

# TR 29113 WORKING DRAFT

N1838

**8th November 2010 11:09**

This is an internal working document of J3.



## Contents

1	Overview . . . . .	1
1.1	Scope . . . . .	1
1.2	Normative references . . . . .	1
1.3	Terms and definitions . . . . .	1
1.4	Compatibility . . . . .	2
1.4.1	New intrinsic procedures . . . . .	2
1.4.2	Fortran 2008 compatibility . . . . .	2
2	Type specifiers and attributes . . . . .	3
2.1	Assumed-type objects . . . . .	3
2.2	Assumed-rank objects . . . . .	3
2.3	OPTIONAL attribute . . . . .	4
3	Procedures . . . . .	5
3.1	Characteristics of dummy data objects . . . . .	5
3.2	Explicit interface . . . . .	5
3.3	Argument association . . . . .	5
3.4	Intrinsic procedures . . . . .	5
3.4.1	SHAPE . . . . .	5
3.4.2	SIZE . . . . .	6
3.4.3	UBOUND . . . . .	6
4	New Intrinsic procedure . . . . .	7
4.1	General . . . . .	7
4.2	RANK (A) . . . . .	7
5	Interoperability with C . . . . .	9
5.1	C descriptors . . . . .	9
5.2	ISO_Fortran_binding.h . . . . .	9
5.2.1	Summary of contents . . . . .	9
5.2.2	CFL_cdesc_t . . . . .	9
5.2.3	CFL_dim_t . . . . .	10
5.2.4	CFL_bounds_t . . . . .	11
5.2.5	Macros . . . . .	11
5.2.6	Functions . . . . .	14
5.2.7	Restrictions on the use of C descriptors . . . . .	16
5.2.8	Interoperability of procedures and procedure interfaces . . . . .	16
5.3	Interaction with the DEALLOCATE statement . . . . .	17
Annex A	(informative) Extended notes . . . . .	19
A.1	Clause 2 notes . . . . .	19
A.1.1	Using assumed-type dummy arguments . . . . .	19
A.1.2	General association with a void * C parameter . . . . .	19
A.1.3	Casting TYPE (*) in Fortran . . . . .	20
A.1.4	Simplifying interfaces for arbitrary rank procedures . . . . .	20
A.2	Clause 5 notes . . . . .	21



# List of Tables

- 5.1 Macros specifying attribute codes . . . . . 12
- 5.2 Macros specifying type codes . . . . . 12
- 5.3 Macros specifying error codes . . . . . 13

## 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.

### Unresolved Technical Issue TR14

Need to add interoperability of interfaces with assumed-length character dummy arguments to the above paragraph.

- 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:
 

Overview	Clause 1
Type specifiers and attributes	Clause 2
Procedure interfaces	Clause 3
New intrinsic procedure	Clause 4
Interoperability with C	Clause 5
- 7 It also contains the following nonnormative material:
 

Extended notes	A
----------------	---



# 1 Technical Report — Further Interoperability of Fortran with 2 C — 3 1 Overview

## 4 1.1 Scope

5 1 ISO/IEC TR 29113 specifies the form and establishes the interpretation of facilities that extend the Fortran lan-  
6 guage defined by ISO/IEC 1539-1:2010. The purpose of ISO/IEC TR 29113 is to promote portability, reliability,  
7 maintainability and efficient execution of programs containing parts written in Fortran and parts written in C for  
8 use on a variety of computing systems.

## 9 1.2 Normative references

10 1 The following referenced standards are indispensable for the application of this document.

11 2 ISO/IEC 1539-1:2010, *Information technology—Programming languages—Fortran*

12 3 ISO/IEC 9899:1999, *Information technology—Programming languages—C*

## 13 1.3 Terms and definitions

14 1 For the purposes of this document, the following terms and definitions apply. Terms not defined in ISO/IEC TR  
15 29113 are to be interpreted according to ISO/IEC 1539-1:2010.

### 16 1 1.3.1

#### 17 **assumed-rank object**

18 dummy variable whose rank is assumed from its effective argument

### 19 1 1.3.2

#### 20 **assumed-type object**

21 dummy variable whose type and type parameters are assumed from its effective argument

### 22 1 1.3.3

#### 23 **C descriptor**

24 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.

#### **Unresolved Technical Issue TR11**

Note 1.1 should be expanded to include assumed-length character objects. It is possible to have an assumed-length character object that is in none of the currently listed categories.

1 **1.4 Compatibility**

2 **1.4.1 New intrinsic procedures**

3 1 ISO/IEC TR 29113 defines an intrinsic procedure in addition to those specified in ISO/IEC 1539-1:2010. There-  
4 fore, a Fortran program conforming to ISO/IEC 1539-1:2010 might have a different interpretation under ISO/IEC  
5 TR 29113 if it invokes an external procedure having the same name as the new intrinsic procedure, unless that  
6 procedure is specified to have the EXTERNAL attribute.

7 **1.4.2 Fortran 2008 compatibility**

8 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.

- 1 3 An assumed-rank variable shall appear only as a dummy argument, an actual argument associated with a dummy  
2 argument that is assumed-rank, the argument of the C\_LOC function in the ISO\_C\_BINDING intrinsic module,  
3 or the first argument in a reference to an intrinsic inquiry function. The RANK inquiry intrinsic may be used to  
4 inquire about the rank of an array or scalar object.
- 5 4 The rank of an assumed-rank object may be zero.

#### Unresolved Technical Issue TR9

[Optional.] A Note might be helpful that explains that assumed-rank is a Fortran-only concept. The C programmer will see a C descriptor corresponding to an assumed-rank argument that has an attribute code of CFI\_attribute\_allocatable if that dummy in the Fortran interface also has the allocatable attribute, CFI\_attribute\_pointer for pointer, and CFI\_attribute\_assumed otherwise. An assumed-rank argument that is not allocatable or pointer looks like an assumed-shape argument to the C function, except that the rank could be zero. Alternatively, a Note like this could follow Table 5.1 where the attribute codes are described.

## 6 2.3 OPTIONAL attribute

- 7 1 The OPTIONAL attribute may be specified for a dummy argument in a procedure interface that has the BIND  
8 attribute.
- 9 2 The constraint C1255 of subclause 12.6.2.2 of ISO/IEC 1539-1:2010 is replaced by
- 10 C1255 (R1229) If *proc-language-binding-spec* is specified for a procedure, each dummy argument of the procedure  
11 shall be an interoperable procedure (15.3.7) or an interoperable variable (15.3.5, 15.3.6) that does not have  
12 both the OPTIONAL and VALUE attributes. If *proc-language-binding-spec* is specified for a function,  
13 the function result shall be an [interoperable](#) scalar variable.

#### Unresolved Technical Issue TR5

Constraint C516 in ISO/IEC 1539-1:2010, 5.3.1 Attributes/Constraints, says that ALLOCATABLE, POINTER, and OPTIONAL attributes cannot be specified for arguments in a bind(c) interface. We need to say that constraint is removed by the TR.

## 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).

#### Unresolved Technical Issue TR8

In ISO/IEC 1539-1:2010, 12.5.2.4 Ordinary dummy variables, paragraph 13, we allow sequence association by argument association if the actual argument is an array element and the corresponding dummy is "... an array that is not an assumed-shape, pointer, or polymorphic array.". This list of "is not" needs to also include assumed-rank.

#### 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

1 (ARRAY, I), I=1, RANK (ARRAY)) ]

## 2 3.4.2 SIZE

3 1 The description of SIZE in ISO/IEC 1539-1:2010 is changed in the following cases:

- 4 (1) for an assumed-rank object that is associated with an assumed-size array, the result has a value of  $-1$   
5 if DIM is present and equal to the rank of ARRAY, and a negative value that is equal to PRODUCT  
6 ( [ (SIZE (ARRAY, I), I=1, RANK (ARRAY)) ] ) if DIM is not present;  
7 (2) for an assumed-rank object that is associated with a scalar, the result has a value of 1.

## 8 3.4.3 UBOUND

9 1 The description of UBOUND in ISO/IEC 1539-1:2010 is changed for an assumed-rank object that is associated  
10 with an assumed-size array; the result has a value of LBOUND (ARRAY, RANK (ARRAY)) - 2.

### Unresolved Technical Issue TR12

In 2.2, paragraph 3, we allow assumed-rank objects to be the first argument to intrinsic inquiry functions. Several have a description of a DIM argument that includes “a value in the range  $1 \leq \text{DIM} \leq n$ , where  $n$  is the rank of ARRAY”. If the rank is zero, as is allowed for assumed-rank, there is a contraction. Either several functions need fixes, or the list of inquiry functions that allow assumed-rank arguments should be narrowed.

## 1    **4    New Intrinsic procedure**

### 2    **4.1    General**

3    1 Detailed specification of the RANK generic intrinsic procedure is provided in 4.2. The types and type parameters  
4    of the RANK intrinsic procedure argument and function result are determined by this specification. The “Argu-  
5    ment” paragraph specifies requirements on the [actual arguments](#) of the procedure. The RANK intrinsic function  
6    is a pure function.

### 7    **4.2    RANK (A)**

8    1 **Description.** Rank of a data object.

9    2 **Class.** [Inquiry function](#).

10   3 **Arguments.**

11    A            shall be a scalar or array of any type.

12   4 **Result Characteristics.** Default integer scalar.

13   5 **Result Value.** The result is the rank of A.

14   6 **Example.** For an array X declared REAL :: X(:, :, :), RANK(X) is 3.



## 1 5 Interoperability with C

### 2 5.1 C descriptors

3 1 A **C descriptor** is a struct of type `CFI_cdesc_t`. The C descriptor along with library functions with standard  
4 prototypes provide the means for describing an assumed-shape, assumed-rank, allocatable, or data pointer object  
5 within a C function. This struct is defined in the file `ISO_Fortran_binding.h`.

### 6 5.2 ISO\_Fortran\_binding.h

#### 7 5.2.1 Summary of contents

8 1 The `ISO_Fortran_binding.h` file contains the definitions of the C structs `CFI_cdesc_t`, `CFI_dim_t`, and `CFI-`  
9 `bounds_t`, typedef definitions for `CFI_attribute_t`, `CFI_index_t`, `CFI_rank_t`, and `CFI_type_t`, macro definitions  
10 that expand to integer constants, and C prototypes for the C functions `CFI_allocate`, `CFI_deallocate`, `CFI_is-`  
11 `contiguous`, `CFI_bounds_to_cdesc`, and `CFI_cdesc_to_bounds`. The contents of `ISO_Fortran_binding.h` can be  
12 used by a C function to interpret a C descriptor and allocate and deallocate objects represented by a C descriptor.  
13 These provide a means to specify a C prototype that interoperates with a Fortran interface that has allocatable,  
14 data pointer, assumed-rank, or assumed-shape dummy arguments.

15 2 `ISO_Fortran_binding.h` may be included in any order relative to the standard C headers, and may be included  
16 more than once in a given scope, with no effect different from being included only once, other than the effect on  
17 line numbers.

18 3 A C source file that includes the header `ISO_Fortran_binding.h` shall not use any names starting `CFI_` that are  
19 not defined in the header. All names defined in the header begin with `CFI_` or an underscore character, or are  
20 defined by a standard C header that it includes.

#### 21 5.2.2 CFI\_cdesc\_t

22 1 `CFI_cdesc_t` is a named struct type defined by a typedef, containing a flexible array member. It shall contain at  
23 least the following members. The first three members of the struct shall be `base_addr`, `elem_len`, and `version` in  
24 that order. The final member shall be `dim`, with the other members after `version` and before `dim` in any order.

25 **void \* base\_addr;** If the object is an unallocated allocatable or a pointer that is disassociated, the value is  
26 NULL. If the object has zero size, the value is not NULL but is otherwise processor-dependent. Otherwise,  
27 the value is the base address of the object being described. The base address of a scalar is its C address.  
28 The base address of an array is the C address of the element for which each subscript has the value of the  
29 corresponding lower bound.

30 **size\_t elem\_len;** equal to the `sizeof()` of an element of the object being described

31 **int version;** shall be set equal to the value of `CFI_VERSION` in the `ISO_Fortran_binding.h` header file that  
32 defined the format and meaning of this descriptor.

33 **CFI\_rank\_t rank;** equal to the number of dimensions of the object being described. If the object is a scalar,  
34 the value is zero. `CFI_rank_t` shall be a typedef name for a standard integer type capable of representing  
35 the largest supported rank.

36 **CFI\_type\_t type;** equal to the identifier for the type of the object. Each interoperable intrinsic C type has an  
37 identifier. The identifier for interoperable structures has a different value from any of the identifiers for  
38 intrinsic types. An identifier is also provided to indicate that the type of the object is unknown. Its value

1 is different from that of any other type identifier. Macros and the corresponding values for the identifiers  
 2 are defined in the `ISO_Fortran_binding.h` file. `CFI_type_t` shall be a typedef name for a standard integer  
 3 type capable of representing the values for the supported type specifiers.

4 **CFI\_attribute\_t attribute;** equal to the value of an attribute code that indicates whether the object described  
 5 is a data pointer, allocatable, assumed-shape, or assumed-size. Macros and the corresponding values for the  
 6 attribute codes are supplied in the `ISO_Fortran_binding.h` file. `CFI_attribute_t` shall be a typedef name  
 7 for a standard integer type capable of representing the values of the attribute codes.

8 **CFI\_dim\_t dim[ ];** Each element of the array contains the lower bound, extent, and stride multiplier information  
 9 for the corresponding dimension of the object. The number of elements in the array shall be equal to the  
 10 rank of the object.

### 11 5.2.3 CFI\_dim\_t

12 1 `CFI_dim_t` is a named struct type defined by a typedef. It is used to represent lower bound, extent, and stride  
 13 multiplier information for one dimension of an array. `CFI_index_t` is a typedef name for a standard signed integer  
 14 type capable of representing the result of subtracting two pointers. `CFI_dim_t` contains at least the following  
 15 members in any order:

16 **CFI\_index\_t lower\_bound;** equal to the value of the lower bound of an array for a specified dimension.

17 **CFI\_index\_t extent;** equal to the number of elements of an array along a specified dimension.

18 **CFI\_index\_t sm;** equal to the stride multiplier for a dimension. The value is the distance in bytes between the  
 19 beginnings of successive elements of the array along a specified dimension.

#### Unresolved Technical Issue TR4

Requirement 8(c) in N1820 says that the lower bounds for an assumed-shape dummy (or an assumed-rank dummy that is not allocatable or pointer) are assumed to be zero, as opposed to the default of 1 in Fortran. This needs to be stated. Also, note the contradiction in the following paragraph which specifies a lower bound of 1.

#### Unresolved Technical Issue TR6

Constraint 5 in N1820 says that “Array strides of zero and overlapping elements are not supported.” This needs to be stated, either here or in 5.2.7 Restrictions on the use of C descriptors.

20 2 If the actual argument is of type CHARACTER, or is of assumed type eventually associated with an actual  
 21 argument of type CHARACTER, the member `elem_len` shall contain the `sizeof()` of a variable of character length  
 22 1 of that type and kind. The first element of member `dim` shall contain a lower bound of 1 with a stride equal  
 23 to `elem_len` and upper bound equal to the character length of the actual argument; all other elements shall  
 24 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 corrected by replacing the quoted text with “and the corresponding formal parameter is `CFI_cdesc_t *`”. This change would also cover the case of existing interoperability 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:

(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.

- 1 3 If any actual argument associated with the dummy argument is an assumed-size array, the array shall be simply  
2 contiguous, the member attribute shall be `CFI_attribute_unknown_size` and the member extent of the last dimen-  
3 sion of member `dim` is equal to  $(CFI\_index\_t)-2$ .

#### 4 **5.2.4 CFI\_bounds\_t**

- 5 1 `CFI_bounds_t` is a named struct type defined by a typedef. It is used to represent bounds and stride information  
6 for one dimension of an array. `CFI_bounds_t` contains at least the following members in any order:

7 **CFI\_index\_t lower\_bound;** equal to the value of the lower bound of an array for a specified dimension.

8 **CFI\_index\_t upper\_bound;** equal to the value of the upper bound of an array for a specified dimension.

9 **CFI\_index\_t stride;** equal to the difference between the subscript values of consecutive elements of an array  
10 along a specified dimension.

#### 11 **5.2.5 Macros**

- 12 1 The following macros are defined in `ISO_Fortran_binding.h`. Except for `CFIDESC_T`, each evaluates to an  
13 integer constant expression suitable for use in `#if` preprocessing directives.

14 2 `CFI_CDESC_T` - a function-like macro that takes one argument, which is the rank of the descriptor to create,  
15 and evaluates to a type suitable for declaring a descriptor of that rank. A pointer to a variable declared using  
16 `CFI_CDESC_T` can be cast to `CFI_cdesc_t *`.

17 3 `CFI_MAX_RANK` - a value equal to the largest rank supported. The value shall be greater than or equal to 15.

18 4 `CFI_VERSION` - an integer constant that encodes the version of the `ISO_Fortran_binding.h` header file con-  
19 taining this macro.

**Unresolved Technical Issue TR13**

The value of `CFI_VERSION` should be described as being processor dependent.

**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);
```

- 1 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

- 2 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 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

## Macros specifying type codes (cont.)

Macro	C Type
CFL_type_int_fast64_t	fast64_t
CFL_type_intmax_t	intmax_t
CFL_type_intptr_t	intptr_t
CFL_type_float	float
CFL_type_double	double
CFL_type_long_double	long double
CFL_type_float_Complex	float Complex
CFL_type_double_Complex	double Complex
CFL_type_long_double_Complex	long double Complex
CFL_type_Bool	_Bool
CFL_type_char	char
CFL_type_cptra	void *
CFL_type_cfunptr	pointer to a function
CFL_type_unspecified	unspecified

**NOTE 5.3**

The specifiers for two intrinsic types can have the same value. For example, CFL\_type\_int and CFL\_type\_int32\_t might have the same value.

- 1 8 The macros in Table 5.3 are for use as error codes. The macro CFL\_SUCCESS shall be defined to be the integer  
2 constant 0.
- 3 9 The values of the error codes returned for the error conditions listed below are named by the indicated macros.  
4 The value of each macro shall be nonzero and shall be different from the values of the other macros specified in  
5 this section. Error conditions other than those listed in this section should be indicated by error codes different  
6 from the values of the macros named in this section.
- 7 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
CFL_SUCCESS	No error detected.
CFL_ERROR_BASE_ADDR_NULL	The base address member of a C descriptor is NULL in a context that requires a non-null value.
CFL_ERROR_BASE_ADDR_NOT_NULL	The base address member of a C descriptor is not NULL in a context that requires a null value.
CFL_INVALID_ELEM_LEN	The value of the element length member of a C descriptor is not valid.
CFL_INVALID_RANK	The value of the rank member of a C descriptor is not valid.
CFL_INVALID_TYPE	The value of the type member of a C descriptor is not valid.
CFL_INVALID_ATTRIBUTE	The value of the attribute member of a C descriptor is not valid.

## Macros specifying error codes

(cont.)

Macro	Error
CFL_INVALID_EXTENT	The value of the extent member of a CFL_dim_t structure is not valid.
CFL_INVALID_SM	The value of the stride multiplier member of a CFL_dim_t structure is not valid.
CFL_INVALID_UPPER_BOUND	The value of the upper bound member of a CFL_bounds_t structure is not valid.
CFL_INVALID_STRIDE	The value of the stride member of a CFL_bounds_t structure is not valid.
CFL_INVALID_DESCRIPTOR	A general error condition for C descriptors.
CFL_ERROR_MEM_ALLOCATION	Memory allocation failed.
CFL_ERROR_OUT_OF_BOUNDS	A reference is out of bounds.

1 **5.2.6 Functions**2 **5.2.6.1 General**

3 1 Functions are provided for use in C functions. These functions and the structure of the C descriptor provide the  
4 C program with the capability to interact with Fortran procedures that have allocatable, data pointer, assumed-  
5 rank, or assumed-shape arguments.

6 2 Within a C function, allocatable objects shall be allocated or deallocated only through execution of the CFL-  
7 allocate and CFL\_deallocate functions. Pointer objects can become associated with a target by execution of the  
8 CFL\_allocate function.

9 3 Some of the functions described in 5.2.6 return an integer value that indicates if an error condition was detected.  
10 If no error condition was detected an integer zero is returned; if an error condition was detected, a nonzero integer  
11 is returned. A list of error conditions and macro names for the corresponding error codes is supplied in 5.2.5. A  
12 processor is permitted to detect other error conditions. If an invocation of a function defined in 5.2.6 could detect  
13 more than one error condition and an error condition is detected, which error condition is detected is processor  
14 dependent.

15 4 Prototypes for these functions are provided in the ISO\_Fortran\_binding.h file as follows:

16 **5.2.6.2 CFL\_cdesc\_t \* CFL\_create\_cdesc (const size\_t elem\_len, const CFL\_rank\_t rank, const CFL\_type\_t**  
17 **type, const CFL\_attribute\_t attribute);**

18 1 **Description.** CFL\_create\_cdesc allocates memory using malloc for a C descriptor of the rank specified by the  
19 rank argument. If the memory allocation is successful and the values of the arguments are valid, the elem-  
20 len, rank, type, and attribute members are initialized to the values of the corresponding arguments, the version  
21 member is initialized to CFL\_VERSION that is specified in the ISO\_Fortran\_binding.h header file, the base-  
22 addr member is initialized to NULL, and the function result is a pointer to the C descriptor created. If memory  
23 allocation fails or any of the arguments have invalid values, the function result is NULL.

24 **5.2.6.3 int CFL\_initialize\_cdesc (CFL\_cdesc\_t \* , const size\_t elem\_len, const CFL\_rank\_t rank, const CFL-**  
25 **type\_t type, const CFL\_attribute\_t attribute);**

26 1 **Description.** CFL\_initialize\_cdesc initializes members of an existing C descriptor. If the values of the arguments

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

6 **5.2.6.4 void \* CFI\_address ( const CFI\_cdesc\_t \*, const CFI\_index\_t subscripts[] );**

7 1 **Description.** `CFI_address` returns the address of the object described by the C descriptor or an element of it.  
 8 The object shall not be an unallocated allocatable or a pointer that is not associated. The number of elements  
 9 in the `subscripts` array shall be greater than or equal to the rank  $r$  of the object. If the object is an array, the  
 10 result is the address of the element of the object that the first  $r$  elements of the `subscripts` array would specify if  
 11 used as subscripts. If the object is scalar, the result is its address and the `subscripts` array is ignored.

12 **5.2.6.5 int CFI\_associate ( CFI\_cdesc\_t \* , void \* , base\_addr, const CFI\_bounds\_t bounds[] );**

13 1 **Description.** `CFI_associate` associates memory with an assumed-shape or Fortran pointer object described by  
 14 the C descriptor. If the object has rank zero, the `bounds[]` argument is ignored and the amount of memory  
 15 required for the object is specified by the `elem_len` member of the descriptor. If the rank is greater than zero,  
 16 the amount of memory required for the object is specified by the `bounds[]` array and `elem_len`. If the `base_addr`  
 17 is not `NULL`, the amount of memory starting at address `base_addr` that is currently allocated by the program  
 18 shall be large enough to provide storage for the object. If the `base_addr` is `NULL`, memory for the object is  
 19 allocated using `malloc` and `base_addr` is the value returned by `malloc`. The function result is an error indicator.  
 20 If memory allocation during execution of the function fails, the `attribute` member of the C descriptor is not equal  
 21 to `CFI_attribute_assumed` or `CFI_attribute_pointer`, or the values of members of the C descriptor or the first rank  
 22 elements of the `bounds[]` array are invalid, the function result is nonzero. Otherwise, the function result is zero  
 23 and the C descriptor is updated.

24 **5.2.6.6 int CFI\_allocate ( CFI\_cdesc\_t \*, const CFI\_bounds\_t bounds[] );**

25 1 **Description.** `CFI_allocate` allocates memory for an object using the same mechanism as the Fortran `ALLOCATE`  
 26 statement. If the base address in the C descriptor is not `NULL` on entry and the object is allocatable, the C  
 27 descriptor is not modified and `CFI_ERROR_BASE_ADDR_NOT_NULL` is returned. If the C descriptor is not  
 28 for an allocatable or pointer data object, the C descriptor is not modified and `CFI_INVALID_ATTRIBUTE`  
 29 is returned. The number of elements in the `bounds` array shall be greater than or equal to the rank specified in  
 30 the descriptor. The stride values are ignored and assumed to be one. If a memory allocation failure is detected,  
 31 the C descriptor is not modified and `CFI_ERROR_MEM_ALLOCATION` is returned. On successful execution of  
 32 `CFI_allocate`, the supplied `bounds` override any current dimension information in the descriptor. The C descriptor  
 33 is updated by this function. The result is an error indicator.

34 **5.2.6.7 int CFI\_deallocate ( CFI\_cdesc\_t \* );**

35 1 **Description.** `CFI_deallocate` deallocates memory for an object that was allocated using the same mechanism as  
 36 the Fortran `ALLOCATE` statement. It uses the same mechanism as the Fortran `DEALLOCATE` statement. If  
 37 the base address in the C descriptor is `NULL` on entry, the C descriptor is not modified and `CFI_ERROR_BASE_-`  
 38 `ADDR_NULL` is returned. If the C descriptor is not for an allocatable or pointer data object, the C descriptor is  
 39 not modified and `CFI_INVALID_ATTRIBUTE` is returned. If the object is a pointer it shall be associated with  
 40 a target satisfying the conditions for successful deallocation by the Fortran `DEALLOCATE` statement (6.7.3.3 of  
 41 ISO/IEC 1539-1:2010). The C descriptor is updated by this function. The result is an error indicator.

42 **5.2.6.8 int CFI\_is\_contiguous ( const CFI\_cdesc\_t \* );**

43 1 **Description.** `CFI_is_contiguous` returns 1 if the argument is a valid C descriptor and the object described is  
 44 determined to be contiguous, and 0 otherwise.

1 **5.2.6.9** `int CFI_cdesc_to_bounds ( const CFI_cdesc_t *, CFI_bounds_t bounds[] );`

2 1 **Description.** `CFI_cdesc_to_bounds` computes upper bound and stride values based on the extent and stride  
 3 multiplier values in a C descriptor. The number of elements in the bounds array shall be equal to or greater  
 4 than the rank specified in the descriptor. The lower bounds in the bounds array become those in the input C  
 5 descriptor. Since computation of strides from stride multipliers requires the element size, the whole C descriptor  
 6 is used as one of the arguments. The result is an error indicator.

## 7 **5.2.7 Restrictions on the use of C descriptors**

- 8 1 The base address in the C descriptor for a data pointer may be modified by assignment. The initial allocation  
 9 status of an allocatable object shall be unallocated. Subsequently, the base address in a C descriptor that  
 10 describes an allocatable object shall be modified only by the `CFI_allocate` or `CFI_deallocate` functions or because  
 11 of allocation or deallocation in a Fortran procedure.
- 12 2 The `elem_len`, `version`, `rank`, `type`, and `attribute` members of a C descriptor shall be defined with valid values  
 13 before the descriptor is used to represent an object, and once defined the values of these members shall not be  
 14 redefined or become undefined while the members of the descriptor are capable of being referenced.
- 15 3 It is possible for a C function to acquire memory through a function such as `malloc` and associate that memory  
 16 with a data pointer in a C descriptor. A C descriptor associated with such memory shall not be supplied as an  
 17 argument to `CFI_deallocate` and a corresponding dummy argument in a called Fortran procedure shall not be  
 18 specified in a context that would cause the dummy argument to be deallocated.

### Unresolved Technical Issue TR7

Requirement 9(a) in N1820 says: “The requirements and constraints that apply to assumed-shape arrays also apply to ALLOCATABLE ones (including converting them to assumed-shape), excluding the interpretation of lower bounds of an assumed-shape array.” We need to say this, or verify that it is implied by the text somewhere.

- 19 4 A C descriptor that describes an object of type CHARACTER shall have  $\text{rank} \geq 1$ , and  $\text{dim}[0].\text{sm} = \text{elem\_len}$ .

## 20 **5.2.8 Interoperability of procedures and procedure interfaces**

- 21 1 The rules in this subclause replace the contents of paragraphs one and two of subclause 15.3.7 of ISO/IEC  
 22 1539-1:2010 entirely.
- 23 2 A Fortran procedure is **interoperable** if it has the **BIND attribute**, that is, if its interface is specified with a  
 24 *proc-language-binding-spec*.
- 25 3 A Fortran procedure interface is interoperable with a C function prototype if
- 26 (1) the interface has the **BIND attribute**,
  - 27 (2) either
    - 28 (a) the interface describes a function whose **result variable** is a scalar that is interoperable with  
 29 the result of the prototype or
    - 30 (b) the interface describes a subroutine and the prototype has a result type of void,
  - 31 (3) the number of dummy arguments of the interface is equal to the number of formal parameters of the  
 32 prototype,
  - 33 (4) the prototype does not have variable arguments as denoted by the ellipsis (...),
  - 34 (5) any dummy argument with the **VALUE attribute** is interoperable with the corresponding formal  
 35 parameter of the prototype, and
  - 36 (6) any dummy argument without the **VALUE attribute** corresponds to a formal parameter of the pro-  
 37 totype that is of a pointer type, and either

- 1 (a) the dummy argument is interoperable with an entity of the referenced type (C International  
2 Standard, 6.2.5, 7.17, and 7.18.1) of the formal parameter,  
3 (b) the dummy argument is a nonallocatable, nonpointer variable of type CHARACTER with  
4 assumed length, and corresponds to a formal parameter of type CFI\_cdesc\_t,  
5 (c) the dummy argument is allocatable, assumed-shape, assumed-rank, or a pointer, and corres-  
6 ponds to a formal parameter of the prototype that is a pointer to CFI\_cdesc\_t, or  
7 (d) the dummy argument is assumed-type and not allocatable, assumed-shape, assumed-rank, or  
8 a pointer, and corresponds to a formal parameter of the prototype that is a pointer to void.

### Unresolved Technical Issue TR10

In (6)(b) above, the corresponding C formal parameter should be a pointer to CFI\_cdesc\_t, similar to (6)(c).

- 9 4 If a dummy argument in an interoperable interface is of type CHARACTER and is allocatable or a pointer, its  
10 character length shall be deferred.  
11 5 If a dummy argument in an interoperable interface is allocatable, assumed-shape, assumed-rank, or a pointer,  
12 the corresponding formal parameter is interpreted as a pointer to a C descriptor for the effective argument in a  
13 reference to the procedure. The C descriptor shall describe an object of interoperable type and type parameters  
14 with the same characteristics as the effective argument.  
15 6 An absent actual argument in a reference to an interoperable procedure is indicated by a corresponding formal  
16 parameter with the value NULL.

## 5.3 Interaction with the DEALLOCATE statement

- 18 1 The DEALLOCATE statement shall treat a pointer whose target was allocated using CFI\_allocate in exactly the  
19 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 EXAMPLE_send_old
end interface EXAMPLE_send_abstract
```

```

1      end subroutine
2      subroutine EXAMPLE_send_new (buffer) bind(c,name="EXAMPLE_send_abstract_new")
3          type(*), dimension(..), contiguous :: buffer
4          ! Passed by descriptor including the shape and type
5      end subroutine
6  end interface
7
8  real :: x(100), y(10,10)
9
10     ! These will invoke EXAMPLE_send_old
11     call EXAMPLE_send_abstract(x,c_sizeof(x)) ! Passed by address
12     call EXAMPLE_send_abstract(y,c_sizeof(y)) ! Sequence association
13     call EXAMPLE_send_abstract(y(:,1),size(y,dim=1)*c_sizeof(y(1,1))) ! Pass first column of y
14     call EXAMPLE_send_abstract(y(1,5),size(y,dim=1)*c_sizeof(y(1,1))) ! Pass fifth column of y
15
16     ! These will invoke EXAMPLE_send_new
17     call EXAMPLE_send_abstract(x) ! Pass a rank-1 descriptor
18     call EXAMPLE_send_abstract(y) ! Pass a rank-2 descriptor
19     call EXAMPLE_send_abstract(y(:,1)) ! Passed by descriptor without copy
20     call EXAMPLE_send_abstract(y(1,5)) ! Pass a rank-0 descriptor

```

### 21 A.1.3 Casting TYPE (\*) in Fortran

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

- 23 1 It is possible to “cast” a TYPE (\*) object to a usable type, exactly as is done for void \* objects in C. For example,  
 24 this code fragment casts a block of memory to be used as an integer array.

```

25     subroutine process(block, nbytes)
26         type(*), target :: block(*)
27         integer, intent(in) :: nbytes ! Number of bytes in block(*)
28
29         integer :: nelems
30         integer, pointer :: usable(:)
31
32         nelems=nbytes/(bit_size(usable)/8)
33         call c_f_pointer (c_loc(block), usable, [nelems] )
34         usable=0 ! Instead of the disallowed block=0
35     end subroutine

```

### 36 A.1.4 Simplifying interfaces for arbitrary rank procedures

#### 37 Example of assumed-rank usage in Fortran

- 38 1 Assumed-rank variables are not restricted to be assumed-type. For example, many of the IEEE intrinsic proced-  
 39 ures in Clause 14 of ISO/IEC 1539-1:2010 could be written using an assumed-rank dummy argument instead of  
 40 writing 16 separate specific routines, one for each possible rank.
- 41 2 An example of an assumed-rank dummy argument for the specific procedures for the IEEE\_SUPPORT\_DIVIDE  
 42 function.

```

43     interface ieee_support_divide
44         module procedure ieee_support_divide_noarg
45         module procedure ieee_support_divide_onearg_r4
46         module procedure ieee_support_divide_onearg_r8
47     end interface ieee_support_divide

```

```

1
2     ...
3
4     logical function ieee_support_divide_noarg ()
5         ieee_support_divide_noarg = .true.
6     end function ieee_support_divide_noarg
7
8     logical function ieee_support_divide_onearg_r4 (x)
9         real(4),dimension(..) :: x
10        ieee_support_divide_onearg_r4 = .true.
11    end function ieee_support_divide_onearg_r4
12
13    logical function ieee_support_divide_onearg_r8 (x)
14        real(8),dimension(..) :: x
15        ieee_support_divide_onearg_r8 = .true.
16    end function ieee_support_divide_onearg_r8

```

## 17 A.2 Clause 5 notes

18 1 The example shown below calculates the product of individual elements of arrays A and B and returns the result  
19 in array C. The Fortran interface of `elemental_mult` will accept arguments of any type and rank. However, the  
20 C function will return an error code if any argument is not a two-dimensional `int` array. Note that the arguments  
21 are permitted to be array sections, so the C function does not assume that any argument is contiguous.

22 2 The Fortran interface is:

```

23
24     interface
25         function elemental_mult(A, B, C) bind(C,name="elemental_mult_c"), result(err)
26             use,intrinsic :: iso_c_binding
27             integer(c_int) :: err
28             type(*), dimension(..) :: A, B, C
29         end function elemental_mult
30     end interface
31

```

32 3 The definition of the C function is:

```

33
34     #include "ISO_Fortran_binding.h"
35
36     int elemental_mult_c(CFI_cdesc_t * a_desc,
37                         CFI_cdesc_t * b_desc, CFI_cdesc_t * c_desc) {
38         size_t i, j, ni, nj;
39
40         int err = 1; /* this error code represents all errors */
41
42         char * a_col = (char*) a_desc->base_addr;
43         char * b_col = (char*) b_desc->base_addr;
44         char * c_col = (char*) c_desc->base_addr;
45         char *a_elt, *b_elt, *c_elt;
46
47         /* only support integers */
48         if (a_desc->type != CFI_type_int || b_desc->type != CFI_type_int ||
49             c_desc->type != CFI_type_int) {

```

```
1     return err;
2 }
3
4 /* only support two dimensions */
5 if (a_desc->rank != 2 || b_desc->rank != 2 || c_desc->rank != 2) {
6     return err;
7 }
8
9 ni = a_desc->dim[0].extent;
10 nj = a_desc->dim[1].extent;
11
12 /* ensure the shapes conform */
13 if (ni != b_desc->dim[0].extent || ni != c_desc->dim[0].extent) return err;
14 if (nj != b_desc->dim[1].extent || nj != c_desc->dim[1].extent) return err;
15
16 /* multiply the elements of the two arrays */
17 for (j = 0; j < nj; j++) {
18     a_elt = a_col;
19     b_elt = b_col;
20     c_elt = c_col;
21     for (i = 0; i < ni; i++) {
22         *(int*)a_elt = *(int*)b_elt * *(int*)c_elt;
23         a_elt += a_desc->dim[0].sm;
24         b_elt += b_desc->dim[0].sm;
25         c_elt += c_desc->dim[0].sm;
26     }
27     a_col += a_desc->dim[1].sm;
28     b_col += b_desc->dim[1].sm;
29     c_col += c_desc->dim[1].sm;
30 }
31 return 0;
32 }
33
```