TR 29113 WORKING DRAFT

10-251

16th October 2010 12:38

This is an internal working document of J3.

Contents

List of Tables

Foreword

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

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

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

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

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

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

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

Introduction

Technical Report on Further Interoperability of Fortran with C

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

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

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

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

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

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

7 It also contains the following nonnormative material:

Technical Report — Further Interoperability of Fortran with C —

1 Overview

1.1 Scope

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

1.2 Normative references

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

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

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

1.3 Terms and definitions

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

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

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

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

NOTE 1.1

C descriptors are used by the processor to describe an object that is assumed-shape, assumed-rank, allocatable, or a data pointer.

1.4 Compatibility

1.4.1 New intrinsic procedures

1 ISO/IEC TR 29113 defines 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 ISO/IEC TR 29113 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.

1.4.2 Fortran 2008 compatibility

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

2 Type specifiers and attributes

2.1 Assumed-type objects

1 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 (*)

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

C407x1 An assumed-type entity shall be a dummy variable that does not have the CODIMENSION or VALUE attribute.

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

4 An assumed-type object is unlimited polymorphic.

Unresolved Technical Issue TR1

Note that "unlimited polymorphic" just means that its dynamic type is not limited. It does not mean that it is CLASS(*), or can be used everywhere that CLASS(*) can be used, because we already have a rule that limits the places that TYPE(*) can be used.

It might prove less confusing to use a new term e.g. "unknown polymorphic" but that probably needs even more edits.

2.2 Assumed-rank objects

1 The syntax rule R515 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*

2 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*.

R522x1 *assumed-rank-spec* **is** ..

C535x1 An assumed-rank entity shall be a dummy variable that does not have the CODIMENSION or VALUE attribute.

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

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

2.3 OPTIONAL attribute

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

2 The constraint C1255 of 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.

3 Procedures

3.1 Characteristics of dummy data objects

1 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.

3.2 Explicit interface

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

3.3 Argument association

1 An assumed-rank dummy argument may correspond to an actual argument of any rank. If the actual argument is scalar, the dummy argument has rank zero and the shape and bounds are arrays of zero size. If the actual argument is an array, the bounds of the dummy argument are assumed from the actual argument. The value of the lower and upper bound of dimension N of the dummy argument are equal to the result of applying the LBOUND and UBOUND intrinsic inquiry functions to the actual argument with DIM=N specified.

2 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.

3 When 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 a C descriptor that describes an allocated allocatable variable, the variable is deallocated on entry to the Fortran procedure.

4 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).

NOTE 3.1

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

3.4 Intrinsic procedures

3.4.1 SHAPE

1 The description of 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 of [(SIZE (ARRAY, I), I=1, RANK (ARRAY))]

3.4.2 SIZE

1 The description of 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 a value of −1 if DIM is present and equal to the rank of ARRAY, and a negative value that is equal to PRODUCT ([(SIZE (ARRAY, I), I=1, RANK (ARRAY))]) if DIM is not present;

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

3.4.3 UBOUND

1 The description of UBOUND in ISO/IEC 1539-1:2010 is changed for an assumed-rank object that is associated with an assumed-size array; the result has a value of LBOUND (ARRAY, RANK (ARRAY)) - 2.

4 New Intrinsic procedure

4.1 General

1 Detailed specification of the RANK generic intrinsic procedure is provided in 4.2. 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 RANK intrinsic function is a pure function.

4.2 RANK (A)

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

2 **Class.** Inquiry function.

3 **Arguments.**

A shall be a scalar or array of any type.

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

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

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

5 Interoperability with C

5.1 C descriptors

1 A **C descriptor** is a struct of type CFI_cdesc_t. The C descriptor along with library functions with standard prototypes provide the means for describing an assumed-shape, assumed-rank, allocatable, or data pointer object within a C function. This struct is defined in the file `ISO_Fortran_binding.h`.

5.2 ISO_Fortran_binding.h

5.2.1 Summary of contents

1 The `ISO_Fortran_binding.h` file contains the definitions of the C structs CFI_cdesc_t, CFI_dim_t, and CFI_bounds_t, typedef definitions for CFI_attribute_t, CFI_index_t, CFI_rank_t, and CFI_type_t, macro definitions that expand to integer constants, and C prototypes for the C functions CFI_allocate, CFI_deallocate, CFI_is_contiguous, CFI_bounds_to_cdesc, and CFI_cdesc_to_bounds. The contents of `ISO_Fortran_binding.h` can be used by a C function to interpret a 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 allocatable, data pointer, assumed-rank, or assumed-shape dummy arguments.

2 `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.

3 A C source file that includes the header `ISO_Fortran_binding.h` shall not use any names starting 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.

5.2.2 CFI_cdesc_t

1 CFI_cdesc_t is a named struct type defined by a typedef, containing a flexible array member. It shall contain at least the following members. The first three members of the struct 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 NULL. If the object has zero size, the value is not NULL 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 element for which each subscript has the value of the corresponding lower bound.

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

int 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 descriptor.

CFI_rank_t rank; equal to the number of dimensions of the 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 identifier for the type of the object. Each interoperable intrinsic C type has an identifier. The identifier for interoperable structures has a different value from any of the identifiers for intrinsic types. An identifier is also provided to indicate that the type of the object is unknown. Its value

is different from that of any other type identifier. Macros and the corresponding values for the identifiers are defined in the `ISO_Fortran_binding.h` file. 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 a data pointer, allocatable, assumed-shape, or assumed-size. Macros and the corresponding values for the attribute codes are supplied in the `ISO_Fortran_binding.h` file. 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 array contains the lower bound, extent, and stride multiplier information for the corresponding dimension of the object. The number of elements in the array shall be equal to the rank of the object.

5.2.3 CFI_dim_t

1 CFI_dim_t is a named struct type defined by a typedef. It is used to represent lower bound, extent, and stride multiplier information for one dimension of an array. 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 of an array for a specified dimension.

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

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

2 If the actual argument is of type CHARACTER, or is of assumed type eventually associated with an actual argument of type CHARACTER, the member elem_len shall contain the sizeof() of a variable of character length 1 of that type and kind. The first element of member dim shall contain a lower bound of 1 with a stride equal to elem_len and upper bound equal to the character length of the actual argument; all other elements shall correspond to a dimension one less than for non-CHARACTER types.

Unresolved Technical Issue TR2

For the discussion about dim and elem_len members to be relevant, the above, "or is of assumed type eventually associated with an actual argument of type CHARACTER" presumes that the assumed-type actual argument is passed by a C descriptor. There is no such requirement, and it contradicts a big part of the expected use of type(*). Could be corected by replacing the quoted text with "and the corresponding formal parameter is CFI_cdesc_t *". This change would also cover the case of existing interoperabiity where a CHARACTER actual corresponds to a char formal parameter, which works fine and has nothing to do with descriptors.

Unresolved Technical Issue TR3

This approach has not achieved consensus for several reasons.

People want the rank field to match the RANK intrinsic (and, generally, the rank of the corresponding Fortran object).

The elem_len field would no longer have the value of C_SIZEOF() applied to an element of the object in Fortran. Currently the only interoperable character kind is 'char' for which sizeof() is defined to return 1, so the elem_len field would contain 1. This could change in the future if we added interoperability for wchar_t type.

The other main contenders are:

Unresolved Technical Issue TR3 (cont.)

(1) fold the character length into elem_len (which, unless we add support for wchar_t, would be equal to the value of elem_len).

(2) add an additional character length member.

(3) require that the value (or perhaps the lower 4 bits of the value) of CFI_type_char be equal to 1 and for CFI_type_wchar_t (if we add it) be equal to sizeof(wchar_t).

In any case more edits are required.

3 If any actual argument associated with the dummy argument is an assumed-size array, the array shall be simply contiguous, the member attribute shall be CFI_attribute_unknown_size and the member extent of the last dimension of member dim is equal to (CFI_index_t)-2.

5.2.4 CFI_bounds_t

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

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

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

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

5.2.5 Macros

1 The following macros are defined in `ISO_Fortran_binding.h`. Except for CFI_DESC_T, each evaluates to an integer constant expression suitable for use in `#if` preprocessing directives.

2 CFI_CDESC_T - a function-like macro that takes one argument, which is the rank of the descriptor to create, and evaluates to a type suitable for declaring a descriptor of that rank. A pointer to a variable declared using CFI_CDESC_T can be cast to CFI_cdesc_t *.

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

4 CFI_VERSION - an integer constant that encodes the version of the `ISO_Fortran_binding.h` header file containing this macro.

NOTE 5.1

The intent is that the version should be increased every time that the header is incompatibly changed, and that the version in a descriptor may be used to provide a level of upwards compatibility, by using means not defined by ISO/IEC TR 29113.

NOTE 5.2

The following code uses CFI_CDESC_T to declare a descriptor of rank 5 and pass it to CFI_deallocate.

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

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

Table 5.1: **Macros specifying attribute codes**

Macro	Code
CFI_attribute_assumed	assumed
CFI_attribute_allocatable	allocatable
CFI_attribute_pointer	pointer
CFI_attribute_unknown_size	assumed-size

6 CFI_attribute_pointer specifies an object with the Fortran POINTER attribute. CFI_attribute_allocatable specifies an object with the Fortran ALLOCATABLE attribute. CFI_attribute_assumed specifies an assumed-shape object or an assumed-rank object that is not allocatable, a pointer, or associated with an assumed-size argument. CFI_attribute_unknown_size specifies an object that is, or is argument-associated with, an assumed-size dummy argument.

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

Table 5.2: **Macros specifying type codes**

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

NOTE 5.3

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.

8 The macros in Table 5.3 are for use as error codes. The macro CFI_SUCCESS shall be defined to be the integer constant 0.

9 The values of the error codes returned for the error conditions listed below are named by the indicated macros. The value of each macro shall be nonzero and shall be different from the values of the other macros specified in this section. Error conditions other than those listed in this section should be indicated by error codes different from the values of the macros named in this section.

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

Table 5.3: **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 NULL in a context that requires a non-null value.
CFI_ERROR_BASE_ADDR_NOT_NULL	The base address member of a C descriptor is not NULL in a context that requires a null value.
CFI_INVALID_ELEM_LEN	The value of the element length member of a C descriptor is not valid.
CFI_INVALID_RANK	The value of the rank member of a C descriptor is not valid.
CFI_INVALID_TYPE	The value of the type member of a C descriptor is not valid.
CFI_INVALID_ATTRIBUTE	The value of the attribute member of a C descriptor is not valid.
CFI_INVALID_EXTENT	The value of the extent member of a CFI_dim_t structure is not valid.
CFI_INVALID_SM	The value of the stride multiplier member of a CFI_dim_t structure is not valid.
CFI_INVALID_UPPER_BOUND	The value of the upper bound member of a CFI_bounds_t structure is not valid.
CFI_INVALID_STRIDE	The value of the stride member of a CFI_bounds_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.

5.2.6 Functions

5.2.6.1 General

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

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

3 Some of the functions described in 5.2.6 return an integer value that indicates if an error condition was detected. If no error condition was detected an integer zero is returned; if an error condition was detected, a nonzero integer is returned. A list of error conditions and macro names for the corresponding error codes is supplied in 5.2.5. A processor is permitted to detect other error conditions. If an invocation of a function defined in 5.2.6 could detect more than one error condition and an error condition is detected, which error condition is detected is processor dependent.

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

5.2.6.2 CFI_cdesc_t * CFI_create_cdesc (const size_t elem_len, const CFI_rank_t rank, const CFI_type_t type, const CFI_attribute_t attribute);

1 **Description.** CFI_create_cdesc allocates memory using malloc for a C descriptor of the rank specified by the rank argument. If the memory allocation is successful and the values of the arguments are valid, the elem_len, rank, type, and attribute members are initialized to the values of the corresponding arguments, the version member is initialized to CFI_VERSION that is specified in the `ISO_Fortran_binding.h` header file, the base_addr member is initialized to NULL, and the function result is a pointer to the C descriptor created. If memory allocation fails or any of the arguments have invalid values, the function result is NULL.

5.2.6.3 int CFI_initialize_cdesc (CFI_cdesc_t * , const size_t elem_len, const CFI_rank_t rank, const CFI_type_t type, const CFI_attribute_t attribute);

1 **Description.** CFI_initalize_cdesc initializes members of an existing C descriptor. If the values of the arguments are valid, the elem_len, rank, type, and attribute members are initialized to the values of the corresponding arguments, the version member is initialized to CFI_VERSION that is specified in the `ISO_Fortran_binding.h` header file, the base_addr member is initialized to NULL, and the function result is zero. If any of the arguments have invalid values, the function result is nonzero and the C descriptor is not modified. The function result is an error indicator.

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

1 **Description.** CFI_address returns the address of the object described by the C descriptor or an element of it. The object shall not be an unallocated allocatable or a pointer that is not associated. The number of elements in the subscripts array shall be greater than or equal to the rank r of the object. If the object is an array, the result is the address of the element of the object that the first r elements of the subscripts array would specify if used as subscripts. If the object is scalar, the result is its address and the subscripts array is ignored.

5.2.6.5 int CFI_associate (CFI_cdesc_t * , void *, base_addr, const CFI_bounds_t bounds[]);

1 **Description.** CFI_associate associates memory with an assumed-shape or Fortran pointer object described by the C descriptor. If the object has rank zero, the bounds[] argument is ignored and the amount of memory required for the object is specified by the elem_len member of the descriptor. If the rank is greater than zero, the amount of memory required for the object is specified by the bounds[] array and elem_len. If the base_addr is not NULL, the amount of memory starting at address base_addr that is currently allocated by the program shall be large enough to provide storage for the object. If the base_addr is NULL, memory for the object is

allocated using malloc and base_addr is the value returned by malloc. The function result is an error indicator. If memory allocation during execution of the function fails, the attribute member of the C descriptor is not equal to CFI_attribute_assumed or CFI_attribute_pointer, or the values of members of the C descriptor or the first rank elements of the bounds[ ] array are invalid, the function result is nonzero. Otherwise, the function result is zero and the C descriptor is updated.

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

1 **Description.** CFI_allocate allocates memory for an object using the same mechanism as the Fortran ALLOCATE statement. If the base address in the C descriptor is not NULL on entry and the object is allocatable, the C descriptor is not modified and CFI_ERROR_BASE_ADDR_NOT_NULL is returned. If the C descriptor is not for an allocatable or pointer data object, the C descriptor is not modified and CFI_INVALID_ATTRIBUTE is returned. The number of elements in the bounds array shall be greater than or equal to the rank specified in the descriptor. The stride values are ignored and assumed to be one. If a memory allocation failure is detected, the C descriptor is not modified and CFI_ERROR_MEM_ALLOCATION is returned. On successful execution of CFI_allocate, the supplied bounds override any current dimension information in the descriptor. The C descriptor is updated by this function. The result is an error indicator.

5.2.6.7 int CFI_deallocate (CFI_cdesc_t *);

1 **Description.** CFI_deallocate deallocates memory for an object that was allocated using the same mechanism as the Fortran ALLOCATE statement. It uses the same mechanism as the Fortran DEALLOCATE statement. If the base address in the C descriptor is NULL on entry, the C descriptor is not modified and CFI_ERROR_BASE_ADDR_NULL is returned. If the C descriptor is not for an allocatable or pointer data object, the C descriptor is not modified and CFI_INVALID_ATTRIBUTE is returned. 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). The C descriptor is updated by this function. The result is an error indicator.

5.2.6.8 int CFI_is_contiguous (const CFI_cdesc_t *);

1 **Description.** CFI_is_contiguous returns 1 if the argument is a valid C descriptor and the object described is determined to be contiguous, and 0 otherwise.

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

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

5.2.7 Restrictions on the use of C descriptors

1 The base address in the C descriptor for a data pointer may be modified by assignment. The initial allocation status of an allocatable object shall be unallocated. Subsequently, the base address in a C descriptor that describes an allocatable object shall be modified only by the CFI_allocate or CFI_deallocate functions or because of allocation or deallocation in a Fortran procedure.

2 The elem_len, version, rank, type, and attribute members of a C descriptor shall be defined with valid values before the descriptor is used to represent an object, and once defined the values of these members shall not be redefined or become undefined while the members of the descriptor are capable of being referenced.

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

4 A C descriptor that describes an object of type CHARACTER shall have rank ≥ 1, and dim[0].sm = elem_len.

5.2.8 Interoperability of procedures and procedure interfaces

1 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.

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

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

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

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

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

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

5.3 Interaction with the DEALLOCATE statement

1 The DEALLOCATE statement shall treat a pointer whose target was allocated using CFI_allocate in exactly the same way as if it were allocated using an ALLOCATE statement.

Annex A

(Informative)

Extended notes

A.1 Clause 2 notes

A.1.1 Using assumed-type dummy arguments

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

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

2 In C:

```
void EXAMPLE_send ( void * buffer, int n);
```

3 In the Fortran module:

```
interface EXAMPLE_send
   subroutine EXAMPLE_send (buffer, n) bind(c,name="EXAMPLE_send")
      type(*),dimension(*) :: buffer
      integer(c_int),value :: n
   end subroutine EXAMPLE_send
   module procedure EXAMPLE_send_i8
end interface EXAMPLE_send

...

subroutine EXAMPLE_send_i8 (buffer, n)
   type(*),dimension(*) :: buffer
   integer(selected_int_kind(17)) :: n
   call EXAMPLE_send(buffer, int(n,c_int))
end subroutine EXAMPLE_send_i8
```

A.1.2 General association with a void * C parameter

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

1 In C:

```
 void EXAMPLE_send_abstract ( void * buffer, int n);
 void EXAMPLE_send_abstract_new ( void * buffer_desc);
```

2 In the Fortran module:

```
interface EXAMPLE_send_abstract
   subroutine EXAMPLE_send_old (buffer, n) bind(c,name="EXAMPLE_send_abstract")
      type(*), dimension(*) :: buffer ! Passed by simple address
      integer(c_int),value :: n
```

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

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

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

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

A.1.3 Casting TYPE (*) in Fortran

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

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

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

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

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

A.1.4 Simplifying interfaces for arbitrary rank procedures

Example of assumed-rank usage in Fortran

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

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

```
   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 5 notes

1 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.

2 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
```

3 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;
  }
```