TS 29113 Further Interoperability of Fortran with C

WG5/N1904

30th January 2012 17:51

Draft document for DTS Ballot

Contents

List of Tables

Foreword

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.

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

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.

In other circumstances, particularly when there is an urgent market requirement for such documents, the joint technical committee may decide to publish an ISO/IEC Technical Specification (ISO/IEC TS), which represents an agreement between the members of the joint technical committee and is accepted for publication if it is approved by 2/3 of the members of the committee casting a vote.

An ISO/IEC TS is reviewed after three years in order to decide whether it will be confirmed for a further three years, revised to become an International Standard, or withdrawn. If the ISO/IEC TS is confirmed, it is reviewed again after a further three years, at which time it must either be transformed into an International Standard or be withdrawn.

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.

ISO/IEC TS 29113:2012 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC22, *Programming languages, their environments and system software interfaces.*

Introduction

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.

The existing system for interoperability does not provide for interoperability of interfaces with Fortran dummy arguments that are assumed-shape arrays, have assumed character length, or have the ALLOCATABLE, POINTER, or OPTIONAL attributes. As a consequence, a significant class of Fortran subprograms is not portably accessible from C, limiting the usefulness of the facility.

The provision in the existing system for interoperability with a C formal parameter that is a pointer to void is inconvenient to use and error-prone. C functions with such parameters are widely used.

This Technical Specification extends the facility of Fortran for interoperating with C to provide for interoperability of procedure interfaces that specify dummy arguments that are assumed-shape arrays, have assumed character length, or have the ALLOCATABLE, POINTER, or OPTIONAL attributes. New Fortran concepts of assumed type and assumed rank are introduced. The former simplifies interoperation with formal parameters of type (void *). The latter facilitates interoperability with C functions that can accept arguments of arbitrary rank. An intrinsic function, RANK, is specified to obtain the rank of an assumed-rank variable.

The facility specified in this Technical Specification 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.

It is the intention of ISO/IEC JTC 1/SC22 that the semantics and syntax specified by this Technical Specification be included in the next revision of ISO/IEC 1539-1 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.

This Technical Specification is organized in 9 clauses:

Scope	Clause 1
Normative references	Clause 2
Terms and definitions	Clause 3
Compatibility	Clause 4
Type specifiers and attributes	Clause 5
Procedures	Clause 6
New intrinsic procedure	Clause 7
Interoperability with C	Clause 8
Required editorial changes to ISO/IEC 1539-1:2010(E)	Clause 9

It also contains the following nonnormative material:

Extended notes	Annex A

1 Scope

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

2 Normative references

The following referenced standards are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

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

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

3 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 1539-1:2010 and the following apply.

3.1
assumed-rank object
dummy variable whose rank is assumed from its effective argument

3.2
assumed-type object
dummy variable declared with the TYPE(*) type specifier

3.3
C descriptor
C structure of type CFI_cdesc_t

NOTE 3.1
A C descriptor is used to describe a Fortran object that has no exact analog in C.

4 Compatibility

4.1 New intrinsic procedures

This Technical Specification defines an intrinsic procedure 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 this Technical Specification if it invokes an external procedure having the same name as the new intrinsic procedure, unless that procedure is specified to have the EXTERNAL attribute.

4.2 Fortran 2008 compatibility

This Technical Specification specifies an upwardly compatible extension to ISO/IEC 1539-1:2010.

5 Type specifiers and attributes

5.1 Assumed-type objects

The syntax rule R403 *declaration-type-spec* in subclause 4.3.1.1 of ISO/IEC 1539-1:2010 is replaced by

R403	*declaration-type-spec*	**is**	*intrinsic-type-spec*
		or	TYPE (*intrinsic-type-spec*)
		or	TYPE (*derived-type-spec*)
		or	CLASS (*derived-type-spec*)
		or	CLASS (*)
		or	TYPE (*)

An entity declared with a *declaration-type-spec* of TYPE (*) is an assumed-type entity. It has no declared type and its dynamic type and type parameters are assumed from its effective argument.

C407a An assumed-type entity shall be a dummy variable that does not have the ALLOCATABLE, CODIMENSION, POINTER, or VALUE attribute and is not an explicit-shape array.

C407b An assumed-type variable name shall not appear in a designator or expression except as an actual argument corresponding to a dummy argument that is assumed-type, or as the first argument to any of the intrinsic and intrinsic module functions IS_CONTIGUOUS, LBOUND, PRESENT, RANK, SHAPE, SIZE, UBOUND, and C_LOC.

C407c An assumed-type actual argument that corresponds to an assumed-rank dummy argument shall be assumed-shape or assumed-rank.

An assumed-type object is unlimited polymorphic.

NOTE 5.1

An assumed-type object that is not assumed-shape and not assumed-rank is intended to be passed as the C address of the object. This means that there would be insufficient information passed for an assumed-type explicit-shape array that is an actual argument corresponding to an assumed-shape dummy argument. Therefore TYPE(*) explicit-shape is not permitted.

NOTE 5.2

This Technical Specification provides no mechanism for a Fortran procedure to determine the actual type of an assumed-type argument.

5.2 Assumed-rank objects

The syntax rule R515 *array-spec* in subclause 5.3.8.1 of ISO/IEC 1539-1:2010 is replaced by

R515	*array-spec*	**is**	*explicit-shape-spec-list*
		or	*assumed-shape-spec-list*
		or	*deferred-shape-spec-list*
		or	*assumed-size-spec*
		or	*implied-shape-spec-list*
		or	*assumed-rank-spec*

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

R522a *assumed-rank-spec* **is** ..

C535a An assumed-rank entity shall be a dummy variable that does not have the CODIMENSION or VALUE attribute.

An assumed-rank object may have the CONTIGUOUS attribute.

C535b An assumed-rank variable name shall not appear in a designator or expression except as an actual argument corresponding to 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 intrinsic inquiry function RANK can be used to inquire about the rank of a data object. The rank of an assumed-rank object is zero if the rank of the corresponding actual argument is zero.

The definition of TKR compatible in paragraph 2 of subclause 12.4.3.4.5 of ISO/IEC 1539-1:2010 is changed to:

> A dummy argument is type, kind, and rank compatible, or TKR compatible, with another dummy argument if the first is type compatible with the second, the kind type parameters of the first have the same values as the corresponding kind type parameters of the second, and both have the same rank or either is assumed-rank.

NOTE 5.3

Assumed rank is an attribute of a Fortran dummy argument. When a C function is invoked with an actual argument that corresponds to an assumed-rank dummy argument in a Fortran interface for that C function, the corresponding formal parameter is the address of a descriptor of type CFI_cdesc_t (8.7). The rank member of the descriptor provides the rank of the actual argument. The C function should therefore be able to handle any rank. On each invocation, the rank is available to it.

5.3 ALLOCATABLE, OPTIONAL, and POINTER attributes

The ALLOCATABLE, OPTIONAL, and POINTER attributes may be specified for a dummy argument in a procedure interface that has the BIND attribute.

The constraint C1255 in subclause 12.6.2.2 of ISO/IEC 1539-1:2010 is replaced by

C1255 (R1229) If *proc-language-binding-spec* is specified for a procedure, each dummy argument of the procedure shall be an interoperable procedure (15.3.7) or an interoperable variable (15.3.5, 15.3.6) that does not have both the OPTIONAL and VALUE attributes. If *proc-language-binding-spec* is specified for a function, the function result shall be an interoperable scalar variable.

Constraint C516 in subclause 5.3.1 of ISO/IEC 1539-1:2010 says "The ALLOCATABLE, POINTER, or OPTIONAL attribute shall not be specified for a dummy argument of a procedure that has a *proc-language-binding-spec*." This is replaced by the much less restrictive constraint:

C516 The ALLOCATABLE or POINTER attribute shall not be specified for a default-initialized dummy argument of a procedure that has a *proc-language-binding-spec*.

NOTE 5.4

It would be a severe burden to implementors to require that CFI_allocate initialize components of an object of a derived type with default initialization. The alternative of not requiring initialization would have been inconsistent with the effect of ALLOCATE in Fortran.

5.4 ASYNCHRONOUS attribute

5.4.1 Introduction

The ASYNCHRONOUS attribute is extended to apply to variables that are used for asynchronous communication.

5.4.2 Asynchronous communication

Asynchronous communication for a Fortran variable occurs through the action of procedures defined by means other than Fortran. It is initiated by execution of an asynchronous communication initiation procedure and completed by execution of an asynchronous communication completion procedure. Between the execution of the initiation and completion procedures, any variable of which any part is associated with any part of the asynchronous communication variable is a pending communication affector. Whether a procedure is an asynchronous communication initiation or completion procedure is processor dependent.

Asynchronous communication is either input communication or output communication. For input communication, a pending communication affector shall not be referenced, become defined, become undefined, become associated with a dummy argument that has the VALUE attribute, or have its pointer association status changed. For output communication, a pending communication affector shall not be redefined, become undefined, or have its pointer association status changed.

NOTE 5.5

Asynchronous communication can be used for nonblocking MPI calls such as MPI_IRECV and MPI_ISEND. For example,

```
REAL :: BUF(100,100)
   ... ! Code that involves BUF
BLOCK
  ASYNCHRONOUS :: BUF
  CALL MPI_IRECV(BUF,...REQ,...)
     ... ! Code that does not involve BUF
  CALL MPI_WAIT(REQ,...)
END BLOCK
   ... ! Code that involves BUF
```

In this example, there is asynchronous input communication and BUF is a pending communication affector between the two calls. MPI_IRECV may return while the communication (reading values into BUF) is still underway. The intent is that the code between MPI_IRECV and MPI_WAIT executes without waiting for this communication to complete. The restrictions are the same as for asynchronous input data transfer.

Similar code with the call of MPI_IRECV replaced by a call of MPI_ISEND is asynchronous output communication. The restrictions are the same as for asynchronous output data transfer.

6 Procedures

6.1 Characteristics of dummy data objects

Additionally to the characteristics listed in subclause 12.3.2.2 of ISO/IEC 1539-1:2010, whether the type or rank of a dummy data object is assumed is a characteristic of the dummy data object.

6.2 Explicit interface

Additionally to the rules of subclause 12.4.2.2 of ISO/IEC 1539-1:2010, a procedure shall have an explicit interface if it has a dummy argument that is assumed-rank.

NOTE 6.1

An explicit interface is also required for a procedure if it has a dummy argument that is assumed-type because an assumed-type dummy argument is polymorphic.

6.3 Argument association

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; the shape is a zero-sized array and the LBOUND and UBOUND intrinsic functions, with no DIM argument, return zero-sized arrays. If the actual argument is an array, the rank and extents of the dummy argument are assumed from the actual argument, including the lack of a final extent in the case of an assumed-size array. If the actual argument is an array and the dummy argument is allocatable or a pointer, the bounds of the dummy argument are assumed from the actual argument.

An assumed-type dummy argument shall not correspond to an actual argument that is of a derived type that has type parameters, type-bound procedures, or final procedures.

If a Fortran procedure that has an INTENT(OUT) allocatable dummy argument is invoked by a C function, and the actual argument in the C function is the address of a C descriptor that describes an allocated allocatable variable, the variable is deallocated on entry to the Fortran procedure.

When a C function is invoked from a Fortran procedure via an interface with an INTENT(OUT) allocatable dummy argument, and the actual argument in the reference to the C function is an allocated allocatable variable, the variable is deallocated on invocation (before execution of the C function begins).

6.4 Intrinsic procedures

6.4.1 SHAPE

The description of the intrinsic function SHAPE in ISO/IEC 1539-1:2010 is changed for an assumed-rank array that is associated with an assumed-size array; an assumed-size array has no shape, but in this case the result has a value equal to [(SIZE (ARRAY, I, KIND), I=1, RANK (ARRAY))] with KIND omitted from SIZE if it was omitted from SHAPE.

6.4.2 SIZE

The description of the intrinsic function SIZE in ISO/IEC 1539-1:2010 is changed in the following cases:

(1) for an assumed-rank object that is associated with an assumed-size array, the result has the value −1 if DIM is present and equal to the rank of ARRAY, and a negative value that is equal to PRODUCT ([(SIZE (ARRAY, I, KIND), I=1, RANK (ARRAY))]) if DIM is not present;

(2) for an assumed-rank object that is associated with a scalar, the result has the value 1.

6.4.3 UBOUND

The description of the intrinsic function UBOUND in ISO/IEC 1539-1:2010 is changed for an assumed-rank object that is associated with an assumed-size array; the result of UBOUND (ARRAY, RANK(ARRAY), KIND) has a value equal to LBOUND (ARRAY, RANK (ARRAY), KIND) −2 with KIND omitted from LBOUND if it was omitted from UBOUND.

NOTE 6.2

If LBOUND or UBOUND is invoked for an assumed-rank object that is associated with a scalar and DIM is absent, the result is a zero-sized array. LBOUND or UBOUND cannot be invoked for an assumed-rank object that is associated with a scalar if DIM is present because the rank of a scalar is zero and DIM must be ≥ 1.

7 New intrinsic procedure

7.1 General

Detailed specification of the generic intrinsic function RANK is provided in 7.2. 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 intrinsic function RANK is pure.

7.2 RANK (A)

Description. Rank of a data object.

Class. Inquiry function.

Argument.

A	shall be a scalar or array of any type.

Result Characteristics. Default integer scalar.

Result Value. The result is the rank of A.

Example. If X is declared REAL X(:,:,:), RANK(X) has the value 3.

8 Interoperability with C

8.1 Removed restrictions on C_F_POINTER, C_LOC, and C_FUNLOC

The subroutine C_F_POINTER from the intrinsic module ISO_C_BINDING has the restriction in ISO/IEC 1539-1:2010 that if FPTR is an array, it shall be of interoperable type.

The function C_LOC from the intrinsic module ISO_C_BINDING has the restriction in ISO/IEC 1539-1:2010 that if X is an array, it shall be of interoperable type.

The function C_FUNLOC from the intrinsic module ISO_C_BINDING has the restriction in ISO/IEC 1539-1:2010 that its argument shall be interoperable.

These restrictions are removed.

8.2 C descriptors

A C descriptor is a C structure of type CFI_cdesc_t. The C descriptor along with library functions with standard prototypes provides the means for describing within a C function an allocatable or data pointer object, a non-allocatable nonpointer data object of known shape, or an assumed-size array. This C structure is defined in the file `ISO_Fortran_binding.h`.

8.3 ISO_Fortran_binding.h

8.3.1 Summary of contents

The `ISO_Fortran_binding.h` file contains the definitions of the C structures CFI_cdesc_t and CFI_dim_t, typedef definitions for CFI_attribute_t, CFI_index_t, CFI_rank_t, and CFI_type_t, the definition of the macro CFI_CDESC_T, macro definitions that expand to integer constants, and C prototypes for the C macro and functions CFI_address, CFI_allocate, CFI_deallocate, CFI_establish, CFI_is_contiguous, CFI_section, CFI_select_part, and CFI_setpointer. The contents of `ISO_Fortran_binding.h` can be used by a C function to interpret a C descriptor and allocate and deallocate objects represented by a C descriptor. These provide a means to specify a C prototype that interoperates with a Fortran interface that has an allocatable, assumed character length, assumed-rank, assumed-shape, or data pointer dummy argument.

`ISO_Fortran_binding.h` may be included in any order relative to the standard C headers, and may be included more than once in a given scope, with no effect different from being included only once, other than the effect on line numbers.

A C source file that includes the header `ISO_Fortran_binding.h` shall not use any names starting with CFI_ that are not defined in the header. All names defined in the header begin with CFI_ or an underscore character, or are defined by a standard C header that it includes.

8.3.2 CFI_dim_t

CFI_dim_t is a named C structure type defined by a typedef. It is used to represent lower bound, extent, and memory stride information for one dimension of an array. CFI_index_t is a typedef name for a standard signed integer type capable of representing the result of subtracting two pointers. CFI_dim_t contains at least the following members in any order:

CFI_index_t lower_bound; equal to the value of the lower bound for the dimension being described.

CFI_index_t extent; equal to the number of elements along the dimension being described, or the value -1 for the final dimension of an assumed-size array.

CFI_index_t sm; equal to the memory stride for a dimension. The value is the distance in bytes between the beginnings of successive elements along the dimension being described.

8.3.3 CFI_cdesc_t

CFI_cdesc_t is a named C structure type defined by a typedef, containing a flexible array member. It shall contain at least the following members. The first three members of the structure shall be `base_addr`, `elem_len`, and `version` in that order. The final member shall be `dim`, with the other members after `version` and before `dim` in any order.

void * base_addr; If the object is an unallocated allocatable or a pointer that is disassociated, the value is a null pointer. If the object has zero size, the value is not a null pointer but is otherwise 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 first element in Fortran array element order (6.5.3.2 of ISO/IEC 1539-1:2010).

size_t elem_len; If the object corresponds to a Fortran CHARACTER object, the value is equal to the length of the CHARACTER object times the `sizeof()` of a scalar of the character type and kind; otherwise, the value is equal to the `sizeof()` of an element of the object.

int version; shall be set equal to the value of CFI_VERSION in the `ISO_Fortran_binding.h` header file that defined the format and meaning of this C descriptor when the descriptor is established and otherwise not changed.

CFI_rank_t rank; equal to the number of dimensions of the Fortran object being described. If the object is a scalar, the value is zero. CFI_rank_t shall be a typedef name for a standard integer type capable of representing the largest supported rank.

CFI_type_t type; equal to the specifier for the type of the object. Each interoperable intrinsic C type has a specifier. Specifiers are also provided to indicate that the type of the object is an interoperable structure, or is unknown. The macros listed in Table 8.2 provide values that correspond to each specifier. CFI_type_t shall be a typedef name for a standard integer type capable of representing the values for the supported type specifiers.

CFI_attribute_t attribute; equal to the value of an attribute code that indicates whether the object described is allocatable, a data pointer, or a nonallocatable nonpointer data object. The macros listed in Table 8.1 provide values that correspond to each specifier. CFI_attribute_t shall be a typedef name for a standard integer type capable of representing the values of the attribute codes.

CFI_dim_t dim[ ]; Each element of the `dim` array contains the lower bound, extent, and memory stride information for the corresponding dimension of the Fortran object. The number of elements in the array shall be equal to the rank of the object.

For a C descriptor of an array pointer or allocatable array, the value of the `lower_bound` member of each element of the `dim` member of the descriptor is determined by argument association, allocation, or pointer association. For a C descriptor of a nonallocatable nonpointer object, the value of the `lower_bound` member of each element of the `dim` member of the descriptor is zero.

There shall be an ordering of the dimensions such that the absolute value of the `sm` member of the first dimension is not less than the `elem_len` member of the descriptor and the absolute value of the `sm` member of each subsequent dimension is not less than the absolute value of the `sm` member of the previous dimension multiplied by the extent of the previous dimension.

In a C descriptor of an assumed-size array, the `extent` member of the last element of the `dim` member has the value -1.

NOTE 8.1

The reason for the restriction on the absolute values of the `sm` members is to ensure that there is no overlap between the elements of the array that is being described, while allowing for the reordering of subscripts. Within Fortran, such a reordering can be achieved with the intrinsic function TRANSPOSE or the intrinsic function RESHAPE with the optional argument ORDER, and an optimizing compiler can accommodate it without making a copy by constructing the appropriate descriptor whenever it can determine that a copy is not needed.

NOTE 8.2

If the type of the Fortran object is CHARACTER with kind C_CHAR, the value of the `elem_len` member will be equal to the character length.

8.3.4 Macros

The macros described in this subclause are defined in `ISO_Fortran_binding.h`. Except for CFI_CDESC_T, each expands to an integer constant expression suitable for use in `#if` preprocessing directives.

CFI_CDESC_T is a function-like macro that takes one argument, which is the rank of the C descriptor to create, and evaluates to a type suitable for declaring a C descriptor of that rank. A pointer to a variable declared using CFI_CDESC_T can be cast to CFI_cdesc_t *. A variable declared using CFI_CDESC_T shall not have an initializer.

NOTE 8.3

The CFI_CDESC_T macro provides the memory for a C descriptor. The address of an entity declared using the macro is not usable as an actual argument corresponding to a formal parameter of type CFI_cdesc_t * without an explicit cast. For example, the following code uses CFI_CDESC_T to declare a C descriptor of rank 5 and pass it to CFI_deallocate (8.3.5.4).

```
CFI_CDESC_T(5) object;
int ind;
... code to define and use C descriptor ...
ind = CFI_deallocate((CFI_cdesc_t *) &object);
```

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

CFI_VERSION has a processor-dependent value that encodes the version of the `ISO_Fortran_binding.h` header file containing this macro.

NOTE 8.4

The intent is that the version should be increased every time that the header is incompatibly changed, and that the version in a C descriptor may be used to provide a level of upwards compatibility, by using means not defined by this Technical Specification.

The macros in Table 8.1 are for use as attribute codes. The values shall be nonnegative and distinct.

Table 8.1: **Macros specifying attribute codes**

Macro	Code
CFI_attribute_pointer	data pointer
CFI_attribute_allocatable	allocatable
CFI_attribute_other	nonallocatable nonpointer

CFI_attribute_pointer specifies a data object with the Fortran POINTER attribute. CFI_attribute_allocatable specifies an object with the Fortran ALLOCATABLE attribute. CFI_attribute_other specifies an assumed-shape array, a nonallocatable nonpointer scalar, an assumed-size array, or an array that is argument associated with an assumed-size array.

The macros in Table 8.2 are for use as type specifiers. The value for CFI_type_other shall be negative and distinct from all other type specifiers. CFI_type_struct specifies a C structure that is interoperable with a Fortran derived type; its value shall be positive and distinct from all other type specifiers. If a C type is not interoperable with a Fortran type and kind supported by the Fortran processor, its macro shall evaluate to a negative value. Otherwise, the value for an intrinsic type shall be positive.

Additional nonnegative processor-dependent type specifier values may be defined for Fortran intrinsic types that are not represented by other type specifiers and noninteroperable Fortran derived types that do not have type parameters, type-bound procedures, final procedures, nor components that have the ALLOCATABLE or POINTER attributes, or correspond to CFI_type_other.

Table 8.2: **Macros specifying type codes**

Macro	C Type
CFI_type_signed_char	signed char
CFI_type_short	short int
CFI_type_int	int
CFI_type_long	long int
CFI_type_long_long	long long int
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	int_least8_t
CFI_type_int_least16_t	int_least16_t
CFI_type_int_least32_t	int_least32_t
CFI_type_int_least64_t	int_least64_t
CFI_type_int_fast8_t	int_fast8_t
CFI_type_int_fast16_t	int_fast16_t
CFI_type_int_fast32_t	int_fast32_t
CFI_type_int_fast64_t	int_fast64_t
CFI_type_intmax_t	intmax_t
CFI_type_intptr_t	intptr_t
CFI_type_ptrdiff_t	ptrdiff_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 *
CFI_type_cfunptr	pointer to a function
CFI_type_struct	interoperable C structure
CFI_type_other	Not otherwise specified

NOTE 8.5

The specifiers for two intrinsic types can have the same value. For example, CFI_type_int and CFI_type_-int32_t might have the same value.

The macros in Table 8.3 are for use as error codes. The macro CFI_SUCCESS shall be defined to be the integer constant 0. The value of each macro other than CFI_SUCCESS shall be nonzero and shall be different from the values of the other macros specified in this subclause. Error conditions other than those listed in this subclause should be indicated by error codes different from the values of the macros named in this subclause.

The error codes that indicate the following error conditions are named by the associated macro name.

Table 8.3: **Macros specifying error codes**

Macro	Error
CFI_SUCCESS	No error detected.
CFI_ERROR_BASE_ADDR_NULL	The base address member of a C descriptor is a null pointer in a context that requires a non-null pointer value.
CFI_ERROR_BASE_ADDR_NOT_NULL	The base address member of a C descriptor is not a null pointer in a context that requires a null pointer value.
CFI_INVALID_ELEM_LEN	The value supplied for the element length member of a C descriptor is not valid.
CFI_INVALID_RANK	The value supplied for the rank member of a C descriptor is not valid.
CFI_INVALID_TYPE	The value supplied for the type member of a C descriptor is not valid.
CFI_INVALID_ATTRIBUTE	The value supplied for the attribute member of a C descriptor is not valid.
CFI_INVALID_EXTENT	The value supplied for the extent member of a CFI_dim_t structure is not valid.
CFI_INVALID_DESCRIPTOR	A general error condition for C descriptors.
CFI_ERROR_MEM_ALLOCATION	Memory allocation failed.
CFI_ERROR_OUT_OF_BOUNDS	A reference is out of bounds.

8.3.5 Functions

8.3.5.1 General

The macro and functions described in this subclause and the structure of the C descriptor provide a C function with the capability to interoperate with a Fortran procedure that has an allocatable, assumed character length, assumed-rank, assumed-shape, or data pointer argument.

In function arguments representing subscripts, bounds, extents, or strides, the ordering of the elements is the same as the ordering of the elements of the `dim` member of a C descriptor.

Prototypes for these functions are provided in the `ISO_Fortran_binding.h` file as follows:

8.3.5.2 void * CFI_address (const CFI_cdesc_t * dv, const CFI_index_t subscripts[]);

Description. Compute the C address of an object described by a C descriptor.

Formal Parameters.

`dv` shall be the address of a C descriptor describing the object. The object shall not be an unallocated allocatable variable or a pointer that is not associated.

`subscripts` is ignored if the object is scalar. If the object is an array, `subscripts` is the address of a subscripts array. The number of elements shall be greater than or equal to the rank *r* of the object. The subscript values shall be within the bounds specified by the corresponding elements of the `dim` member of the C descriptor.

Result Value. If the object is an array of rank *r*, the result is the C address of the element of the object that the first *r* elements of the `subscripts` argument would specify if used as subscripts. If the object is scalar, the result is its C address.

NOTE 8.6

When the subscripts argument is ignored, its value may be either a null pointer or a valid pointer value, but it need not be the address of an object.

Example. If `dv` is the address of a C descriptor for the Fortran array `a` declared as

```
real(C_float) :: a(100,100)
```

the following code returns the C address of `a(5,10)`

```
CFI_index_t subscripts[2];
float *address;
subscripts[0] = 4;
subscripts[1] = 9;
address = (float *) CFI_address( dv, subscripts );
```

8.3.5.3 int CFI_allocate (CFI_cdesc_t * dv, const CFI_index_t lower_bounds[], const CFI_index_t upper_bounds[], size_t elem_len) ;

Description. Allocate memory for an object described by a C descriptor.

Formal Parameters.

`dv` shall be the address of a C descriptor specifying the rank and type of the object. On entry, the `base_addr` member of the C descriptor shall be a null pointer, and the `elem_len` member shall specify the element length for the type unless the type is a character type. The `attribute` member of the C descriptor shall have a value of CFI_attribute_allocatable or CFI_attribute_pointer.

`lower_bounds` is the address of a lower bounds array. The number of elements shall be greater than or equal to the rank *r* specified in the C descriptor.

`upper_bounds` is the address of an upper bounds array. The number of elements shall be greater than or equal to the rank *r* specified in the C descriptor.

`elem_len` is ignored unless the type specified in the C descriptor is a character type. If the object is of Fortran character type, the value of `elem_len` shall be the number of characters in an element of the object times the `sizeof()` of a scalar of the character type and kind.

Successful execution of CFI_allocate allocates memory for the object described by the C descriptor with the address dv using the same mechanism as the Fortran ALLOCATE statement. The first r elements of the lower_bounds and upper_bounds arguments provide the lower and upper Fortran bounds, respectively, for each corresponding dimension of the object. The supplied lower and upper bounds override any current dimension information in the C descriptor. If the rank is zero, the lower_bounds and upper_bounds arguments are ignored. If the type specified in the C descriptor is a character type, the supplied element length overrides the current element-length information in the descriptor.

If an error is detected, the C descriptor is not modified.

Result Value. The result is an error indicator.

Example. If dv is the address of a C descriptor for the Fortran array a declared as

```
real, allocatable :: a(:,:)
```

and the array is not allocated, the following code allocates it to be of shape [100, 500]

```
CFI_index_t lower[2], upper[2];
int ind;
size_t dummy = 0;
lower[0] = 1; lower[1] = 1;
upper[0] = 100; upper[1] = 500;
ind = CFI_allocate( dv, lower, upper, dummy );
```

8.3.5.4 int CFI_deallocate (CFI_cdesc_t * dv);

Description. Deallocate memory for an object described by a C descriptor.

Formal Parameters.

dv shall be the address of a C descriptor describing the object. It shall have been allocated using the same mechanism as the Fortran ALLOCATE statement. If the object is a pointer, it shall be associated with a target satisfying the conditions for successful deallocation by the Fortran DEALLOCATE statement (6.7.3.3 of ISO/IEC 1539-1:2010).

Successful execution of CFI_deallocate deallocates memory for the object using the same mechanism as the Fortran DEALLOCATE statement. The base_addr member of the C descriptor becomes a null pointer.

If an error is detected, the C descriptor is not modified.

Result Value. The result is an error indicator.

Example. If dv is the address of a C descriptor for the Fortran array a declared as

```
real, allocatable :: a(:,:)
```

and the array is allocated, the following code deallocates it

```
int ind;
ind = CFI_deallocate( dv );
```

8.3.5.5 int CFI_establish (CFI_cdesc_t * dv, void * base_addr, CFI_attribute_t attribute, CFI_type_t type, size_t elem_len, CFI_rank_t rank, const CFI_index_t extents[]);

Description. Establish a C descriptor.

Formal Parameters.

`dv` shall be the address of a C object large enough to hold a C descriptor of the rank specified by `rank`. It shall not have the same value as either a C formal parameter that corresponds to a Fortran actual argument or a C actual argument that corresponds to a Fortran dummy argument. It shall not be the address of a C descriptor that describes an allocated allocatable object.

`base_addr` shall be a null pointer or the base address of the object to be described. If it is not a null pointer, it shall be the address of a contiguous storage sequence that is appropriately aligned (ISO/IEC 9899:1999 3.2) for an object of the type specified by `type`.

`attribute` shall be one of the attribute codes in Table 8.1. If it is CFI_attribute_allocatable, `base_addr` shall be a null pointer.

`type` shall be one of the type codes in Table 8.2.

`elem_len` is ignored unless `type` is CFI_type_struct, CFI_type_other, or a character type. If the type is CFI_-type_struct or CFI_type_other, `elem_len` shall be greater than zero and equal to `sizeof()` for the type. If the type is a Fortran character type, the value of `elem_len` shall be the required character length times the `sizeof()` for the type.

`rank` specifies the rank. It shall be between 0 and CFI_MAX_RANK inclusive.

`extents` is ignored if the `rank` r is zero or if `base_addr` is a null pointer. Otherwise, it shall be the address of an array with r elements specifying the corresponding extents of the described array. These extents shall all be nonnegative.

Successful execution of CFI_establish updates the object with the address `dv` to be an established C descriptor for a nonallocatable nonpointer data object of known shape, an unallocated allocatable object, or a data pointer. If `base_addr` is not a null pointer, it is for a nonallocatable entity that is a scalar or a contiguous array; if the `attribute` argument has the value CFI_attribute_pointer, the lower bounds of the object described by `dv` are set to zero. If `base_addr` is a null pointer, the established C descriptor is for an unallocated allocatable, a disassociated pointer, or is a C descriptor that has the `attribute` CFI_attribute_other but does not describe a Fortran assumed-shape array. The properties of the object are given by the other arguments.

It is unspecified whether CFI_establish is a macro or an identifier declared with external linkage. If a macro definition is suppressed in order to access an actual function, the behavior is undefined.

If an error is detected, the object with the address `dv` is not modified.

Result Value. The function returns an error indicator.

NOTE 8.7

CFI_establish is used to initialize a C descriptor declared in C with CFI_CDESC_T before passing it to any other functions as an actual argument, in order to set the rank, attribute, type and element length.

NOTE 8.8

A C descriptor with `attribute` CFI_attribute_other and `base_addr` a null pointer can be used as the argument `result` in calls to CFI_section or CFI_select_part, which will produce a C descriptor for a nonallocatable nonpointer data object.

NOTE 8.9

This function is allowed to be a macro to provide extra implementation flexibility. For example, it could include the value of CFI_VERSION in the header used to compile the call to CFI_establish as an extra argument of the actual function used to establish the C descriptor.

Example 1. The following code fragment establishes a C descriptor for an unallocated rank-one allocatable array to pass to Fortran for allocation there.

```
CFI_rank_t rank;
CFI_CDESC_T(1) field;
int ind;
rank = 1;
ind = CFI_establish ( (CFI_cdesc_t *)  &field, NULL, CFI_attribute_allocatable,
                  CFI_type_double, 0, rank, NULL );
```

Example 2. Given the Fortran type definition

```
type, bind(c) :: t
  real(c_double) :: x
  complex(c_double_complex) :: y
end type
```

and a Fortran subprogram that has an assumed-shape dummy argument of type `t`, the following code fragment creates a descriptor `a_fortran` for an array of size 100 which can be used as the actual argument in an invocation of the subprogram from C:

```
typedef struct {double x; double _Complex y;} t;
t a_c[100];
CFI_CDESC_T(1) a_fortran;
int ind;
CFI_index_t extent[1];

extent[0] = 100;
ind = CFI_establish( (CFI_cdesc_t *) &a_fortran, a_c, CFI_attribute_other,
                     CFI_type_struct, sizeof(t), 1, extent);
```

8.3.5.6 int CFI_is_contiguous (const CFI_cdesc_t * dv);

Description. Test contiguity of an array.

Formal Parameter.

`dv` shall be the address of a C descriptor describing an array. The `base_addr` member of the C descriptor shall not be a null pointer.

Result Value. CFI_is_contiguous returns 1 if the array described is determined to be contiguous, and 0 otherwise.

NOTE 8.10

A C descriptor for an array describes a contiguous object if it has extent -1 in its final `dim` element or if its attribute member indicates that the array is allocatable.

8.3.5.7 int CFI_section (CFI_cdesc_t * result, const CFI_cdesc_t * source, const CFI_index_t lower_bounds[], const CFI_index_t upper_bounds[], const CFI_index_t strides[]);

Description. Update a C descriptor for an array section for which each element is an element of a given array.

Formal Parameters.

`result` shall be the address of a C descriptor with rank equal to the rank of `source` minus the number of zero strides. The `attribute` member shall have the value CFI_attribute_other or CFI_attribute_pointer. If the value of `result` is the same as either a C formal parameter that corresponds to a Fortran actual argument or a C actual argument that corresponds to a Fortran dummy argument, the `attribute` member shall have the value CFI_attribute_pointer.

`source` shall be the address of a C descriptor that describes an nonallocatable nonpointer array, an allocated allocatable array, or an associated array pointer. The corresponding values of the `elem_len` and `type` members shall be the same in the C descriptors with the addresses `source` and `result`.

`lower_bounds` shall be a null pointer or the address of an array specifying the subscripts of the element in the array described by the C descriptor with the address `source` that is the first element, in Fortran array element order (6.5.3.2 of ISO/IEC 1539-1:2010), of the array section. If it is a null pointer, the subscripts of the first element of the array described by the C descriptor with the address `source` are used; otherwise, the number of elements shall be greater than or equal to `source->rank`.

`upper_bounds` shall be a null pointer or the address of an array specifying the subscripts of the element in the array described by the C descriptor with the address `source` that is the last element, in Fortran array element order (6.5.3.2 of ISO/IEC 1539-1:2010), of the array section. If it is a null pointer, the C descriptor with the address `source` shall not describe an assumed-size array and the subscripts of the last element of the array are used; otherwise, the number of elements shall be greater than or equal to `source->rank`.

`strides` shall be a null pointer or the address of an array specifying the strides of the array section in units of elements of the array described by the C descriptor with the address `source`; if a stride is 0, the section subscript for the dimension is a subscript and the corresponding elements of `lower_bounds` and `upper_bounds` shall be equal. If it is a null pointer, the strides are treated as being all 1; otherwise, the number of elements shall be greater than or equal to `source->rank`.

Successful execution of CFI_section updates the `base_addr` and `dim` members of the C descriptor with the address `result` to describe a section of the array described by the C descriptor with the address `source`.

If an error is detected, the C descriptor with the address `result` is not modified.

Result Value. The function returns an error indicator.

Example 1. If `source` is already the address of a C descriptor for the rank-one Fortran array `A` declared as

```
real A(100)
```

the following code fragment establishes a C descriptor and updates it to describe the array section A(3::5).

```
CFI_index_t lower_bounds[] = {2}, strides[] = {5};
CFI_CDESC_T(1) section;
int ind;
CFI_rank_t rank = 1 ;
ind = CFI_establish ((CFI_cdesc_t *) &section, NULL,
      CFI_attribute_other, CFI_type_float, 0, rank, NULL);
ind = CFI_section ( (CFI_cdesc_t *) &section, source,
      lower_bounds, NULL, strides );
```

Example 2. If `source` is already the address of a C descriptor for the rank-two assumed-shape array `A` declared in Fortran as

```
real A(100,100)
```

the following code fragment establishes a C descriptor and updates it to describe the rank-one array section `A(:,42)`.

```
CFI_index_t lower_bounds[] = {source->dim[0].lower_bound,41},
    upper_bounds[] = {source->dim[0].lower_bound+source->dim[0].extent-1,41},
    strides[] = {1,0};
CFI_CDESC_T(1) section;
int ind;
```

```
    CFI_rank_t rank = 1 ;
    ind = CFI_establish ((CFI_cdesc_t *) &section, NULL,
          CFI_attribute_other, CFI_type_float, 0, rank, NULL);
    ind = CFI_section ( (CFI_cdesc_t *) &section, source,
          lower_bounds, upper_bounds, strides );
```

8.3.5.8 int CFI_select_part (CFI_cdesc_t * result, const CFI_cdesc_t * source, size_t displacement, size_t elem_len);

Description. Update a C descriptor for an array section for which each element is a part of the corresponding element of an array.

Formal Parameters.

`result` shall be the address of a C descriptor. The `type` member of the C descriptor shall specify the type of the array section. The `attribute` member shall have the value CFI_attribute_other or CFI_attribute_pointer. If the address specified by `result` is the value of a C formal parameter that corresponds to a Fortran actual argument or a C actual argument that corresponds to a Fortran dummy argument, the `attribute` member shall have the value CFI_attribute_pointer. The `rank` member of the C descriptor shall have the same value as the `rank` member of the C descriptor at the address specified by `source`.

`source` shall be the address of a C descriptor for an nonallocatable nonpointer array, an allocated allocatable array, or an associated array pointer.

`displacement` is the value in bytes to be added to the base address of the array described by the C descriptor with the address `source` to give the base address of the array section. The resulting base address shall be appropriately aligned (ISO/IEC 9899:1999 3.2) for an object of the specified type. The value of `displacement` shall be between 0 and `source->elem_len - 1` inclusive.

`elem_len` is ignored unless `type` is a character type. If the array section is of Fortran character type, the value of `elem_len` shall be the number of characters in an element of the array section times the `sizeof()` for a scalar of the character type and kind. The value of `elem_len` shall be between 1 and `source->elem_len` inclusive.

Successful execution of CFI_select_part updates the `base_addr`, `dim`, and `elem_len` members of the C descriptor with the address `result` for an array section for which each element is a part of the corresponding element of the array described by the C descriptor with the address `source`. The part may be a component of a structure, a substring, or the real or imaginary part of a complex value.

If an error is detected, the C descriptor with the address `result` is not modified.

Result Value. The function returns an error indicator.

Example. If `source` is already the address of a C descriptor for the Fortran array `a` declared thus:

```
  type,bind(c):: t
    real(C_DOUBLE) :: x
    complex(C_DOUBLE_COMPLEX) :: y
  end type
  type(t) a(100)
```

the following code fragment establishes a C descriptor for the array `a(:)%y`.

```
   typedef struct { double x; double _Complex y;} t;
   CFI_CDESC_T(1) component;
   int ind;
   CFI_cdesc_t * comp_cdesc = (CFI_cdesc_t *)&component;
```

```
CFI_index_t extent[] = {100};

ind = CFI_establish (comp_cdesc, NULL, CFI_attribute_other,
      CFI_type_double_Complex, sizeof(double _Complex), 1, extent);

ind = CFI_select_part (comp_cdesc, source, offsetof(t,y), 0);
```

8.3.5.9 int CFI_setpointer (CFI_cdesc_t * result, CFI_cdesc_t * source, const CFI_index_t lower_bounds[]);

Description. Update a C descriptor for a Fortran pointer to be associated with the whole of a given object or to be disassociated.

Formal Parameters.

`result` shall be the address of a C descriptor for a Fortran pointer. It is updated using information from the `source` and `lower_bounds` arguments.

`source` shall be a null pointer or the address of a C descriptor for a nonallocatable nonpointer data object, an allocated allocatable object, or a data pointer object. If `source` is not a null pointer, the corresponding values of the `elem_len`, `rank`, and `type` members shall be the same in the C descriptors with the addresses `source` and `result`.

`lower_bounds` is ignored if `source` is a null pointer or the rank specified by `source` is zero. Otherwise, the number of elements in the array `lower_bounds` shall be greater than or equal to the rank specified in the `source` C descriptor. The elements provide the lower bounds for each corresponding dimension of the `result` C descriptor. The extents and memory strides are copied from the `source` C descriptor.

Successful execution of CFI_setpointer updates the `base_addr` and `dim` members of the C descriptor with the address `result` with information in the C descriptor with the address `source` and in `lower_bounds`.

If `source` is a null pointer or the address of a C descriptor for a disassociated pointer, the updated C descriptor describes a disassociated pointer. Otherwise, the C descriptor with the address `result` becomes a C descriptor for the object described by the C descriptor with the address `source`, except that the lower bounds are replaced by the values of the `lower_bounds` array if the rank is greater than zero and `lower_bounds` is not a null pointer.

If an error is detected, the C descriptor with the address `result` is not modified.

Result Value. The function returns an error indicator.

Example. If `ptr` is already the address of a C descriptor for an array pointer of rank 1, the following code updates it to a C descriptor for a pointer to the same array with lower bound 0.

```
CFI_index_t lower_bounds[1];
int ind;
lower_bounds[0] = 0;
ind = CFI_setpointer ( ptr, ptr, lower_bounds );
```

8.4 Use of C descriptors

A C descriptor shall not be initialized, updated or copied other than by calling the functions specified here.

If the address of a C descriptor is a formal parameter that corresponds to a Fortran actual argument or a C actual argument that corresponds to a Fortran dummy argument,

- the C descriptor shall not be modified if either the corresponding dummy argument in the Fortran interface has the INTENT(IN) attribute or the C descriptor is for a nonallocatable nonpointer object, and

- the `base_addr` member of the C decriptor shall not be accessed before it is given a value if the corresponding dummy argument in the Fortran interface has the POINTER and INTENT(OUT) attributes.

NOTE 8.11

In this context, modification refers to any change to the location or contents of the C descriptor, including establishment and update. The intent of these restrictions is that C descriptors remain intact at all times they are accessible to an active Fortran procedure, so that the Fortran code is not required to copy them. C programmers should note that doing things with C descriptors that are not possible in Fortran will cause undefined behavior.

8.5 Restrictions on formal parameters

Within a C function, an allocatable object shall be allocated or deallocated only by execution of the CFI_allocate and CFI_deallocate functions. A Fortran pointer can become associated with a target by execution of the CFI_allocate function.

Calling CFI_allocate or CFI_deallocate for a C descriptor changes the allocation status of the Fortran variable it describes and causes the allocation status of any associated allocatable variable to change accordingly (6.7.1.3 of ISO/IEC 1539-1:2010).

If the address of an object is the value of a formal parameter that corresponds to a nonpointer dummy argument in a BIND(C) interface, then

- if the dummy argument has the INTENT(IN) argument, the object shall not be defined or become undefined, and
- if the dummy argument has the INTENT(OUT) attribute, the object shall not be referenced before it is defined.

Some of the functions described in 8.3.5 return an integer value that indicates whether an error condition was detected. A nonzero value is returned if an error condition was detected, and the value zero is returned otherwise. A list of error conditions and macro names for the corresponding error codes is supplied in 8.3.4. A processor is permitted to detect other error conditions. If an invocation of a function defined in 8.3.5 could detect more than one error condition and an error condition is detected, which error condition is detected is processor dependent.

8.6 Restrictions on lifetimes

When a Fortran object is deallocated, execution of its host instance is completed, or its association status becomes undefined, all C descriptors and C pointers to any part of it become undefined, and any further use of them is undefined behavior (ISO/IEC 9899:1999 3.4.3).

A C descriptor whose address is a formal parameter that corresponds to a Fortran dummy argument becomes undefined on return from a call to the function from Fortran. If the dummy argument does not have either the TARGET or ASYNCHRONOUS attribute, all C pointers to any part of the object it describes become undefined on return from the call, and any further use of them is undefined behavior.

If the address of a C descriptor is passed as an actual argument to a Fortran procedure, the lifetime (ISO/IEC 9899:1999 6.2.4) of the C descriptor shall not end before the return from the procedure call. If an object is passed to a Fortran procedure as a nonallocatable, nonpointer dummy argument, its lifetime shall not end before the return from the procedure call.

If the lifetime of a C descriptor for an allocatable object that was established by C ends before the program exits, the object shall be unallocated at that time.

If a Fortran pointer becomes associated with a C object, the association status of the Fortran pointer becomes undefined when the lifetime of the C object ends.

NOTE 8.12

The following illustrates a C descriptor that becomes undefined after a call to a C function.

```
real ary(1000),b
interface
   real function Cfun(array) bind(c, name="Cfun")
      real array(:)
   end function Cfun
end interface
b = Cfun(ary)
```

`Cfun` is a C function. Before `Cfun` is invoked, the processor creates a C descriptor for the array `ary`. On return from `Cfun`, the C descriptor becomes undefined. Because the dummy argument `array` does not have the TARGET or ASYNCHRONOUS attribute, all C pointers whose values were set during execution of `Cfun` to be the address of any part of `ary` become undefined.

8.7 Interoperability of procedures and procedure interfaces

The rules in this subclause replace the contents of paragraphs one and two of subclause 15.3.7 of ISO/IEC 1539-1:2010 entirely.

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

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 (ISO/IEC 9899:1999, 6.2.5, 7.17, and 7.18.1) of the formal parameter,

   (b) the dummy argument is a nonallocatable, nonpointer variable of type CHARACTER with assumed length, and corresponds to a formal parameter of the prototype that is a pointer to CFI_cdesc_t,

   (c) the dummy argument is allocatable, assumed-shape, assumed-rank, or a pointer without the CONTIGUOUS attribute, and corresponds to a formal parameter of the prototype that is a pointer to CFI_cdesc_t, or

   (d) the dummy argument is assumed-type and not assumed-shape or assumed-rank, and corresponds to a formal parameter of the prototype that is a pointer to void.

If a dummy argument in an interoperable interface is of type CHARACTER and is allocatable or a pointer, its character length shall be deferred.

If a dummy argument in an interoperable interface is allocatable, assumed-shape, of assumed character length, assumed-rank, or a data pointer, the corresponding formal parameter is interpreted as the address of a C descriptor for the effective argument in a reference to the procedure. The C descriptor shall describe an object with the same characteristics as the effective argument; the `type` member shall have a value from Table 8.2 that depends on the effective argument as follows:

- if the dynamic type of the effective argument is an interoperable type listed in Table 8.2, the corresponding value for that type;
- if the dynamic type of the effective argument is an intrinsic type with no corresponding type listed in Table 8.2, or a noninteroperable derived type that does not have type parameters, type-bound procedures, final procedures, nor components that have the ALLOCATABLE or POINTER attributes, or correspond to CFI_type_other, one of the processor-dependent nonnegative type specifier values;
- otherwise, CFI_type_other.

In an invocation of an interoperable procedure whose Fortran interface has an assumed-shape or assumed-rank dummy argument with the CONTIGUOUS attribute, the corresponding actual argument may be an array that is not contiguous or the address of a C descriptor for such an array. If the procedure is invoked from Fortran or the procedure is a Fortran procedure, the Fortran processor will handle the difference in contiguity. If the procedure is invoked from C and the procedure is a C procedure, the C code within the procedure shall be prepared to handle the situation of receiving a discontiguous argument.

An absent actual argument in a reference to an interoperable procedure is indicated by a corresponding formal parameter with the value of a null pointer.

9 Required editorial changes to ISO/IEC 1539-1:2010(E)

9.1 General

The following editorial changes, if implemented, would provide the facilities described in foregoing clauses of this Technical Specification. Descriptions of how and where to place the new material are enclosed in braces {}. Edits to different places within the same clause are separated by horizontal lines.

In the edits, except as specified otherwise by the editorial instructions, underwave (underwave) and strike-out (~~strike-out~~) are used to indicate insertion and deletion of text.

9.2 Edits to Introduction

{In paragraph 1 of the Introduction }

After "informally known as Fortran 2008"
insert ", plus the facilities defined in ISO/IEC TS 29113:2012".

{After paragraph 3 of the Introduction, insert new paragraph}

ISO/IEC TS 29113 provides additional facilities with the purpose of improving interoperability with the C programming language:

- assumed-type objects provide more convenient interoperability with C pointers;
- assumed-rank objects provide more convenient interoperability with the C memory model;
- it is now possible for a C function to interoperate with a Fortran procedure that has an allocatable, assumed character length, assumed-shape, optional, or pointer dummy data object.

9.3 Edits to clause 1

{Insert new term definitions before term **1.3.9 attribute**}

1.3.8a
assumed rank
⟨dummy variable⟩ the property of assuming the rank from its effective argument (5.3.8.7, 12.5.2.4)

1.3.8b
assumed type
⟨dummy variable⟩ being declared as TYPE (*) and therefore assuming the type and type parameters from its effective argument (4.3.1)

{Insert new term definition before **1.3.20 character context**}

1.3.19a
C descriptor
C structure of type CFI_cdesc_t defined in the header `ISO_Fortran_binding.h` (15.5)

{Insert new subclause before 1.6.2 Fortran 2003 compatibility}

1.6.1a Fortran 2008 compatibility

This part of ISO/IEC 1539 is an upward compatible extension to the preceding Fortran International Standard, ISO/IEC 1539-1:2010(E). Any standard-conforming Fortran 2008 program remains standard-conforming under this part of ISO/IEC 1539.

9.4 Edits to clause 4

{In 4.3.1.1 Type specifier syntax, insert additional production for R403 *declaration-type-spec* after the one for CLASS (*)}

or TYPE (*)

{In 4.3.1.2 TYPE, edit the first paragraph as follows}

A TYPE type specifier is used to declare entities that are of assumed type, or of an intrinsic or derived type.

{In 4.3.1.2 TYPE, insert new paragraphs at the end of the subclause}

An entity that is declared using the TYPE(*) type specifier has assumed type and is an unlimited polymorphic entity (4.3.1.3). Its dynamic type and type parameters are assumed from its associated effective argument.

C407a An assumed-type entity shall be a dummy variable that does not have the ALLOCATABLE, CODIMENSION, POINTER or VALUE attributes.

C407b An assumed-type variable name shall not appear in a designator or expression except as an actual argument corresponding to a dummy argument that is assumed-type, or as the first argument to any of the intrinsic and intrinsic module functions IS_CONTIGUOUS, LBOUND, PRESENT, RANK, SHAPE, SIZE, UBOUND, and C_LOC.

C407c An assumed-type actual argument that corresponds to an assumed-rank dummy argument shall be assumed-shape or assumed-rank.

9.5 Edits to clause 5

{In 5.3.1 Constraints, replace C516 with}

C516 The ALLOCATABLE or POINTER attribute shall not be specified for a default-initialized dummy argument of a procedure that has a *proc-language-binding-spec*.

{In 5.3.4 ASYNCHRONOUS attribute, edit paragraphs 1 and 2 as follows:}

An entity with the ASYNCHRONOUS attribute is a variable that may be subject to asynchronous input/output or asynchronous communication (15.6.4)

The base object of a variable shall have the ASYNCHRONOUS attribute in a scoping unit if

- the variable appears in an executable statement or specification expression in that scoping unit and
- any statement of the scoping unit is executed while the variable is a pending I/O storage sequence affector (9.6.2.5) or a pending communication affector (15.6.4).

{In 5.3.7 CONTIGUOUS attribute, edit C530 as follows}

C530 An entity with the CONTIGUOUS attribute shall be an array pointer, ~~or~~ an assumed-shape array, or have assumed rank.

{In 5.3.7 CONTIGUOUS attribute, edit paragraph 1 as follows}

The CONTIGUOUS attribute specifies that an assumed-shape array can only be argument associated with a contiguous effective argument, ~~or~~ that an array pointer can only be pointer associated with a contiguous target, or that an assumed-rank object can only be argument associated with a scalar or contiguous effective argument.

{In 5.3.7 CONTIGUOUS attribute, paragraph 2, item (3)}

Change first "array" to "or assumed-rank dummy argument",
change second "array" to "object".

{In 5.3.8.1 General, edit paragraph 1 as follows}

The DIMENSION attribute specifies that an entity has assumed rank or is an array. An assumed-rank entity has the rank and shape of its associated actual argument; otherwise, the~~The~~ rank or rank and shape is specified by its *array-spec*.

{In 5.3.8.1 General, insert additional production for R515 *array-spec*, after *implied-shape-spec-list*}

or *assumed-rank-spec*

{At the end of 5.3.8, immediately before 5.3.9, insert new subclause}

5.3.8.7 Assumed-rank entity

An assumed-rank entity is a dummy variable whose rank is assumed from its effective argument; this rank may be zero. An assumed-rank entity is declared with an *array-spec* that is an *assumed-rank-spec*.

R522a *assumed-rank-spec* **is** ..

C535a An assumed-rank entity shall be a dummy variable that does not have the CODIMENSION or VALUE attribute.

C535b An assumed-rank variable name shall not appear in a designator or expression except as an actual argument corresponding to 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 intrinsic function RANK can be used to inquire about the rank of a data object.

9.6 Edits to clause 6

{In 6.5.4 Simply contiguous array designators, paragraph 2, edit the second bullet item as follows}

- an *object-name* that is not a pointer, not ~~or~~ assumed-shape, and not assumed-rank,

---

{In 6.7.3.2 Deallocation of allocatable variables, append to paragraph 6}

If a Fortran procedure that has an INTENT (OUT) allocatable dummy argument is invoked by a C function and the corresponding argument in the C function call is a C descriptor that describes an allocated allocatable variable, the variable is deallocated on entry to the Fortran procedure. When a C function is invoked from a Fortran procedure via an interface with an INTENT (OUT) allocatable dummy argument and the corresponding actual argument in the reference of the C function is an allocated allocatable variable, the variable is deallocated on invocation (before execution of the C function begins).

9.7 Edits to clause 12

{In 12.3.2.2, edit paragraph 1 as follows}

The characteristics of a dummy data object are its type, its type parameters (if any), its shape (unless it is assumed-rank), its corank, its codimensions, its intent (5.3.10, 5.4.10), whether it is optional (5.3.12, 5.4.10), whether it is allocatable (5.3.3), whether it has the ASYNCHRONOUS (5.3.4), CONTIGUOUS (5.3.7), VALUE (5.3.18), or VOLATILE (5.3.19) attributes, whether it is polymorphic, and whether it is a pointer (5.3.14, 5.4.12) or a target (5.3.17, 5.4.15). If a type parameter of an object or a bound of an array is not a constant expression, the exact dependence on the entities in the expression is a characteristic. If a rank, shape, size, type, or type parameter is assumed or deferred, it is a characteristic.

{In 12.4.2.2 Explicit interface, after item (2)(c) insert new item}

(c2) has assumed rank,

{Replace paragraph 2 of 12.4.3.4.5 with}

A dummy argument is type, kind, and rank compatible, or TKR compatible, with another dummy argument if the first is type compatible with the second, the kind type parameters of the first have the same values as the corresponding kind type parameters of the second, and both have the same rank or either is assumed-rank.

{In 12.5.2.4 Ordinary dummy variables, append to paragraph 2}

If the actual argument is of a derived type that has type parameters, type-bound procedures, or final subroutines, the dummy argument shall not be of assumed type.

{In 12.5.2.4 Ordinary dummy variables, paragraphs 3 and 4}

Change "not assumed shape" to "explicit-shape or assumed-size" (twice).

{In 12.5.2.4 Ordinary dummy variables, paragraph 9}

After "dummy argument is a scalar"
Change "or" to ", has assumed rank, or is".

{In 12.5.2.4 Ordinary dummy variables, paragraph 10}

After "the CONTIGUOUS attribute" insert ", an assumed-rank entity with the CONTIGUOUS attribute that is associated with an array actual argument".

{In 12.5.2.4 Ordinary dummy variables, insert new paragraph after paragraph 14}

An actual argument of any rank may correspond to an assumed-rank dummy argument. The rank and shape of the dummy argument are the rank and shape of the corresponding actual argument. If the rank is nonzero, the lower and upper bounds of the dummy argument are those that would be given by the intrinsic functions LBOUND and UBOUND respectively if applied to the actual argument, except that when the actual argument is assumed size, the upper bound of the last dimension of the dummy argument is 2 less than the lower bound of that dimension.

{In 12.5.2.4 Ordinary dummy variables, paragraph 18, constraint C1239}

Change "or an assumed-shape ... attribute" to ", an assumed-shape array without the CONTIGUOUS attribute, or an assumed-rank array without the CONTIGUOUS attribute".

{In 12.5.2.4 Ordinary dummy variables, paragraph 18, constraint C1240}

Change "or an assumed-shape ... attribute" to ", an assumed-shape array without the CONTIGUOUS attribute, or an assumed-rank array without the CONTIGUOUS attribute".

{In 12.5.2.13 Restrictions on entities associated with dummy arguments, paragraph 1, item (3) (b)}

Change "or an assumed-shape array without the CONTIGUOUS attribute" to ", an assumed-rank entity that is associated with a scalar actual argument, an assumed-rank entity without the CONTIGUOUS attribute, or an assumed-shape array without the CONTIGUOUS attribute".

{In 12.5.2.13 Restrictions on entities associated with dummy arguments, paragraph 1, item (4) (b)}

Change "or an assumed-shape array without the CONTIGUOUS attribute" to ", an assumed-rank entity that is associated with a scalar actual argument, an assumed-rank entity without the CONTIGUOUS attribute, or an assumed-shape array without the CONTIGUOUS attribute".

{In 12.6.2.2 Function subprogram, edit C1255 as follows}

C1255 (R1229) If *proc-language-binding-spec* is specified for a procedure, each of the procedure's dummy arguments shall be an ~~nonoptional~~ interoperable variable (15.3.5, 15.3.6) that does not have both the OPTIONAL and VALUE attributes, or an ~~nonoptional~~ interoperable procedure (15.3.7). If *proc-language-binding-spec* is specified for a function, the function result shall be an interoperable scalar variable.

9.8 Edits to clause 13

{In 13.5 Standard generic intrinsic procedures, Table 13.1, LBOUND and UBOUND intrinsic functions}

Delete " of an array" (twice).

{In 13.5 Standard generic intrinsic procedures, Table 13.1}

Insert new entry into the table, alphabetically

RANK	(A)	I	Rank of a data object.

{In 13.7.86, IS_CONTIGUOUS, edit paragraph 3 as follows}

Argument. ARRAY may be of any type. It shall be an array or an assumed-rank object. If it is a pointer it shall be associated.

{In 13.7.86, IS_CONTIGUOUS, edit paragraph 5 as follows}

Result Value. The result has the value true if ARRAY has rank zero or is contiguous, and false otherwise.

{In 13.7.90 LBOUND, edit paragraph 1 as follows}

Description. Lower bound(s) ~~of an array~~.

{In 13.7.90 LBOUND, edit paragraph 3, ARRAY argument, as follows}

ARRAY shall be an array or assumed-rank object of any type. It shall not be an unallocated allocatable variable or a pointer that is not associated.

{In 13.7.90 LBOUND, insert note after paragraph 3}

"NOTE 13.14a
If ARRAY is an assumed-rank object of rank zero, DIM cannot be present."

{In 13.7.93 LEN, paragraph 3}

Change "a type character scalar or array"
to "of type character".

{Immediately before subclause 13.8.138 REAL, insert new subclause}

13.7.137a RANK (A)

Description. Rank of a data object.

Class. Inquiry function.

Argument. A shall be a data object of any type.

Result Characteristics. Default integer scalar.

Result Value. The result is the rank of A.

Example. If X is declared as REAL X (:, :, :), RANK(X) has the value 3.

{In 13.7.149 SHAPE, replace paragraph 5 with}

Result Value. The result has a value equal to [(SIZE(SOURCE, *i*, KIND), *i*=1, RANK(SOURCE))].

{In 13.7.156 SIZE, edit paragraph 3, argument ARRAY, as follows}

ARRAY	shall be an array or assumed-rank object of any type. It shall not be an unallocated allocatable variable or a pointer that is not associated. If ARRAY is an assumed-size array, DIM shall be present with a value less than the rank of ARRAY.

{In 13.7.156 SIZE, insert note after paragraph 3}

"NOTE 13.21a
If ARRAY is an assumed-rank object of rank zero, DIM cannot be present."

{In 13.7.156 SIZE, replace paragraph 5 with}

Result Value. If ARRAY is an assumed-rank object associated with an assumed-size array and DIM is present with a value equal to the rank of ARRAY, the result is -1; otherwise, if DIM is present, the result has a value equal to the extent of dimension DIM of ARRAY. If DIM is not present, the result has a value equal to PRODUCT([(SIZE(ARRAY, *i*, KIND), *i*=1, RANK(ARRAY))]).

{In 13.7.160 STORAGE_SIZE, paragraph 3}

Change "a scalar or array of any type"
to "a data object of any type".

{In 13.7.171 UBOUND, paragraph 1}

Delete " of an array".

{In 13.7.171 UBOUND, paragraph 3, ARRAY argument}

After "shall be an array"
insert "or assumed-rank object".

{In 13.7.171 UBOUND, insert note after paragraph 3}

"NOTE 13.24a
If ARRAY is an assumed-rank object of rank zero, DIM cannot be present."

{In 13.7.171 UBOUND, edit paragraph 5 as follows}

Result Value.

Case (i): For an array section or for an array expression, other than a whole array, UBOUND (ARRAY, DIM) has a value equal to the number of elements in the given dimension~~; otherwise,~~.

Case (ii): For an assumed-rank object associated with an assumed-size array, UBOUND(ARRAY, n, KIND) where n is the rank of ARRAY has a value equal to LBOUND(ARRAY, n, KIND) $-$ 2.

Case (iii): Otherwise, UBOUND(ARRAY, DIM) has a value equal to the upper bound for subscript DIM of ARRAY if dimension DIM of ARRAY does not have size zero and has the value zero if dimension DIM has size zero.

Case (iv): UBOUND (ARRAY) has a value whose i^{th} element is equal to UBOUND (ARRAY, i), for $i = 1, 2, \ldots, n$, where n is the rank of ARRAY.

9.9 Edits to clause 15

{In 15.1 General, at the end of the subclause, insert new paragraph}

The header `ISO_Fortran_binding.h` provides definitions and prototypes to enable a C function to interoperate with a Fortran procedure with an allocatable, assumed character length, assumed-shape, assumed-rank, or pointer dummy data object.

{In 15.2.3.3 C_F_POINTER (CPTR, FPTR [,SHAPE]), paragraph 3, append a new paragraph to the description of FPTR:}

"If the value of CPTR is the C address of a storage sequence, FPTR becomes associated with that storage sequence. If FPTR is an array, its shape is specified by SHAPE and each lower bound is 1. The storage sequence shall be large enough to contain the target object described by FPTR, shall not be in use by another Fortran entity, and shall satisfy any other processor-dependent requirements for association."

{At the end of 15.2.3.3 C_F_POINTER (CPTR, FPTR [,SHAPE], insert new note}

"NOTE 15.xx
In the case of associating FPTR with a storage sequence, there might be processor-dependent requirements such as alignment of the memory address or placement in memory."

{In 15.2.3.5 C_FUNLOC(X), paragraph 3}

Delete "that is interoperable" and change "associated with an interoperable procedure" to "associated with a procedure".

{In 15.2.3.6 C_LOC(X), paragraph 3}

Delete "scalar,".

{In 15.3.2 Interoperability of intrinsic types, Table 15.2, add a new Named constant / C type pair in the Fortran type = INTEGER block, following C_INTPTR_T | intptr_t, as follows}

C_PTRDIFF_T | ptrdiff_t

{In 15.3.7 Interoperability of procedures and procedure interfaces, paragraph 2, edit item (5) as follows}

(5) any dummy argument without the VALUE attribute corresponds to a formal parameter of the prototype that is of pointer type, and either

(a) the dummy argument is interoperable with an entity of the referenced type (ISO/IEC 9899:1999, 6.25, 7.17, and 7.18.1) of the formal parameter,

(b) the dummy argument is a nonallocatable, nonpointer variable of type CHARACTER with assumed length, and corresponds to a formal parameter of the prototype that is a pointer to CFI_desc_t,

(c) the dummy argument is allocatable, assumed-shape, assumed-rank, or a pointer without the CONTIGUOUS attribute, and corresponds to a formal parameter of the prototype that is a pointer to CFI_cdesc_t, or

(d) the dummy argument is assumed-type and not allocatable, assumed-shape, assumed-rank, or a pointer, and corresponds to a formal parameter of the prototype that is a pointer to void,

(5a) each allocatable or pointer dummy argument of type CHARACTER has deferred character length, and,

---

{In 15.3.7 Interoperability of procedures and procedure interfaces, insert new paragraphs at the end of the subclause}

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

In an invocation of an interoperable procedure whose Fortran interface has an assumed-shape or assumed-rank dummy argument with the CONTIGUOUS attribute, the corresponding actual argument may be an array that is not contiguous or the address of a C descriptor for such an array. If the procedure is invoked from Fortran or the procedure is a Fortran procedure, the Fortran processor will handle the difference in contiguity. If the procedure is invoked from C and the procedure is a C procedure, the C code within the procedure shall be prepared to handle the situation of receiving a discontiguous argument.

An absent actual argument in a reference to an interoperable procedure is indicated by a corresponding formal parameter with the value of a null pointer.

{Before subclause 15.5}

Insert subclauses 8.3 to 8.7 of this Technical Specification as subclauses 15.5 to 15.9, including subclauses 8.3.1 to 8.3.5 as subclauses 15.5.1 to 15.5.5, with the existing 15.5 to be renumbered 15.10 and its subclauses to be renumbered accordingly.

{In 15.5.1 Definition and reference of interoperable procedures, paragraph 4}

Append the sentence "A C function shall not invoke a Fortran procedure that is not interoperable.".

{At the end of subclause 15.5}

Insert subclause 5.4.2 of this Technical Specification at the very end of clause 15 where it will become 15.10.4.

9.10 Edits to clause 16

{In 16.5.2.4 Events that cause pointers to become disassociated, insert a new list item following list item (3)}

"(3a) the target of the pointer is a C object and the lifetime of the C object ends,"

9.11 Edits to annex A

{At the end of A.2 Processor dependencies, replace the final full stop with a semicolon and add new items as follows}

- the value of CFI_MAX_RANK in the file CFI_Fortran_binding.h;
- the value of CFI_VERSION in the file CFI_Fortran_binding.h;
- which error condition is detected if more than one error condition is detected for an invocation of one of the functions specified in the file CFI_Fortran_binding.h;

- the values of the attribute specifier macros defined in the file CFI_Fortran_binding.h;
- the values of the type specifier macros defined in the file CFI_Fortran_binding.h;
- which additional type specifier values are defined in the file CFI_Fortran_binding.h;
- the values of the error code macros, except for CFI_SUCCESS, defined in the file CFI_Fortran_binding.h;
- the base address of a zero-sized array;
- the requirements on the storage sequence to be associated with the pointer FPTR by the C_F_POINTER subroutine;
- whether a procedure defined by means other than Fortran is an asynchronous communication initiation or completion procedure.

9.12 Edits to annex C

{In C.11 Clause 15 notes, at the end of the subclause}

Insert subclauses A.1.1 to A.1.4 as subclauses C.11.6 to C.11.9.

Insert subclause A.2.1 as C.11.10 with the revised title "Processing assumed-shape arrays in C".

Insert subclauses A.2.2 to A.2.6 as subclauses C.11.11 to C.11.15.

Annex A

(Informative)

Extended notes

A.1 Clause 5 notes

A.1.1 Using assumed type in the context of interoperation with C

The mechanism for handling unlimited polymorphic entities whose dynamic type is interoperable with C is designed to handle the following two situations:

(1) A formal parameter that is a C pointer to void. This is an address, and no further information about the entity is provided. The formal parameter corresponds to a dummy argument that is a nonallocatable nonpointer scalar or is an array of assumed size.

(2) A formal parameter that is the address of a C descriptor. Additional information on the status, type, size, and shape is implicitly provided. The formal parameter corresponds to a dummy argument that is of assumed shape or assumed rank.

In the first situation, it is the programmer's responsibility to explicitly provide any information needed on the status, type, size, and shape of the entity.

The examples A.1.2 and A.1.3 illustrate some uses of assumed-type entities.

A.1.2 Mapping of interfaces with void * C parameters to Fortran

A C interface for message passing or I/O functionality could be provided in the form

```
int EXAMPLE_send(const void *buffer, size_t buffer_size, const HANDLE_t *handle);
```

where the `buffer_size` argument is given in units of bytes, and the `handle` argument (which is of a type aliased to `int`) provides information about the target the buffer is to be transferred to. In this example, type resolution is not required.

The first method provides a thin binding; a call to `EXAMPLE_send` from Fortran directly invokes the C function.

```
interface
  integer(c_int) function EXAMPLE_send(buffer, buffer_size, handle)  &
                  bind(c,name="EXAMPLE_send")
    use,intrinsic :: iso_c_binding
    type(*), dimension(*), intent(in) :: buffer
    integer(c_size_t), value :: buffer_size
    integer(c_int), intent(in) :: handle
  end function EXAMPLE_send
end interface
```

It is assumed that this interface is declared in the specification part of a module `mod_EXAMPLE_old`. Example invocations from Fortran then are

```
use, intrinsic :: iso_c_binding
use mod_EXAMPLE_old
```

```
real(c_float) :: x(100)
integer(c_int) :: y(10,10)
real(c_double) :: z
integer(c_int) :: status, handle
:
! assign values to x, y, z and initialize handle
:
! send values in x, y, and z using EXAMPLE_send:
status = EXAMPLE_send(x, c_sizeof(x), handle)
status = EXAMPLE_send(y, c_sizeof(y), handle)
status = EXAMPLE_send((/ z /), c_sizeof(z), handle)
```

In these invocations, x and y are passed by address, and for y the sequence association rules (12.5.2.11 of ISO/IEC 1539-1:2010) allow this. For z, it is necessary to explicitly create an array expression.

```
status = EXAMPLE_send(y, c_sizeof(y(:,1)), handle)
```

passes the first column of y (again by address).

```
status = EXAMPLE_send(y(1,5), c_sizeof(y(:,5)), handle)
```

passes the fifth column of y using the sequence association rules.

The second method provides a Fortran interface which is easier to use, but requires writing a separate C wrapper routine; this is commonly called a "fat binding". In this implementation, a C descriptor is created because the buffer is declared with assumed rank in the Fortran interface; the use of an optional argument is also demonstrated.

```
interface
  subroutine example_send(buffer, handle, status) &
            BIND(C, name="EXAMPLE_send_fortran")
    use,intrinsic :: iso_c_binding
    type(*), dimension(..), contiguous, intent(in) :: buffer
    integer(c_int), intent(in) :: handle
    integer(c_int), intent(out), optional :: status
  end subroutine example_send
end interface
```

It is assumed that this interface is declared in the specification part of a module mod_EXAMPLE_new. Example invocations from Fortran then are

```
use, intrinsic :: iso_c_binding
use mod_EXAMPLE_new

type, bind(c) :: my_derived
  integer(c_int) :: len_used
  real(c_float) ::  stuff(100)
end type
type(my_derived) :: w(3)
real(c_float) :: x(100)
integer(c_int) :: y(10,10)
real(c_double) :: z
integer(c_int) :: status, handle
:
! assign values to w, x, y, z and initialize handle
:
```

```
! send values in w, x, y, and z using EXAMPLE_send
call EXAMPLE_send(w, handle, status)
call EXAMPLE_send(x, handle)
call EXAMPLE_send(y, handle)
call EXAMPLE_send(z, handle)

call EXAMPLE_send(y(:,5), handle) ! fifth column of y
call EXAMPLE_send(y(1,5), handle) ! scalar y(1,5) passed by descriptor
```

However, the following call from Fortran is not allowed

```
type(*) :: d(*) ! is a dummy argument
:
call EXAMPLE_send(d(1:4), handle, status)
```

The wrapper routine implemented in C reads

```
#include "ISO_Fortran_binding.h"

void EXAMPLE_send_fortran(const CFI_cdesc_t *buffer,
                          const HANDLE_t *handle, int *status) {
  int status_local;
  size_t buffer_size;
  int i;

  buffer_size = buffer->elem_len;
  for (i=0; i<buffer->rank; i++) {
    buffer_size *= buffer->dim[i].extent;
  }
  status_local = EXAMPLE_send(buffer->base_addr,buffer_size, handle);
  if (status != NULL) *status = status_local;
}
```

A.1.3 Using assumed-type variables in Fortran

An assumed-type dummy argument in a Fortran procedure can be used as an actual argument corresponding to an assumed-type dummy in a call to another procedure. In the following example, the Fortran subroutine `SIMPLE_Send` serves as a wrapper to hide complications associated with calls to a C function named `ACTUAL_Send`. Module `comm_info` contains node and address information for the current data transfer operations.

```
subroutine SIMPLE_Send (buffer, nbytes)
  use comm_info, only: my_node, r_node, r_addr
  use,intrinsic :: iso_c_binding
  implicit none

  type(*),dimension(*),intent(in) :: buffer
  integer                         :: nbytes
  integer                         :: ierr

  interface
     subroutine ACTUAL_Send (buffer, nbytes, node, addr, ierr) &
        bind(C, name="ACTUAL_Send")
        import :: C_size_t, C_int, C_intptr_t
        type(*),dimension(*),intent(in) :: buffer
        integer(C_size_t),value         :: nbytes
```

```
      integer(C_int),value            :: node
      integer(C_intptr_t),value       :: addr
      integer(C_int),intent(out)      :: ierr
    end subroutine ACTUAL_Send
  end interface

  call ACTUAL_Send (buffer, int(nbytes, C_size_t), r_node, r_addr, ierr)

  if (ierr /= 0) then
     print *, "Error sending from node", my_node, "to node",r_node
     print *, "Program Aborting"  ! Or call a recovery procedure
     error stop                   ! Omit in the recovery case
  end if
end subroutine SIMPLE_Send
```

A.1.4 Simplifying interfaces for arbitrary rank procedures

Example of assumed-rank usage in Fortran

There are situations where an assumed-rank dummy argument can be useful in Fortran, although a Fortran procedure cannot itself access its value. For example, the IEEE inquiry functions 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.

The specific procedures for the IEEE_SUPPORT_DIVIDE function might be implemented in Fortran as follows:

```
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_noarg

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 8 notes

A.2.1 Dummy arguments of any type and rank

The example shown below calculates the product of individual elements of arrays A and B and returns the result in array C. The Fortran interface of `elemental_mult` will accept arguments of any type and rank. However, the

C function will return an error code if any argument is not a two-dimensional `int` array. Note that the arguments are permitted to be array sections, so the C function does not assume that any argument is contiguous.

The Fortran interface is:

```
interface
   function elemental_mult(A, B, C) bind(C,name="elemental_mult_c"), result(err)
      use,intrinsic  :: iso_c_binding
      integer(c_int) :: err
      type(*), dimension(..) :: A, B, C
   end function elemental_mult
end interface
```

The definition of the C function is:

```
#include "ISO_Fortran_binding.h"

int elemental_mult_c(CFI_cdesc_t * a_desc,
                     CFI_cdesc_t * b_desc, CFI_cdesc_t * c_desc) {
  size_t i, j, ni, nj;

  int err = 1;  /* this error code represents all errors */

  char * a_col = (char*) a_desc->base_addr;
  char * b_col = (char*) b_desc->base_addr;
  char * c_col = (char*) c_desc->base_addr;
  char *a_elt, *b_elt, *c_elt;

  /* only support integers */
  if (a_desc->type != CFI_type_int || b_desc->type != CFI_type_int ||
      c_desc->type != CFI_type_int) {
     return err;
  }

  /* only support two dimensions */
  if (a_desc->rank != 2 || b_desc->rank != 2 || c_desc->rank != 2) {
     return err;
  }

  ni = a_desc->dim[0].extent;
  nj = a_desc->dim[1].extent;

  /* ensure the shapes conform */
  if (ni != b_desc->dim[0].extent || ni != c_desc->dim[0].extent) return err;
  if (nj != b_desc->dim[1].extent || nj != c_desc->dim[1].extent) return err;

  /* multiply the elements of the two arrays */
  for (j = 0; j < nj; j++) {
    a_elt = a_col;
    b_elt = b_col;
    c_elt = c_col;
    for (i = 0; i < ni; i++) {
      *(int*)a_elt = *(int*)b_elt * *(int*)c_elt;
```

```
        a_elt += a_desc->dim[0].sm;
        b_elt += b_desc->dim[0].sm;
        c_elt += c_desc->dim[0].sm;
      }
      a_col += a_desc->dim[1].sm;
      b_col += b_desc->dim[1].sm;
      c_col += c_desc->dim[1].sm;
    }
    return 0;
  }
```

A.2.2 Creating a contiguous copy of an array

A C function might need to create a contiguous copy of an array section, such as when the section is an actual argument corresponding to a dummy argument with the CONTIGUOUS attribute. The following example provides functions that can be used to copy an array described by a CFI_cdesc_t descriptor to a contiguous buffer. The input array need not be contiguous.

The C functions are:

```
#include "ISO_Fortran_binding.h"
/* other necessary includes omitted */

/*
 * Returns the number of elements in the object described by desc.
 * If it is an array, it need not be contiguous.
 * (The number of elements could be zero).
 */
size_t numElements(const CFI_cdesc_t * desc) {
   CFI_rank_t r;
   size_t num = 1;

   for (r = 0; r < desc->rank; r++) {
      num *= desc->dim[r].extent;
   }
   return num;
}

/*
 * Auxiliary recursive function to copy an array of a given rank.
 * Recursion is useful because an array of rank n is composed of an
 * ordered set of arrays of rank n-1.
 */
static void * _copyToContiguous (const CFI_cdesc_t * vald,
               void * output, const void * input, CFI_rank_t rank) {
   CFI_index_t e;

   if (rank == 0) {
      /* copy scalar element */
      memcpy (output, input, vald->elem_len);
      output = (void *)((char *)output + vald->elem_len);
   }
   else {
      for (e = 0; e < vald->dim[rank-1].extent; e++) {
         /* recurse on subarrays of lesser rank */
         output = _copyToContiguous (vald, output, input, rank-1);
         input = (void *) ((char *)input + vald->dim[rank].sm);
      }
   }
   return output;
}

/*
 * General routine to copy the elements in the array described by vald
 * to buffer, as done by sequence association.  The array itself may
 * be non-contiguous.  This is not the most efficient approach.
 */
void copyToContiguous (void * buffer, const CFI_cdesc_t * vald) {
   _copyToContiguous (vald, buffer, vald->base_addr, vald->rank);
}
```

A.2.3 Changing the attributes of an array

A C programmer might want to call more than one Fortran procedure and the attributes of an array involved might differ between the procedures. In this case, it is necessary to set up more than one C descriptor for the array. For example, this code fragment initializes the first C descriptor for an allocatable entity of rank 2, calls a procedure that allocates the array described by the first C descriptor, constructs the second C descriptor by invoking CFI_section with the value CFI_attribute_other for the `attribute` parameter, then calls a procedure that expects an assumed-shape array.

```
  CFI_CDESC_T(2) loc_alloc, loc_assum;
  CFI_cdesc_t * desc_alloc = (CFI_cdesc_t *)&loc_alloc,
              * desc_assum = (CFI_cdesc_t *)&loc_assum;
  CFI_index_t extents[2];
  CFI_rank_t rank = 2;
  int flag;

  flag = CFI_establish(desc_alloc,
                       NULL,
                       CFI_attribute_allocatable,
                       CFI_type_double,
                       sizeof(double),
                       rank,
                       NULL);

  Fortran_factor (desc_alloc, ...); /* Allocates array described by desc_alloc */

  /* Extract extents from descriptor */
  extents[0] = desc_alloc->dim[0].extent;
  extents[1] = desc_alloc->dim[1].extent;

  flag = CFI_establish(desc_assum,
                       desc_alloc->base_addr,
                       CFI_attribute_other,
                       CFI_type_double,
                       sizeof(double),
                       rank,
                       extents);
  Fortran_solve (desc_assum, ...); /* Uses array allocated in Fortran_factor */
```

After invocation of the second CFI_establish, the lower bounds stored in the `dim` member of `desc_assum` will have the value 0 even if the corresponding entries in `desc_alloc` have different values.

A.2.4 Creating an array section in C using CFI_section

Given the Fortran subprogram

```
subroutine set_all(int_array, val) bind(c)
  integer(c_int) :: int_array(:)
  integer(c_int), value :: val
  int_array = val
end subroutine
```

that sets all the elements of an array and the Fortran interface

```
interface
  subroutine set_odd(int_array, val) bind(c)
    use, intrinsic :: iso_c_binding, only : c_int
    integer(c_int) :: int_array(:)
    integer(c_int), value :: val
  end subroutine
end interface
```

for a C function that sets every second array element, beginning with the first one, the implementation in C reads

```
#include "ISO_Fortran_binding.h"

void set_odd(CFI_cdesc_t *int_array, int val) {
   CFI_index_t lower_bound[1], upper_bound[1], stride[1];
   CFI_CDESC_T(1) array;
   int status;
   /* Create a new descriptor which will contain the section */
    status = CFI_establish( (CFI_cdesc_t *) &array,
                             NULL,
                             CFI_attribute_other,
                             int_array->type,
                             int_array->elem_len,
                             /* rank */ 1,
                             /* extents is ignored */ NULL);

    lower_bound[0] = int_array->dim[0].lower_bound;
    upper_bound[0] = lower_bound[0] + (int_array->dim[0].extent - 1);
    stride[0] = 2;

    status = CFI_section( (CFI_cdesc_t *) &array,
                          (CFI_cdesc_t *) &int_array,
                          lower_bound,
                          upper_bound,
                          stride);

   set_all( (CFI_cdesc_t *) &array, val);

   /* here one could make use of int_array and access all its data */
}
```

Let invocation of `set_odd()` from a Fortran program be done as follows:

```
integer(c_int) :: d(5)
d = (/ 1, 2, 3, 4, 5 /)
call set_odd(d, -1)
write(*, *) d
```

Then, the program will print

```
   -1    2   -1    4   -1
```

During execution of the subprogram `set_all()`, its dummy object `int_array` would appear to be an array of size 3 with lower bound 1 and upper bound 3.

It is also possible to invoke `set_odd()` from C. However, it is the C programmer's responsibility to make sure that all members of the C descriptor have the correct value on entry to the function. Inserting additional checking into the function's implementation could alleviate this problem.

```
/* necessary includes omitted */
#define ARRAY_SIZE 5

CFI_CDESC_T(1) d;
CFI_index_t extent[1];
CFI_index_t subscripts[1];
void *base;
int i, status;

base = malloc(ARRAY_SIZE*sizeof(int));
extent[0] = ARRAY_SIZE;
status = CFI_establish( (CFI_cdesc_t *) &d,
                         base,
                         CFI_attribute_other,
                         CFI_type_int,
                         /* element length is ignored */ 0,
                         /* rank */ 1,
                         extent);

set_odd( (CFI_cdesc_t *) &d, -1);

for (i=0; i<ARRAY_SIZE; i++) {
  subscripts[1] = i;
  printf("   %d",*((int *)CFI_address( (CFI_cdesc_t *) &d, subscripts)));
}
printf("\n");
free(base);
```

This C program will print (apart from formatting) the same output as the Fortran program above. It also demonstrates how an assumed shape entity is dynamically generated within C.

A.2.5 Use of CFI_setpointer

The following C function modifies a pointer to an integer variable to become associated with a global variable defined inside C:

```
#include "ISO_Fortran_binding.h"

int y = 2;

void change_target(CFI_cdesc_t *ip) {
   CFI_CDESC_T(0) yp;
   int status;
   /* make local yp point at y */
   status = CFI_establish( (CFI_cdesc_t *) &yp,
                           &y,
                           CFI_attribute_pointer,
                           CFI_type_int,
                           /* elem_len is ignored */ sizeof(int),
                           /* rank */ 0,
                           /* extents are ignored */ NULL);
   /* Pointer association of ip with yp */
   status = CFI_setpointer(ip, (CFI_cdesc_t *) &yp, NULL);
   if (status != CFI_SUCCESS) {
   /* handle run time error */
   }
}
```

The restrictions on the use of CFI_establish prohibit direct modification of the incoming pointer entity `ip` by invoking that function on it.

The following Fortran code

```
use, intrinsic :: iso_c_binding

interface
  subroutine change_target(ip) bind(c)
    import :: c_int
    integer(c_int), pointer :: ip
  end subroutine
end interface

integer(c_int), target :: it = 1
integer(c_int), pointer :: it_ptr

it_ptr => it
write(*,*) it_ptr
call change_target(it_ptr)
write(*,*) it_ptr
```

will then print

```
1
2
```

A.2.6 Mapping of MPI interfaces to Fortran

The Message Passing Interface (MPI) specifies procedures for exchanging data between MPI processes. This example shows the usage of `MPI_Send` and is similar to the second variant of `EXAMPLE_Send` in subclause A.1.2. It also shows the usage of assumed-length character dummy arguments as well as optional dummy arguments.

MPI_Send has the C prototype,

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm);
```

where `MPI_Datatype` and `MPI_Comm` are opaque handles. Most MPI C functions return an error code, which in Fortran is the last dummy argument to the corresponding subroutine and can be made optional. Thus, the use of a Fortran subroutine requires a wrapper function, declared as

```
void MPI_Send_f(CFI_cdesc_t *buf, int count, MPI_Datatype_f datatype,
                int dest, int tag, MPI_Datatype_f comm, int *ierror);
```

where it is assumed that *in C* there is a conversion between the C handles of type `MPI_Datatype` and `MPI_Comm` and their respective Fortran handles of type `MPI_Datatype_f` and `MPI_Comm_f`. Conversion of the `CFI_cdesc_t *buf` argument to a contiguous `void *` buffer is also done in the wrapper function.

Similarly, the wrapper function for `MPI_Comm_set_name` could have the C prototype,

```
void MPI_Comm_set_name_f(MPI_Comm comm, CFI_cdesc_t *comm_name,
                         int *ierror);
```

The Fortran handle types and interfaces are defined in the module `MPI_f08`. For example,

```
module MPI_f08
...
type, bind(C) :: MPI_Comm
   integer(c_int) :: MPI_VAL
end type MPI_Comm

interface
  subroutine MPI_Send(buf,count,datatype,dest,tag,comm,ierror) &
    bind(C, name="MPI_Send_f")
    use, intrinsic :: iso_c_binding
    import :: MPI_Datatype, MPI_Comm
    type(*), dimension(..), intent(in) :: buf
    integer(c_int), value, intent(in) :: count, dest, tag
    type(MPI_Datatype), intent(in) :: datatype
    type(MPI_Comm), intent(in) :: comm
    integer(c_int), optional, intent(out) :: ierror
  end subroutine MPI_Send
end interface

interface
  subroutine MPI_Comm_set_name(comm,comm_name,ierror) &
    bind(C, name="MPI_Comm_set_name_f")
    use, intrinsic :: iso_c_binding
    import :: MPI_Comm
    type(MPI_Comm), intent(in) :: comm
    character(kind=c_char, len=*), intent(in) :: comm_name
    integer(c_int), optional, intent(out) :: ierror
  end subroutine MPI_Comm_set_name
end interface
...
end module MPI_f08
```

Example invocations from Fortran are

```
use, intrinsic :: iso_c_binding
use :: MPI_f08

type(MPI_Comm) :: comm
real :: x(100)
integer :: y(10,10)
real(kind(1.0d0)) :: z
integer :: dest, tag, ierror
...
! assign values to x, y, z and initialize MPI variables
...

! set the name of the communicator
call MPI_Comm_set_name(comm, "Communicator Name", ierror)

! send values in x, y, and z
call MPI_Send(x, 100, MPI_REAL, dest, tag, comm, ierror)
call MPI_Send(y(3,:), 10, MPI_INTEGER, dest, tag, comm)
call MPI_Send(z, 1, MPI_DOUBLE_PRECISION, dest, tag, comm)
```

The first example sends the entire array x and includes the optional error argument return value. The second example sends a noncontiguous subarray (the third row of y) and the third example sends a scalar z. Note the differences between the calls in this example and those in A.1.2.