer types are not directly supported.

Note 3.37

A pointer type which has the same representation as "pointer to `void`" may be handeled by specifying a dummy data object of type TYPE(C_VOID_PTR). This is processor dependent.

3.4.1.5 Further restrictions on BIND(C) interfaces

Regardless of the form of the C function prototype to which an explicit interface with the BIND(C) attribute might correspond, the following restrictions apply within an interface body which has the BIND(C) attribute:

- OPTIONAL, POINTER or TARGET shall not be specified.

- INTENT other than IN shall not be specified for scalar dummy arguments which do not have a *pass-by-spec* with a *pointer* declarator.

- A dummy argument or function result variable shall not be an assumed shape array.

- A dummy argument or function result variable shall not have the type COMPLEX, LOGICAL, or CHARACTER with a character length greater than one.

- If a dummy argument or function result variable has derived type, that type shall have the BIND(C) attribute.

- All dummy procedures shall have an explicit interface, and that interface shall specify the BIND(C) attribute.

Note 3.38

The BIND(FORTRAN) attribute is redundant, and places no restrictions (other than those specified in IS 1539) on the contents of an interface body having that attribute.

3.4.2 Procedure reference for BIND(C) binding

The C standard specifies that in preparing for the call to a C function, the arguments are evaluated, and each parameter is assigned the value of the corresponding argument (6.3.2.2). A C function may change the values of its parameters, but these changes cannot affect the value of the arguments. On the other hand, it is possible to pass the pointer to an onject, and the function may change the value of the object pointed to (6.3.2.2, footnote 39).

The BIND(C) attribute in an *interface-body* instructs the Fortran processor to adapt its rules concerning actual arguments, dummy arguments, and argument association accordingly when referencing such a procedure. This section describes these modified semantics.

3.4.2.1 Actual arguments associated with dummy data objects

An actual argument to a procedure which has the BIND(C) attribute shall not have the POINTER attribute.

If the dummy argument is an array, the actual argument shall be an array of the same type and type parameters. It shall be a variable, and shall not be an array section, an assumed shape dummy argument, or an allocatable array which is not allocated. The C function parameter is assigned the function result value of C_ADDRESS applied to the actual argument.

Note 3.39

> Note that because a C pointer is passed, the C function may modify the actual argument.

If the dummy argument is a scalar of type TYPE(C_VA_LIST), the behavior is specified in section 3.4.3.

If the dummy argument is a scalar of intrinsic type or of a derived type which has the BIND(C) attribute, and does not have a *pass-by-spec* with a *pointer* declarator, the actual argument shall be a scalar expression or a scalar data object. It shall have a type and type parameters for which assignment to a variable of the type and type parameters of the dummy argument is defined by IS 1539-1 or this Technical Report. The C function parameter is assigned the value of the actual argument, converted as if by such assignment to the type and type parameters of the dummy argument. ASSIGNMENT(=) for the types TYPE(C_VOID_PTR) and TYPE(C_CHAR_PTR) is accessible for this conversion, even if the caller does not contain a module reference to the intrinsic module ISO_C.

Note 3.40

> The conversion (as if by assignment) of the actual argument to the type of the dummy argument corresponds to the conversion which occurs within C when referencing a C function with a prototype declaration visible.

Note 3.41

Note that because a copy of the (possibly converted) value of the actual argument is passed, the C function cannot modify the actual argument.

If the dummy argument is a scalar of intrinsic type or of a derived type which has the BIND(C) attribute, and the PASS_BY("*") attribute is present, the actual argument shall be scalar. It shall be a variable whose type and type parameters agree with those of the dummy argument, or it shall be an expression of type TYPE(C_VOID_PTR) which compares equal to C_NULL. If it is of type TYPE(C_VOID_PTR) or TYPE(C_CHAR_PTR) and its value compares equal to C_NULL, the C function parameter is assigned the value of the C macro `NULL`. Otherwise, the C function parameter is assigned the function result value of C_ADDRESS applied to the actual argument.

Note 3.42

Note that because a C pointer is passed, the C function may modify the actual argument. Also note that if the actual argument denotes a C null pointer and the C function dereferences the corresponding C function parameter, the behavior is undefined.

3.4.2.2 Actual arguments associated with dummy procedures

If a dummy argument is a dummy procedure, it shall have an explicit interface, and the corresponding *function-stmt* or *subroutine-stmt* shall have the BIND(C) prefix. The associated actual argument shall be the specific name of an external or dummy procedure, that procedure shall have an explicit interface, and the corresponding *function-stmt* or *subroutine-stmt* shall have the BIND(C) prefix.

The characterisitics of the associated actual procedure shall agree with those of the dummy procedure.

3.4.2.3 Function results of C pointer types

Note 3.43

Note that if the result variable has type TYPE(C_CHAR_PTR) or TYPE(C_VOID_PTR), the C function may return a pointer to storage which was allocated in the C function (by `calloc`, `malloc` or `realloc`). It may be necessary to `free` that storage later on.
If the value of the function result is used in an expression, its value may be inaccessible after the evaluation of that expression. This means that it will be impossible to later call `free` with that pointer, which will probably cause memory leaks. It is the responsibility of the programmer to take care about such situations.

3.4.3 Support for C variable argument lists `<stdarg.h>`

C functions may be called with a variable number of arguments of varying type. The argument list in the function prototype of such a function must contain one or more parameters followed by an *ellipsis* (. . .). The called function may access the varying number of actual arguments by facilities defined in the standard header `<stdarg.h>`. Fortran does not directly support this kind of procedure interfaces.

Note 3.44

The Fortran concept of OPTIONAL arguments is less flexible. It requires the specification of all possible combinations of arguments at compile time, including their types.

To provide a Fortran interface to an external C function which contains an *ellipsis*, an intrinsic module ISO_C_STDARG_H shall be provided. This module shall provide access to the following:

- A derived type definition with *type-name* C_VA_LIST. This type has the BIND(C) attribute and PRIVATE components.

Note 3.45

The type TYPE(C_VA_LIST) defined in the Fortran binding is used by the Fortran processor to build up the variable length argument list and pass it to the C function. It is not necessarily a translation of the C type `va_list` defined in `<stdarg.h>`. This latter type and the associated macros are only used in the definition of the callee, and are not required on the Fortran side.

- A named constant C_VA_EMPTY of type TYPE(C_VA_LIST). The value of this constant shall be distinct from any value that may result from an argument list construction by means of OPERATOR(//).

- ASSIGNMENT(=) for *variable*s and *expression*s of type TYPE(C_VA_LIST). Execution of this assignment causes the definition of *variable* with a copy of the value of *expression*.

- An extension of OPERATOR(//) to operands x_1 of type TYPE(C_VA_LIST). x_2 may be of any scalar type corresponding to a C data type other than `char`, as specified in section 3.3 of this Technical Report, or it may be of type TYPE(C_VA_LIST). Both x_1 and x_2 shall be scalar. The result type is TYPE(C_VA_LIST). The result value is a copy of x_1 concatenated with the value resulting from C's *default argument promotion* of x_2. The following table shows the type conversions that take place for x_2, for all other types the value of x_2 is used without any conversion.

Type of x_2	Value resulting from promotion
INTEGER(C_SCHAR)	INT(x_2, KIND=C_INT)
INTEGER(C_SHRT)	INT(x_2, KIND=C_INT)
INTEGER(C_UCHAR)	INT(x_2, KIND=C_INT)
INTEGER(C_USHRT)	INT(x_2, KIND=C_INT)
REAL(C_FLT)	REAL(x_2, KIND=C_DBL)

Paranteses used to specify the order of evaluation have no effect on the value of the result.

Note 3.46

If an array A is to be passed as a varying argument, the only way to achieve this is to pass the result value of C_ADDRESS(A), which is a scalar of type TYPE(C_VOID_PTR).

Note 3.47

Default argument promotion takes place when constructing the varying argument list rather than at the time of argument association for a procedure reference. This is motivated by the fact that an explicit interface with the BIND(C) attribute represents a C function prototype (new-style function declarator). In this case no default argument promotion takes place. Varying arguments always suffer default argument promotion, so to avoid complicated argument association rules this promotion is done when constructing the varying argument list.

Note 3.48

Integral promotion of the C type `char` is not supported because this Technical Report does not match that type with a Fortran integer type. If required, one of its signed or unsigned variants may be used. Integral promotion of C bit-fields or enumeration types is not required because these types are not supported in Fortran.

The *interface-body* that specifies a Fortran interface to a C function containing an *ellipsis* shall specify the dummy arguments as described in 3.4.1, and shall specify an additional scalar dummy argument of type TYPE(C_VA_LIST) in the position of the *ellipsis*. INTENT other than IN shall not be specified for this argument. A *pass-by-spec* with a *pointer* declarator shall not be specified for this argument.

The actual argument associated with this dummy argument shall be a scalar expression of type TYPE(C_VA_LIST). To indicate an empty varying argument list, its value shall be C_VA_EMPTY. To pass a non-empty list of varying arguments, its value shall be C_VA_EMPTY concatenated by the OPERATOR(//) with the list of required arguments. C_VA_EMPTY shall be the leftmost operand of this concatenation, all other operands shall be specified from left to right corresponding to their respective position in a C function reference. The Fortran processor

translates the values stored in the TYPE(C_VA_LIST) argument into the representation expected by the C processor for a C function reference.

Note 3.49

For example, the POSIX.1 function *fcntl()* has the prototype

```
int fcntl (int fildes, int cmd, ...);
```

If the parameter *cmd* has the value of the named constant F_DUPFD, *fcntl()* expects a third argument *arg* of type `int`, and returns a new file descriptor that is the lowest numbered available file descriptor greater than or equal to *arg*. If *cmd* has the value of the named constant F_GETFD, *fcntl()* expects no third parameter and returns the file descriptor flags associated with the file descriptor *fildes*. A Fortran interface for this function prototype may be

```
BIND(C,NAME="fcntl") FUNCTION fcntl ( fildes, cmd, va )
 USE iso_c, ONLY: c_int
 USE iso_c_stdarg_h, ONLY: c_va_list
 INTEGER(c_int), INTENT(IN) ::  fildes, cmd
 TYPE(c_va_list), INTENT(IN) ::  va
 INTEGER(c_int) ::  fcntl
END FUNCTION fcntl
```

With this explicit interface accessible, the function references

```
I = FCNTL ( FD, F_DUPFD, C_VA_EMPTY // 10_C_INT )
J = FCNTL ( FD, F_GETFD, C_VA_EMPTY )
```

return the lowest numbered available file descriptor greater than or equal to 10 in `I` and the file descriptor flags associated with `FD` in `J`.

3.5 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 Fortran type corresponding to T (as specified in section 3.3 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, subject to the rules of use association.

To indicate that the storage for the Fortran variable is reserved by the C translation unit containing the definition of that C external variable, the BIND(C) attribute shall be specified with a *name-string* whose value is the identifier of the C extern

data object. Because that data object is a global data object, the BIND(C) attribute for a module variable implies the SAVE attribute for that varible.

Note 3.50

For example, POSIX.1 requires that the standard header `<errno.h>` contains a declaration

```
extern int errno;
```

This is only a declaration, not a definition: storage for this variable is reserved somewhere else, probably in the kernel. The value of `errno` is set by various functions of the C standard library and POSIX.1. A Fortran module like

```
MODULE my_errno
  USE iso_c, ONLY: c_int
  INTEGER(c_int), BIND(C,NAME="errno") ::  errno
END MODULE my_errno
```

might be USEd in scoping units that need to access `errno`.

The following additonal restrictions apply for module variables having the BIND(C) attribute:

- No *initialization* shall appear in the *entity-decl*.
- ALLOCATABLE, PARAMETER, POINTER or TARGET shall not be specified.
- If the object has derived type, that type shall have the BIND(C) attribute.
- The object shall not have type COMPLEX, LOGICAL, or CHARACTER with a character length greater than one.

If two or more module variables with the BIND(C) attribute and the same *name-string* are accessible in a scoping unit, the following rules apply:

Case (i): If they all have the same type, type parameters, and shape, they all refer to the same storage.

Case (ii): If at least one differs in type, type parameters, or shape, the value of all accessible module variables with that *name-string* is undefined in that scoping unit.

Note 3.51

Note that Fortran prohibits the appearance of a module variable as the *variable-name* in a *common-block-object* or as an *equivalence-object*, so module variables having the BIND(C) attribute cannot be equivalenced or be menbers of a common block. However, the C `extern` variable is a global variable, Binding different Fortran module variables to the same C object effectively EQUIVALENCEs them.

The value of a module variable having the BIND(C) attribute may be changed by the execution of C functions which have an external declaration of the identifier in the corresponding *name-string* in scope. The Fortran processor shall not assume that the value of such a module variable remains unchanged after a procedure reference to a procedure which has a BIND(C) prefix.

Note 3.52

For example, if a function reference to FCNTL in Note 3.49 returns a value of -1, the value of the module variable in Note 3.50 after that function reference will be set to the corresponding error number.

The value of a module variable having the BIND(C) attribute may also be changed by other means invisible to the Fortran program. The Fortran processor is not required to guard such behavior.

4 Editorial changes to ISO/IEC 1539-1 : 1997

This section contains the editorial changes to ISO/IEC 1539-1:1997 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 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 and 41:

		or	*bind-stmt*
		or	*pass-by-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).

Note 4.1

Synchronize stuff about intrinsic modules with the Technical Report for Floating-Point Exception Handling (currently the draft PDTR, ISO/IEC JTC1/SC22/WG5/N1231).

Page 38
Subclause 4.4.1
In

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

add after line 31:

		or	*bind-spec*

Note 4.2

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*.

Constraint: If a *bind-spec* is present, all derived types specified in component definitions shall have a *bind-spec* with the same *name-string*.

Constraint: If a *bind-spec* is present, SEQUENCE shall not be specified.

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*
or	*pass-by-spec*

Page 48
Subclause 5.1
In the Constraints list, add after line 18:

Constraint: A *bind-spec* may only be specified for a variable in the *specification-part* of a module.

Constraint: A *pass-by-spec* may only be specified for a dummy data object within an *interface-body* whose *function-stmt* or *subroutine-stmt* has the BIND(C) prefix.

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

If a *bind-spec* is present, the additional constraints of sections 16.4 and 16.5 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. For variables in the *specification-part* of a module, this attribute may also be declared via the BIND statement (16.2).

Note 4.3

Maybe add 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.

Page 192
Subclause 12.2
At line 6, add at the end of the paragraph:

> If the procedure has a *bind-spec* prefix, this is a characteristic.

Page 192
Subclause 12.2.1.1
At line 17, add at the end of the paragraph:

> If the dummy data object has a *pass-by-spec* attribute, this is a characteristic.

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 (16.4).

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.*

Constraint: A *bind-spec* for a dummy procedure shall not have a *name-string.*

Constraint: If a *bind-spec* with a *lang-keyword* other than FORTRAN is present, PURE or ELEMENTAL shall not be present.

Page 208
Subclause 12.5.2.2
Add after line 4:

> If a *bind-spec* with *lang-keyword* C is present in the *prefix* of the function, the additional constraints of section 16.4 apply.

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

> Constraint: If a *bind-spec* with *lang-keyword* C is present in the *prefix* of the subroutine, * shall not appear as *dummy-arg*.

Page 209
Subclause 12.5.2.3
Add after line 3:

> If a *bind-spec* with *lang-keyword* C is present in the *prefix* of the subroutine, the additional constraints of section 16.4 apply.

Page 211
Subclause 12.5.3
After "external subprogram" on line 14, add ", except when the binding mechanisms described in section 16.4 are used".

Page 292
New clause 16
Introduce a new section "Section 16: Interoperability with ISO C":
Take section 3 of this Technical Report, replace all occurences of "Technical Report" by "International Standard", renumber the sectioning 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.

Note 4.4

> Annex 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 :-)