WORKING DRAFT

N1808

26th February 2010 10:32

This is an internal working document of J3.

Contents

List of Tables

Foreword

1 ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and nongovernmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

2 International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

3 The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

4 Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

5 ISO/IEC TR 29113:2010(E) was prepared by Joint Technical Committee ISO/IEC/JTC1,*Information technology*, Subcommittee SC22, *Programming languages, their environments and system software interfaces.*

6 This technical report specifies an enhancement of the C interoperability facilities of the programming language Fortran. Fortran is specified by the International Standard ISO/IEC 1539-1:2010.

7 It is the intention of ISO/IEC JTC1/SC22/WG5 that the semantics and syntax specified by this technical report be included in the next revision of the Fortran International Standard without change unless experience in the implementation and use of this feature identifies errors that need to be corrected, or changes are needed to achieve proper integration, in which case every reasonable effort will be made to minimize the impact of such changes on existing implementations.

Introduction

Technical Report on Further Interoperability of Fortran with C

1 The system for interoperability between the C language, as standardized by ISO/IEC 9899:1999, and Fortran, as standardized by ISO/IEC 1539-1:2010, provides for interoperability of procedure interfaces with arguments that are non-optional scalars, explicit shape arrays, or assumed size arrays. These are the cases where the Fortran and C data concepts directly correspond. Interoperability is not provided for important cases where there is not a direct correspondence between C and Fortran.

2 The existing system for interoperability does not provide for interoperability of interfaces with Fortran dummy arguments that are assumed-shape arrays, or dummy arguments with the Fortran allocatable, pointer, or optional attributes. As a consequence, a significant class of Fortran subprograms are not portably accessible from C, limiting the usefulness of the facility.

3 The existing system also does not provide for interoperability with C prototypes that have formal parameters declared (void *). The class of such C functions includes widely used library functions that involve copying blocks of data, such as those in the MPI library.

4 ISO/IEC TR 29113 extends the facility of Fortran for interoperating with C to provide for interoperability of procedure interfaces that specify assumed shape dummy arguments, or dummy arguments with the allocatable, pointer, or optional attributes. New Fortran concepts of assumed-type and assumed-rank are provided to facilitate interoperability of procedure interfaces with C prototypes with formal parameters declared (void *). An intrinsic function, RANK, is specified to obtain the rank of an assumed-rank variable.

5 The facility specified in ISO/IEC TR 29113 is a compatible extension of Fortran as standardized by ISO/IEC 1539-1:2010. It does not require that any changes be made to the C language as standardized by ISO/IEC 9899:1999.

6 ISO/IEC TR 29113 is organized in 5 clauses:

7 It also contains the following nonnormative material:

Technical Report — Further Interoperability of Fortran with C —

1 Overview

1.1 Scope

1 ISO/IEC TR 29113 specifies the form and establishes the interpretation of facilities that extend the Fortran language defined by ISO/IEC 1539-1:2010. The purpose of ISO/IEC TR 29113 is to promote portability, reliability, maintainability and efficient excution of programs containing parts written in Fortran and parts written in C for use on a variety of computing systems.

1.2 Normative references

1 The following referenced standards are indispensable for the application of this document.

2 ISO/IEC 1539-1:2010, *Information technology—Programming languages—Fortran*

3 ISO/IEC 9899:1999, *Information technology—Programming languages—C*

1.3 Terms and definitions

1 For the purposes of this document, the following terms and definitions apply. Terms not defined in ISO/IEC TR 29113 are to be interpreted according to ISO/IEC 1539-1:2010.

1 **1.3.1**
assumed-rank object
dummy variable whose rank is assumed from its effective argument

1 **1.3.2**
assumed-type object
dummy variable whose type and type parameters are assumed from its effective argument

1 **1.3.3**
C descriptor
struct of type CFI_cdesc_t

1 **1.3.4**
Fortran descriptor
a structure used by the processor to describe an object that is assumed-shape, assumed-rank, allocatable, or a data pointer

1.4 Compatibility

1.4.1 New intrinsic procedures

1 ISO/IEC TR 29113 defines intrinsic procedures in addition to those specified in ISO/IEC 1539-1:2010. Therefore, a Fortran program conforming to ISO/IEC 1539-1:2010 might have a different interpretation under ISO/IEC TR 29113 if it invokes an external procedure having the same name as one of the new intrinsic procedures, unless that procedure is specified to have the EXTERNAL attribute.

1.4.2 Fortran 2008 compatibility

1 ISO/IEC TR 29113 is an upwardly compatible extension to ISO/IEC 1539-1:2010.

2 Data Attributes

2.1 Assumed-type objects

1 An assumed-type object is a dummy variable with no declared type and whose dynamic type and type parameters are assumed from its effective argument. An assumed-type object is declared with a *declaration-type-spec* of TYPE (*).

C201 An assumed-type entity shall be a dummy variable.

C202 An assumed-type variable shall not have the CODIMENSION or VALUE attribute.

2 An assumed-type variable may appear only as a dummy argument, an actual argument associated with a dummy argument that is assumed-type, or the first argument to the intrinsic and intrinsic module function ALLOCATED, ASSOCIATED, IS_CONTIGUOUS, LBOUND, PRESENT, RANK, SHAPE, SIZE, UBOUND, or C_LOC.

2.2 Assumed-rank objects

1 An assumed-rank object is a dummy variable whose rank is assumed from its effective argument. As assumed-rank object is declared with an *array-spec* that is an *assumed-rank-spec*.

R201 *assumed-rank-spec* **is** ..

C203 An assumed-rank entity shall be a dummy variable.

C204 An assumed-rank variable shall not have the CODIMENSION or VALUE attribute.

2 An assumed-rank variable may appear only as a dummy argument, an actual argument associated with a dummy argument that is assumed-rank, the argument of the C_LOC function in the ISO_C_BINDING intrinsic module, or the first argument in a reference to an intrinsic inquiry function. The RANK inquiry intrinsic may be used to inquire about the rank of an array.

3 The rank of an assumed-rank object may be zero.

2.3 OPTIONAL attribute

1 The OPTIONAL attribute may be specified for a dummy argument in a procedure interface that has the BIND attribute.

C205 A dummy argument that has the OPTIONAL attribute and is declared in an interface that is specified with a *proc-language-binding-spec* shall not have the VALUE attribute.

3 Procedures

3.1 Characteristics of dummy data objects

1 Whether the type or rank of a dummy data object is assumed is a characteristic of the dummy data object.

3.2 Explicit interface

1 A procedure shall have an explicit interface if it is referenced and the procedure has a dummy argument that is assumed-type or assumed-rank.

3.3 Argument association

1 An assumed-rank dummy argument may correspond to an actual argument of any rank. If the actual argument is scalar, the dummy argument has rank zero and the shape and bounds are arrays of zero size. If the actual argument is an array, the bounds of the dummy argument are assumed from the actual argument.

2 An assumed-type dummy argument is type and kind compatible with a nonpolymorphic actual data argument of any declared type.

NOTE 3.1

Because the type and type parameters of an assumed-type dummy argument are assumed from its effective argument, two such arguments are not distinguishable based on type for purposes of generic resolution. Similarly, the rank of arguments cannot be used for generic resolution if the dummy argument is assumed-rank.

4 Intrinsic procedure

4.1 Specification of the standard intrinsic procedure

4.1.1 General

1 Detailed specification of the RANK generic intrinsic procedure is provided in 4.1. The types and type parameters of the RANK intrinsic procedure argument and function result are determined by this specification. The "Argument" paragraph specifies requirements on the actual arguments of the procedure. The RANK intrinsic function is a pure function.

4.1.2 RANK (A)

1 **Description.** Rank of a data object.

2 **Class.** Inquiry function.

3 **Arguments.**

A	shall be a scalar or array of any type.

4 **Result Characteristics.** Default integer scalar.

5 **Result Value.** The result is the rank of A.

6 **Example.** For an array X declared REAL :: X(:,:,:), RANK(X) is 3.

5 Interoperability with C

5.1 Object descriptors

1 A **Fortran descriptor** is a structure used by the processor to describe an object that is assumed-shape, assumed-rank, allocatable, or a data pointer. A **C descriptor** is a struct of type CFI_cdesc_t. The C descriptor along with library functions with standard prototypes that allow for conversion between Fortran and C descriptors provide the means for describing an assumed-shape, assumed-rank, allocatable, or data pointer object within a C function.

NOTE 5.1

The Fortran processor may already define a structure for this purpose independent of a use in interface operability with C prototypes. If this structure contains sufficient information it may be used directly as a Fortran descriptor. The internal structure of a Fortran descriptor might not be the same for all implementations, so direct use of this descriptor by a C program is not portable.

NOTE 5.2

This two descriptor model for describing data objects has these characteristics:

(1) The internal structure of the Fortran descriptor is not exposed to the C programmer.
(2) The C descriptor contains enough information to efficiently access the described object within a C function.
(3) The C descriptor contains enough information to create a corresponding Fortran descriptor that is usable by the companion Fortran processor.
(4) Methods are provided for use by a C function that allow for allocation or deallocation of objects that is compatible with effects of the corresponding Fortran ALLOCATE and DEALLOCATE statements.
(5) A method is provided to compute stride multipliers based on lower bound, extent, and stride values.

5.1.1 Fortran descriptors

1 The Fortran descriptor for an object shall contain enough information that the following can be determined by the C library function that defines a C descriptor from a Fortran descriptor.

(1) The base C address of the object unless the object is an unallocated allocatable, or a data pointer that is not associated.
(2) The size, as would be returned by the C sizeof operator, of a scalar of the same type and type parameter values as the object.
(3) The Fortran rank of the object. If this value is zero, the object is scalar.
(4) Whether the object has the pointer attribute.
(5) Whether the object has the allocatable attribute.
(6) If the Fortran object has the pointer attribute, whether or not the pointer is disassociated.
(7) If the Fortran object has the allocatable attribute, whether or not the object is allocated.
(8) The type of the object if the object is of intrinsic interoperable type, or that the object is of derived type.
(9) Triples that contain the lower bound, extent, and stride multiplier for each dimension of the object unless the object is an unallocated allocatable, a pointer that is disassociated, or a scalar.

2 The Fortran descriptor may contain additional information that is not required in order to create a corresponding C descriptor.

5.2 C descriptors

1 The C descriptor is a struct of type CFI_cdesc_t. This struct is defined in the file ISO_Fortran_binding.h.

5.3 ISO_Fortran_binding.h

5.3.1 Summary of contents

1 The ISO_Fortran_binding.h file contains the definitions of the C structs CFI_cdesc_t, CFI_dim_t, and CFI_bounds_t, macro definitions that expand to integer constants with type int, and C prototypes for the C functions CFI_update_cdesc, CFI_update_fdesc, CFI_free_fdesc, CFI_allocate, CFI_deallocate, CFI_is_contiguous, CFI_bounds_to_cdesc, and CFI_cdesc_to_bounds. The contents of ISO_Fortran_binding.h can be used by a C function to interpret a Fortran descriptor, allocate and deallocate objects represented by a Fortran descriptor, and convert between a C descriptor and a corresponding Fortran descriptor. These provide a means to specify a C prototype that interoperates with a Fortran interface that has allocatable, data pointer, or assumed-shape dummy arguments.

2 Multiple inclusion of ISO_Fortran_binding.h within a translation unit shall have no effect, other than line numbers, different from just the first inclusion.

3 No names other than those specified shall be placed in the global namespace by inclusion of the file ISO_Fortran_binding.h.

5.3.2 CFI_cdesc_t

1 CFI_cdesc_t is a named struct type defined by a typedef. It shall contain at least the following members in any order:

void * base_addr; If the object is an unallocated allocatable or a pointer that is disassociated, the value is NULL. If the object has zero size, the value is processor-dependent. Otherwise, the value is the base address of the object being described. The base address of a scalar is its C address. The base address of an array is the C address of the element for which each subscript has the value of the corresponding lower bound.

size_t elem_len; equal to the sizeof() of an element of the object being described

int rank; equal to the number of dimensions of the object being described. If the object is a scalar, the value is zero.

int type; equal to the identifier for the type of the object. Each interoperable intrinsic C type has an identifier. The identifier for interoperable structures has a different value from any of the identifiers for intrinsic types. Macros and the corresponding values for the identifiers are supplied in the ISO_Fortran_binding.h file.

int attribute; equal to the value of an attribute code that indicates whether the object being described is a data pointer, allocatable, or assumed-shape. Macros and the corresponding values for the attribute codes are supplied in the ISO_Fortran_binding.h file.

int state; has the value 1 if the object is an allocated allocatable, an associated pointer, or assumed-shape, and 0 otherwise.

void * fdesc; points to the corresponding Fortran descriptor if one exists; otherwise the value is NULL.

CFI_dim_t dim[CFI_MAX_RANK]; Each element of the array contains the lower bound, extent, and stride multiplier information for the corresponding dimension of the object. CFI_MAX_RANK is a macro defined in the file ISO_Fortran_binding.h. The number of elements actually used is equal to the rank of the object. This member is not used if the object is a scalar.

5.3.3 CFI_dim_t

1 CFI_dim_t is a named struct type defined by a typedef. It is used to represent lower bound, extent, and stride multiplier information for one dimension of an array. It is defined in the file ISO_Fortran_binding.h, and contains at least the following members in any order:

size_t lower_bound; equal to the value of the lower bound of an array for a specified dimension.

size_t extent; equal to the number of elements of an array along a specified dimension.

size_t sm; equal to the stride multiplier for a dimension. The value is the distance in bytes between the beginnings of successive elements of the array along a specified dimension.

5.3.4 CFI_bounds_t

1 CFI_bounds_t is a named struct type defined by a typedef. It is used to represent bounds and stride information for one dimension of an array. It is defined in the file ISO_Fortran_binding.h, and contains at least the following members in any order:

size_t lower_bound; equal to the value of the lower bound of an array for a specified dimension.

size_t upper_bound; equal to the value of the upper bound of an array for a specified dimension.

size_t stride; equal to the difference between the subscript values of consecutive elements of an array along a specified dimension.

5.3.5 Macros

1 The following macros are defined in ISO_Fortran_bindings.h. Each evalutates to an integer constant expression.

2 CFI_MAX_RANK - a value equal to the largest rank supported. The value shall be greater than or equal to 15.

3 The macros in Table 5.1 are for use as attribute codes. The values shall be nonnegative and distinct.

Table 5.1: **Macros specifying attribute codes**

Macro	Code
CFI_attribute_assumed	assumed-shape array
CFI_attribute_allocatable	allocatable object
CFI_attribute_pointer	pointer

4 The macros in Table 5.2 are for use as type specifiers. The value for CFI_type_struct shall be distinct from all the other type specifiers. If an intrinsic C type is not interoperable with a Fortran type and kind supported by the companion processor, its macro shall evaluate to a negative value. Otherwise, the value for an intrinsic type shall be positive.

Table 5.2: **Macros specifying type codes**

Macro	C Type
CFI_type_struct	interoperable struct
CFI_type_signed_char	signed char
CFI_type_short	short
CFI_type_int	int
CFI_type_long	long
CFI_type_long_long	long long
CFI_type_size_t	size_t
CFI_type_int8_t	int8_t
CFI_type_int16_t	int16_t
CFI_type_int32_t	int32_t
CFI_type_int64_t	int64_t
CFI_type_int_least8_t	least8_t
CFI_type_int_least16_t	least16_t
CFI_type_int_least32_t	least32_t
CFI_type_int_least64_t	least64_t
CFI_type_int_fast8_t	fast8_t
CFI_type_int_fast16_t	fast16_t
CFI_type_int_fast32_t	fast32_t
CFI_type_int_fast64_t	fast64_t
CFI_type_intmax_t	intmax_t
CFI_type_intptr_t	intptr_t
CFI_type_float	float
CFI_type_double	double
CFI_type_long_double	long double
CFI_type_float_Complex	float Complex
CFI_type_double_Complex	double Complex
CFI_type_long_double_Complex	long double Complex
CFI_type_Bool	Bool
CFI_type_char	char
CFI_type_cptr	void *
CFT_type_cfunptr	pointer to a function

NOTE 5.3

The specifiers for two intrinsic types may have the same value. For example, CFI_type_int and CFI_type_int32_t might have the same value.

5.3.6 Functions

1 Eight functions are provided for use in C functions. These functions and the structure of the C descriptor provide the C program with the capability to interact with Fortran procedures that have allocatable, data pointer, assumed-rank, or assumed-shape arguments.

2 Within a C function, allocatable objects shall be allocated or deallocated only through execution of the CFI_allocate and CFI_deallocate functions. Pointer objects may become associated with a target by execution of the CFI_allocate function.

3 Each function returns an `int` value. If an error occurs during execution of the function the returned value is nonzero; otherwise zero is returned. Errors might occur because values supplied in an argument are invalid for that function, or a memory allocation failed. Which errors are detected and the corresponding return values are processor dependent. Prototypes for these functions are provided in the `ISO_Fortran_binding.h` file as follows:

5.3.6.1 int CFI_update_cdesc (CFI_cdesc_t *);

1 **Description.** CFI_update_cdesc updates a C descriptor based on information in the corresponding Fortran descriptor. The Fortran descriptor is not modified.

5.3.6.2 int CFI_update_fdesc (CFI_cdesc_t *);

1 **Description.** CFI_update_fdesc creates or updates a Fortran descriptor based on information in the corresponding C descriptor. The C descriptor is not modified. If the address of the Fortran descriptor is NULL, then a new Fortran descriptor is created.

5.3.6.3 int CFI_free_fdesc (CFI_cdesc_t *);

1 **Description.** CFI_free_fdesc destroys the Fortran descriptor pointed to by the fdesc member of the argument that was created by a CFI_update_fdesc.

5.3.6.4 int CFI_allocate (CFI_cdesc_t *, const CFI_bounds_t bounds[]);

1 **Description.** CFI_allocate allocates memory for an object using the same mechanism as the Fortran ALLOCATE statement. On entry, the base address in the C descriptor shall be NULL. The corresponding Fortran descriptor shall be for an unallocated allocatable or disassociated pointer data object. The supplied bounds override any current dimension information in the descriptors. The number of elements in the bounds array shall be greater than or equal to the rank specified in the desriptor. The stride values are ignored and assumed to be one. Both the Fortran and C descriptors are updated by this function.

5.3.6.5 int CFI_deallocate (CFI_cdesc_t *);

1 **Description.** CFI_deallocate deallocates memory for an object that was allocated using the same mechanism as the Fortran ALLOCATE statement. It uses the same mechanism as the Fortran DEALLOCATE statement. On entry, the base address in the C descriptor shall not be NULL. The corresponding Fortran descriptor shall be for the same object and shall be for an allocated allocatable object, or a pointer associated with a target that was allocated using CFI_allocate or the Fortran ALLOCATE statement. Both the Fortran and C descriptors are updated by this function.

5.3.6.6 int CFI_is_contiguous (const CFI_cdesc_t *, _Bool * result);

1 **Description.** CFI_is_contiguous defines result as true if the object described by the C descriptor is contiguous in memory, and false otherwise. If the object is allocatable it shall be allocated. If it is a pointer it shall be associated.

5.3.6.7 int CFI_bounds_to_cdesc (const CFI_bounds_t bounds[], CFI_cdesc_t *);

1 **Description.** CFI_bounds_to_cdesc computes a set of extent and stride multiplier values in a C descriptor given a corresponding set of lower bound, upper bound, and stride values in the bounds array. The number of elements in the bounds array shall be greater than or equal to the rank specified in the descriptor. The lower bounds in the C descriptor become those in the input bounds array. Since computation of stride multipliers requires the element size, the whole C descriptor is used as one of the arguments. If there is a corresponding Fortran descriptor, it is updated to reflect the same bounds, extend, and stride multiplier information.

5.3.6.8 int CFI_cdesc_to_bounds (const CFI_cdesc_t *, CFI_bounds_t bounds[]);

1 **Description.** CFI_cdesc_to_bounds computes a set of upper bound and stride values based on the extent and stride multiplier values in a C descriptor. The number of elements in the bounds array shall be equal to or greater than the rank specified in the descriptor. The lower bounds in the bounds array become those in the input C descriptor. Since computation of strides from stride multipliers requires the element size, the whole C descriptor is used as one of the arguments.

5.3.7 Restrictions on the use of C descriptors

1 The base address in the C descriptor for a data pointer may be modified by assignment and that change later affected in the corresponding Fortran descriptor by the CFI_update_fdesc function. The base address in the C descriptor for an allocatable object may be initialized to NULL and its value shall be modified only by the CFI_allocate, CFI_deallocate, or CFI_update_cdesc functions. If the base address of an object that is neither a data pointer nor an allocatable object is changed from its initial value, the corresponding Fortran descriptor shall not be modified.

2 It is possible for a C function to acquire memory through a function such as malloc and associate that memory with a data pointer in a C descriptor. A C descriptor associated with such memory shall not be supplied as an argument to CFI_deallocate and a corresponding dummy argument in a called Fortran procedure shall not be specified in a context that would cause the dummy argument to be deallocated. The memory may be released by reference to the free library function in a C function.

3 If a Fortran descriptor is created by a C function, the memory for the descriptor may be released by a reference to the free library function in a C function when the descriptor is no longer needed.

5.3.8 Interoperability of procedures and procedure interfaces

1 A Fortran procedure is **interoperable** if it has the BIND attribute, that is, if its interface is specified with a *proc-language-binding-spec*.

2 A Fortran procedure interface is interoperable with a C function prototype if

(1) the interface has the BIND attribute,
(2) either
  (a) the interface describes a function whose result variable is a scalar that is interoperable with the result of the prototype or
  (b) the interface describes a subroutine and the prototype has a result type of void,
(3) the number of dummy arguments of the interface is equal to the number of formal parameters of the prototype,
(4) the prototype does not have variable arguments as denoted by the ellipsis (...),
(5) any dummy argument with the VALUE attribute is interoperable with the corresponding formal parameter of the prototype, and
(6) any dummy argument without the VALUE attribute corresponds to a formal parameter of the prototype that is of a pointer type, and either
  (a) the dummy argument is interoperable with an entity of the referenced type (C International Standard, 6.2.5, 7.17, and 7.18.1) of the formal parameter, or
  (b) the dummy argument is allocatable, assumed-shape, assumed-rank, or a pointer, and corresponds to a formal parameter of the prototype that is a pointer to void.

3 If a dummy argument in an interoperable interface is allocatable, assumed-shape, or a pointer, the corresponding formal parameter is interpreted as a pointer to a Fortran descriptor for the effective argument in a reference to the procedure. The Fortran descriptor shall describe an object of interoperable type and type parameters with the same characteristics as the effective argument.

4 An absent actual argument in a reference to an interoperable procedure is indicated by a correpsonding formal parameter with the value NULL.

Annex A

(Informative)

Extended notes

A.1 Clause 2 notes

A.1.1 Using assumed-type dummy arguments

Example of TYPE (*) for an abstracted MPI routine with two arguments.

1 The first argument is a data buffer of type (void *) and the second is an integer indicating the size of the buffer. The generic interface allows for both 4 and 8 byte integers, as a solution to the "-i8" compiler switch problem.

2 In C:

```
3 void MPI_xxx ( void * buffer, int n);
```

4 In the Fortran MPI module:

```
5 interface MPI_xxx
     subroutine MPI_xxx (buffer, n) bind(c,name=''MPI_xxx'')
        type(*),dimension(*) :: buffer
        integer(c_int),value :: n
     end subroutine MPI_xxx
     module procedure MPI_xxx_i8
  end interface MPI_xxx

  ...

  subroutine MPI_xxx_i8 (buffer, n)
     type(*),dimension(*) :: buffer
     integer(selected_int_kind(17)) :: n
     call MPI_xxx(buffer, int(n,c_int))
  end subroutine MPI_xxx_i8
```

A.1.2 General association with a void * C parameter

Example of assumed-type and assumed-rank for an abstracted MPI_send routine.

1 In C:

```
2        void MPI_send_abstract ( void * buffer, int n);
         void MPI_send_abstract_new ( void * buffer_desc);
```

3 In the Fortran MPI module:

```
4 interface MPI_send_abstract
     subroutine MPI_send_old (buffer, n) bind(c,name=''MPI_send_abstract'')
        type(*), dimension(*) :: buffer ! Passed by simple address
        integer(c_int),value :: n
```

```
   end subroutine
   subroutine MPI_send_new (buffer) bind(c,name=''MPI_send_abstract_new'')
      type(*), dimension(..), contiguous :: buffer
         ! Passed by descriptor including the shape and type
   end subroutine
end interface

real :: x(100), y(10,10)

! These will invoke MPI_send_old
call MPI_send_abstract(x,c_sizeof(x)) ! Passed by address
call MPI_send_abstract(y,c_sizeof(y)) ! Sequence association
call MPI_send_abstract(y(:,1),size(y,dim=1)*c_sizeof(y(1,1)) ! Pass first column of y
call MPI_send_abstract(y(1,5),size(y,dim=1)*c_sizeof(y(1,1)) ! Pass fifth column of y

! These will invoke MPI_send_new
call MPI_send_abstract(x) ! Pass a rank-1 descriptor
call MPI_send_abstract(y) ! Pass a rank-2 descriptor
call MPI_send_abstract(y(:,1)) ! Passed by descriptor without copy
call MPI_send_abstract(y(1,5)) ! Pass a rank-0 descriptor
```

A.1.3 Casting TYPE (*) in Fortran

Example of how to gain access to a TYPE (*) argument

1 It is possible to "cast" a TYPE (*) object to a usable type, exactly as is done for void * objects in C. For example, this code fragment casts a block of memory to be used as an integer array.

2
```
subroutine process(block, nbytes)
   type(*), target :: block(*)
   integer, intent(in) :: nbytes ! Number of bytes in block(*)

   integer :: nelems
   integer, pointer :: usable(:)

   nelems=nbytes/(bit_size(usable)/8)
   call c_f_pointer (c_loc(block), usable, [nelems] )
   usable=0 ! Instead of the disallowed block=0
end subroutine
```

A.1.4 Simplifying interfaces for arbitrary rank procedures

Example of assumed-rank usage in Fortran

1 Assumed-rank variables are not restricted to be assumed-type. For example, many of the IEEE intrinsic procedures in Clause 14 of ISO/IEC 1539-1:2010 could be written using an assumed-rank dummy argument instead of writing 16 separate specific routines, one for each possible rank.

2 An example of an assumed-rank dummy argument for the specific procedures for the IEEE_SUPPORT_DIVIDE function.

3
```
interface ieee_support_divide
    module procedure ieee_support_divide_noarg
    module procedure ieee_support_divide_onearg_r4
    module procedure ieee_support_divide_onearg_r8
```

```
end interface ieee_support_divide

...

logical function ieee_support_divide_noarg ()
    ieee_support_divide_noarg = .true.
end function ieee_support_divide_onearg_r4

logical function ieee_support_divide_onearg_r4 (x)
    real(4),dimension(..) :: x
    ieee_support_divide_onearg_r4 = .true.
end function ieee_support_divide_onearg_r4

logical function ieee_support_divide_onearg_r8 (x)
    real(8),dimension(..) :: x
    ieee_support_divide_onearg_r8 = .true.
end function ieee_support_divide_onearg_r8
```

A.2 Clause 5 notes

1 NOTE: Do we want to add examples here?

Annex B

(Informative)

Index

In this annex, entries in *italics* denote BNF terms, and page numbers in **bold face** denote primary text or definitions.