Date: 7 February 1997
To: X3J3
From: Van Snyder
Subject: Alternative to N1237 and 97-108

This report proposes revisions to ISO/IEC JTC1/SC22/WG5 N1237, also known as X3J3/97-108. Sections or sub-sections having the same title as sections or sub-sections in N1237 are proposed replacements.

N1237 offers dangerous additional functionality, in addition to providing for interoperability of Fortran and C. In particular, POINTER is an attribute in Fortran instead of a type, and the only operation defined is pointer assignment (dereferencing is automatic). This allows optimizations that are prohibited to C compilers by the possibility of undisciplined use of pointers. Offering C_ADDRESS, C_INCREMENT and C_DEREFERENCE undermines the possibility to incorporate these optimizations into Fortran.

One could provide interoperability functionality without introducing new keywords. Fortran has developed a tradition to annotate attributes. This report proposes to annotate EXTERNAL and POINTER attributes, and allow to attach them into contexts where they are presently prohibited.

The mechanism proposed in N1237 by which varying length C argument lists are passed to C functions requires that the Fortran processor be aware of the properties of a derived type, TYPE(C_VA_LIST), and transform values of that type is specific ways. This proposal removes that requirement.

In addition, this proposal extends the functionality of the previous proposal in a "safe" direction by defining how CHARACTER variables of size other than one, Fortran pointers, and assumed-shape arrays can be used as actual arguments to C procedures.

The revisions proposed herein have no impact on existing programs.

This report does not propose any new intrinsic procedures.

1 General

This report does not propose to change the corresponding section of N1237.

2 Rationale

This report does not propose to change the corresponding section of N1237.

3 Technical Specifications

This report does not propose to change the introduction to the corresponding section of N1237.

3.1 Intrinsic modules and C standard headers

This report does not propose to change the corresponding section of N1237.

3.2 Parameterization of the EXTERNAL and POINTER attributes

This section is proposed to replace section 3.2, **The BIND attribute**, of N1237.

The Fortran standard does not specify the mechanisms by which programs are transformed for use on computing systems (1.4). Additionally, a reference in a Fortran program to a procedure defined by means other than Fortran is normally made as though it were defined by an external subprogram (12.5.3).

This Technical Report defines parameters of EXTERNAL and POINTER attributes, that may be employed to adapt the behavior of the Fortran processor to the behavior of another processor, possibly for another language, in a portable way. Parameterization of EXTERNAL and POINTER may be used in all places where it is necessary to inform the Fortran processor that a change of processor dependent or language dependent conventions is required. This section specifies the general form of the parameters of EXTERNAL and POINTER.

R503 *attr-spec* **is** EXTERNAL [([LANG =] ■
■ *lang-keyword* ■
■ [, [NAME =] *name-string*] ■
■ [, PRAGMA = ■
■ *pragma-string*])]
or POINTER [([LANG =] ■
■ *lang-keyword* ■
■ [, PRAGMA = ■
■ *pragma-string*])]

R1601 *lang-keyword* **is** FORTRAN
or C

R1602 *name-string* **is** *scalar-default-char-init-expr*

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

Constraint: If *name-string* is present and *lang-keyword* is FORTRAN, the value of *name-string* shall be a valid Fortran name.
Constraint: If *name-string* is present and *lang-keyword* is C, the value of *name-string* shall be a valid C *external* name.

The processor shall support at least those *lang-keyword*s listed in R1601; support of other *lang-keyword*s is processor dependent. The processor shall report the use of unsupported *lang-keyword*s.

The term "EXTERNAL(*lang-keyword*) attribute" or "POINTER(*lang-keyword*) attribute" denotes the EXTERNAL or POINTER attribute with the given *lang-keyword* parameter; neither term implies the presence or absence of a *name-string* or *pragma-string*.

EXTERNAL(FORTRAN) and POINTER(FORTRAN) specify the default behavior of the Fortran processor. The behavior for *lang-keyword* C is defined in this Technical Report. The behavior for *lang-keyword*s other than those listed in R1601 is processor dependent. The interpretation of *pragma-string*s is processor dependent.

Note 3.4

Selecting the programming language C with the *lang-keyword* alone does not specify the implementation-defined and implementation-dependent behavior of the C processor, and specifying such information would in fact make the program unportable. The Fortran processor should be accompanied by documentation that states which C processor's conventions are followed.

If multiple C processors are supported, selection of a specific C processor should occur separately from the Fortran program (e.g. by command-line arguments to the Fortran processor) rather than by introducing additional *lang-keyword*s for nondefault C processors.

Note 3.5

Although names of C entities are normally case-sensitive, a C processor may ignore the distinctions of alphabetic case of *external* names. This limitation is implementation-defined.

A strictly conforming C program shall not rely on the implementation-defined behavior, and a Fortran processor that does not support lower case letters still conforms to this Technical Report because it will be able to generate bindings to all external names that are allowed in a strictly conforming C program.

Note 3.6

C++ has a *linkage-specification* (7.5) that is similar in usage to parameters of EXTERNAL. The processor must support the values "C" and "C++". C++ does not, however, need a NAME=

clause because C and C++ have the same (case sensitive) rules for names, and the C/C++ pre-processor processes `#pragma`s.

An EXTERNAL attribute with parameters may appear as part of an INTERFACE statement, and as a *attr-spec* in a *type-declaration-stmt*. EXTERNAL and POINTER attributes with parameters may appear as an *attr-spec* for a data object. The EXTERNAL statement with attributes may also be used to attach the EXTERNAL(C) attribute to a data object.

Section 3.3.4 describes the use of parameters of the EXTERNAL attribute to map C structure types to Fortran. Section 3.4 shows how to use parameters of the EXTERNAL and POINTER attributes in explicit procedure interfaces to map C function prototypes to Fortran. Section 3.5 explains how to use parameters of the EXTERNAL and POINTER attributes to bind to C data objects with external linkage.

3.3 Datatype mapping

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

The incomplete type `void` is not supported. The types "pointer to void" and "function returning void" are supported (see sections 3.3.7 and 3.4). *Note 3.7*

Both languages define object types that are intrinsically available. These are called *intrinsic* types in Fortran, and *basic* types in C. *Derived* types can be constructed from them.

The C enumerated types declared with the type specifier `enum` are not specified to be *basic* types in the C standard (they are *integral* types, but not *integer* types), but neither are they specified to be derived types. Section 3.3.3 addresses C enumerated types. *Note 3.8*

Section 3.3.1 specifies a complete mapping of C *basic* types to Fortran types. Access to the corresponding environmental limits is specified in section 3.3.2. The remaining sections deal with some of the *derived* types of

C. The mechanisms defined in this Technical Report do not specify mappings for all possible C data types: Derived type generation in C can be recursively applied. Not all resulting types have a general approximation in Fortran types.

3.3.1 Matching C basic types with Fortran intrinsic types

This report does not propose to change the corresponding section of N1237.

3.3.2 Numerical limits of the C environment

This report does not propose to change the corresponding section of N1237.

3.3.3 C enumerated types

This report does not propose to change the corresponding section of N1237.

3.3.4 C structure types

A *structure* type in C with member objects that all have a type for which this Technical Report establishes a corresponding Fortran type can be mapped to a Fortran derived type by using a derived type definition. To ensure that the memory layout of the Fortran derived type matches the memory layout of the C `struct`, the EXTERNAL(C) attribute shall be specified in the *derived-type-def*. The EXTERNAL(FORTRAN) attribute, or the EXTERNAL attribute alone, may not be used in a *derived-type-def*.

Note 3.15

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

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

Note 3.16

For example, the C structure type declaration

```
struct point {
 int x;
```

```
 int y;
};
```

can be mapped to Fortran in the following way:

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

The Fortran *type-name* need not correspond to the tag of the C `struct` because both are local to their respective scoping units. Consequently, a NAME= clause in an EXTERNAL(C) specification within a derived type definition is not allowed. Similarly, the Fortran member objects need not have the same names as the C structure members.

Nested structures can be treated in a natural way. A Fortran derived type with the EXTERNAL(C) attribute may have components that are of derived types, so long as those derived types also have the EXTERNAL(C) attribute.

The C structure type declaration *Note 3.16.1*

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

can be mapped to the Fortran type declaration

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

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

POINTER(C) may be used for a *component-attr-spec* (see section 3.3.7). The POINTER (FORTRAN) or POINTER attribute may not be used for a *component-attr-spec* in a derived type definition that has the EXTERNAL(C) attribute.

For example, the C structure *Note 3.18*

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

might be used to represent a binary tree with two data fields and two pointers to other nodes of the tree. It can be mapped to Fortran by the derived type definition

```
TYPE, EXTERNAL(C) :: tnode
  CHARACTER(KIND=c_char), POINTER(C) :: word(:)
  INTEGER(c_int)                     :: count
  TYPE(tnode), POINTER(C)            :: left, right
END TYPE tnode
```

C `structs` that specify *bit-fields* cannot be mapped to Fortran by any mechanism specified in this Technical Report.

3.3.5 C union types

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

C objects of `union` type may be accessed by specifying separate EXTERNAL(C) derived types for each union member (as if that member were the only member of a `struct`), and using TRANSFER or EQUIVALENCE to convert between these types. The derived type used in the actual C binding must be "wide" enough to hold the "widest" member of the `union` and at the same time fulfill the most restrictive alignment requirements of all `union` members. In *Note 3.19*

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

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

```
TYPE, EXTERNAL(C) :: u_name
  CHARACTER(c_char)) :: name(13)
END TYPE u_name
TYPE, EXTERNAL(C) :: u_val
  REAL(c_dbl) :: val
END TYPE u_val
```

are suitable definitions for the `union` members, both are probably insufficient to bind to the `union` object `u_tag`. It may be necessary to employ an additional derived type

```
TYPE, EXTERNAL(C) :: u_fits_all
  REAL(c_dbl)       :: alignment
  CHARACTER(c_char) :: fill_up(5)
END TYPE u_fits_all
```

to fulfill both the size and alignment requirements.

3.3.6 C array types

This report does not propose to change the corresponding section of N1237.

3.3.7 C pointer types

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

A C pointer to a C basic or derived type can be mapped to Fortran by including the POINTER(C) attribute in the declaration of objects of the Fortran type to which the corresponding C types are mapped. The POINTER(C) attribute may only be used with scalars, *explicit-shape* arrays or *assumed-shape* arrays of rank one. Neither whole-array operations, nor the UBOUND, LBOUND or SIZE intrinsic functions, may be applied to *assumed-shape* arrays that enjoy the POINTER(C) attribute.

A C pointer to a `char` is usually a pointer to an array of indefinite length. The end of the array is usually denoted by the character having numeric representation zero, denoted in C by `NULL`. This character has the representation CHAR(0) in Fortran. In Fortran, arrays of characters have a specified size, and are not terminated by CHAR(0). Mapping a CHARACTER(1) array to a "pointer to `char`" by using the POINTER(C) attribute does not automatically insert CHAR(0) at the end of a CHARACTER(1) array. One should allow an extra array element, and then insert CHAR(0) after the last character to be considered part of the argument. By using EQUIVALENCE between a CHARACTER(1) array and a character scalar having the same length as the array size, one can use LEN_TRIM to determine the position of the last non-blank character in an array.

Note The CHARACTER type in Fortran needs regularization.

A C pointer to a C pointer may be mapped to a Fortran object having the POINTER(C) attribute, of a type that is a Fortran derived type with the EXTERNAL(C) attribute, having a component annotated with the POINTER(C) attribute.

Pointer assignment is extended to convert between POINTER(C) and POINTER(FORTRAN). A POINTER(C) pointer that accesses an array may be assigned by pointer assignment to a POINTER(FORTRAN) pointer only if both pointers have *explicit-shape*s that are the same. Any POINTER (C) object may be assigned the value NULL() by pointer assignment.

A POINTER(C) object may be assigned to point to a TARGET that is a scalar or an array of explicit shape. If the type is a character type the length must be one.

The NULLIFY statement is extended to nullify POINTER(C) pointers.

The ASSOCIATED intrinsic function is extended to allow arguments with the POINTER(C) attribute.

Note 3.28 POSIX.1 specifies that the values of environment variables are accessible through an external variable declared

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

This is a pointer to a null-terminated array of pointers to null-terminated character strings.

A rank one assumed-shape array with the POINTER(C) and EXTERNAL(C) attributes (see also section 3.5) may be used to access environment variables in POSIX.1 compliant systems:

```
TYPE, EXTERNAL(C) :: env
  CHARACTER(,KIND=c_char), POINTER(C) :: envc(:)
END TYPE env
TYPE(env), POINTER(C), EXTERNAL(C,"environ") :: &
         & envs(1:)
```

The end of the array is denoted by an element of `envs`, say `envs(j)`, for which `ASSOCIATED(envs(j)%envc)` is .false. Individual characters of an environment string may be accessed by using `envs(j)%envc(k)`. The end of an environment string is denoted by `envs(j)%envc(k) == CHAR(0,KIND=c_char)`.

3.3.8 C function types

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

3.3.9 Handling of C `typedef` names

In C, a declaration whose *storage-class-specifier* is `typedef` can be used to define identifiers that name types. These `typedef`-names do not introduce new types – only synonyms for types that could have been specified in another way. They may be used as *type-specifiers*. `typedef`-names may be simulated in Fortran by constructing a Fortran derived type having one component of the type of the name for which a synonym is to be constructed. In Fortran, this introduces a new type.

Note 3.30

The Xlib application programming interface includes a type `Window`. It is defined in <`X11/Xlib.h`>, by the following `typedef`s:

```
typedef unsigned long XID;
typedef XID Window;
```

Rather than directly using an INTEGER(c_ulong) *type-spec* in the application program, these details may be at least partly hidden by declaring the following types:

```
TYPE XID
  INTEGER(c_ulong) :: XID_value
```

```
END TYPE XID
TYPE Window
  TYPE(XID) :: Window_ID
END TYPE Window
```

3.3.10 Type qualifiers

This report does not propose to change the corresponding section of N1237.

3.3.11 Storage class specifiers

This report does not propose to change the corresponding section of N1237.

3.4 Procedure calling conventions

This section defines mechanisms to instruct the Fortran processor to follow the calling conventions of the processor designated by the *lang-keyword* C when an external procedure defined by means of C is referenced. An explicit interface for that procedure shall be accessible in all scoping units containing a procedure reference that must use these modified calling conventions. The *function-stmt* or *subroutine-stmt* in the corresponding *interface-body* shall contain an EXTERNAL(C) attribute.

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

3.4.1 Procedure interface using the EXTERNAL(C) attribute

This Technical Report does not support "K&R style" C function declarations that do not contain a *parameter-type-list*. An explicit interface with the EXTERNAL(C) attribute corresponds to a C function prototype that includes a *parameter-type-list*.

Note 3.32

This restriction is imposed to avoid the complicated rules of C for mixed "K&R" and "ANSI" style function declarations and definitions.

When binding to a C function whose definition is specified using "K&R" style, an explicit interface must be specified that corresponds to the C function prototype that would result from

C's *default argument promotion* of the "K&R" style C function definition.

Prefix specifications The prefix of an *interface-stmt* that introduces an interface body that corresponds fo a C function prototype shall contain a parameterized EXTERNAL attribute with the *lang-keyword* C. Specifying RECURSIVE for a procedure in the interface body is allowed and has no effect. Since a pure procedure must be a subprogram (that is, defined by means of Fortran), neither the PURE nor ELEMENTAL prefix shall be present in the interface body.

If the Fortran entity is a dummy procedure, no *name-string* shall be present. If the Fortran entity is an external procedure and no *name-string* is present, the *function-name* or *subroutine-name* is used to generate an external entry for the procedure, using the Fortran processor's conventions. This implies ignoring alphabetic case for the name. If the Fortran entity is an external procedure and a *name-string* is specified, the external entry is generated using the C processor's conventions, as if the value of the *name-string* were a C external name.

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

Note 3.33

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

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

- a basic type or a structure type, the function result variable shall have the type specified for that C type in section 3.3 of this Technical Report.
- an enumeration type or union type, this is not directly supported.
- a pointer type, the type of the function result variable shall be of an arbitrary derived type with a single component having the POINTER(C) attribute, and of the same as the type from which the C pointer type is derived.

The PASS_BY atrribute This report proposes to delete the corresponding section of N1237.

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

In an interface block with the EXTERNAL(C) attribute, the following interpretations apply:

- If the dummy argument is a dummy procedure it must have the EXTERNAL(C) attribute; a pointer to the procedure is passed to the C function.

- If a dummy argument has neither the DIMENSION nor POINTER attribute, and it is not of a Fortran character type with length other than one, then the corresponding actual argument is passed "by value" to the C function. The corresponding parameter of the C function shall be a basic type or a structure type "T" that corresponds to the dummy argument according to the mapping specified in section 3.3. The dummy argument must be specified to have INTENT(IN).

- If a dummy argument is of COMPLEX type it must have a KIND type parameter for a C floating type; a C pointer to the actual argument is passed to the C function. The corresponding parameter of the C function should have type "array of T" or "pointer to T", where T is the C floating type corresponding to the KIND of the dummy argument. The dummy argument may have any INTENT, or unspecified INTENT.

- If a dummy argument has either a DIMENSION attribute specifying explicit shape or assumed size, or the POINTER(C) attribute, and it is not of a Fortran character type with length other than one, then a C pointer to the actual argument is passed to the C function. The corresponding parameter of the C function should have type "array of T"or "pointer to T" where "T" corresponds to the dummy argument type according to the mapping specified in section 3.3. The dummy argument may have any INTENT, or unspecified INTENT.

Note 3.36

All information about the shape of the argument is lost when a Fortran actual argument is passed to a C dummy argument of type "pointer to T" or "array of T".

- If a dummy argument is of a Fortran character type with length other than one (including "*"), or has either a DIMENSION attribute specifying assumed shape, or a POINTER(FORTRAN) attribute, then a pointer to a descriptor is passed to the C function. The dummy argument may have any INTENT, or unspecified INTENT. The content and layout of the descriptor may be different from the descriptor used for arguments to Fortran subprograms, and are not specified by this Technical Report. The C type of the descriptor shall be "pointer to FortranArgument". A "C header" `isoftn.h` and corresponding functions shall be provided:

```
/* type declaration for FortranArgument here */
void* FortranAddress(*FortranArgument); /* cast it */
long FortranRank(*FortranArgument);
long FortranLbound(*FortranArgument,int);
long FortranUbound(*FortranArgument,int);
long FortranElementSize(*FortranArgument);
  /* same as LEN for CHARACTER arguments */
```

If the type of the C function parameter is

- a basic type or a structure type, the dummy argument shall be scalar and have the type specified for the corresponding C type in section 3.3 of this Technical Report.
- an enumeration type or union type, this is not directly supported.
- a type "function returning T", this type is adapted by the C processor to the type "pointer to function returning T" and the rules for that type shall be followed.
- a type "array of T", this type is adapted by the C processor to the type "pointer to T".
- a type "pointer to T", the Fortran interface shall either use a declaration corresponding to "pointer to T", or shall declare the type corresponding to the C type T, and a DIMENSION attribute corresponding

to the C array declarator, specifying explicit shape or assumed size. The rules further depend on the type from which the pointer is derived. If the type "T" is

- the incomplete type `void` then the Fortran dummy argument can have any type; the TRANSFER function or EQUIVALENCE may be used to convert to the correct type.
- `char`, the dummy argument shall be a scalar of type CHARACTER (LEN=1, KIND=c_char), or an array thereof having explicit shape or assumed size.
- an integer type, floating type, or structure type, the dummy argument shall be a scalar that has the type corresponding to the C type "T", or an array thereof having explicit shape or assumed size.
- an enumeration type or union type, this is not directly supported.
- a type "function returning T1", a dummy procedure shall be declared. The dummy procedure shall have have explicit interface. The *interface-block* shall have the EXTERNAL(C) attribute. The explicit interface of the dummy procedure shall correspond to the function prototype of the C function parameter, as specified in this section.

No other C pointer types are directly supported.

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

- Neither the OPTIONAL nor TARGET attribute shall be specified.
- A dummy argument shall not have the type LOGICAL.
- If a dummy argument or function result has derived type, that type shall have the EXTERNAL(C) attribute.
- A procedure with EXTERNAL(C) interface may be associated as an actual argument only to a dummy procedure with EXTERNAL(C) interface.

3.4.2 Procedure reference for EXTERNAL(C) procedure

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

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

Actual arguments associated with dummy data objects If the dummy argument is an array, the actual argument shall be an array of the same type and type parameters. It shall be a variable.

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

If the dummy argument is a scalar of intrinsic type or of a derived type that has the EXTERNAL(C) attribute, and does not have the POINTER attribute, the actual argument shall be a scalar expression or a scalar data object. It shall have a type and type parameters for which assignment to a variable of the type and type parameters of the dummy argument is defined by ISO 1539-1 or this Technical Report. The C function parameter is assigned the value of the actual parameter, converted as if by such assignment to the type and type parameters of the dummy argument, using defined assignment if necessary.

> The conversion (as if by assignment) of the actual argument to the type of the dummy argument corresponds to the conversion that occurs within C when referencing a C function with a visible prototype declaration. *Note 3.40*

> Because a copy of the (possibly converted) value of the actual argument is passed to the C function, the C function may not modify the actual argument. *Note 3.41*

If the dummy argument is a scalar of intrinsic type, or of a derived type that has the EXTERNAL(C) attribute, and has the POINTER(C) attribute, or is an array of explicit shape or assumed size, it is assumed

to have INTENT(INOUT) if no INTENT is specified. The corresponding actual argument shall be a variable whose type and type parameters are the same as those of the dummy argument. The actual argument may or not have the POINTER(C) attribute. If the dummy argument and actual argument both have the POINTER(C) attribute the actual argument may be nullified, in which case a C null pointer is passed to the C function.

Note 3.42

> Because a C pointer is passed, the C function may modify the actual argument. If the actual argument denotes a C null pointer, and the C function dereferences the corresponding parameter, the behavior is undefined.

If a dummy argument is of a Fortran character type with length other than one (including "*"), or has either a DIMENSION attribute specifying assumed shape, or a POINTER(FORTRAN) attribute, then the actual argument must be a variable that has the same type, type parameters and rank as the dummy argument. A pointer to a descriptor is passed to the C function, as described under "Mapping C function parameters to dummy arguments" above.

Actual arguments associated to dummy procedures If a dummy argument is a dummy procedure, it shall have an explicit interface with the EXTERNAL(C) prefix. The corresponding actual argument shall be the specific name of an external or dummy procedure, that has an explicit interface with the EXTERNAL(C) attribute.

The characteristics of the associated actual argument shall be identical to the characteristics of the dummy procedure.

Function results of C pointer types A C function result that is of a pointer type must have the POINTER(C) attribute. As with all other C pointer types, a function result of pointer type must be a scalar, or an array of explicit shape or assumed size, or a rank one array of assumed shape. The underlying type may not be a character type having length other than one. The value may be used in the same ways that other C values may be used. The pointer may be assigned using pointer assignment, subject to the same restrictions as any other object having the POINTER(C) attribute.

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

C functions may be called with a variable number of arguments of varying type. The argument list in the function prototype of such a function must

contain one or more parameters followed by an *ellipsis* (...). The called function may access the varying number of actual arguments by facilities defined in the standard header <stdarg.h>. Fortran does not directly support this kind of procedure interface.

> The Fortran concept of OPTIONAL arguments is less flexible. It requires the specification of all possible combinations of arguments at compile time, including their types. *Note 3.44*

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

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

  > The type TYPE(C_VA_LIST) is a translation of the C type `va_list` defined in <stdarg.h>. *Note 3.45*

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

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

- An extension of OPERATOR(//) is provided for scalar operands x_1 of type TYPE(C_VA_LIST). x_2 may be of any type corresponding to a C data type, as specified in section 3.3 of this Technical Report, except a CHARACTER with length other than one, or it may be of type TYPE(C_VA_LIST). If x_2 is not a scalar, it shall be an array of explicit shape or assumed size and have the POINTER(C) attribute. The result type is TYPE(C_VA_LIST). The result value is a copy of x_1 concatenated with the value resulting from C's *default argument promotion* of x_2. The following table shows the type conversions that take place for x_2; for all other types the value of x_2 is used without conversion.

Type of x_2	Value resulting from promotion
INTEGER(c_schar)	INT(x_2, KIND=c_int)
INTEGER(c_shrt)	INT(x_2, KIND=c_int)
INTEGER(c_uchar)	INT(x_2, KIND=c_int)
INTEGER(c_ushrt)	INT(x_2, KIND=c_int)
REAL(c_flt)	REAL(x_2, KIND=c_dbl)

Parentheses used to specify the order of evaluation of the extended "//" operator have no effect on the value of the result.

If a scalar varying argument would have INTENT(OUT) or INTENT(INOUT) if used as a fixed argument, it must have the POINTER(C) attribute. *Note 3.46*

Default argument promotion takes place when constructing the varying argument list rather than at the time of argument association for the procedure reference. This is motivated by the fact that an explicit interface with the EXTERNAL(C) attribute represents a C function prototype. In this case no default argument promotion takes place. Varying arguments always suffer argument promotion, so to avoid complicated argument association rules this promotion is done when constructing the varying argument list. *Note 3.47*

Integral promotion of the C type `char` is not supported because this Technical Report does not match that type with a Fortran integer type. If required, one of its signed or unsigned variants may be used. Integral promotion of C bit fields or enumeration types in not required because this Technical Report does not match those types with any Fortran type. *Note 3.48*

The *interface-body* that specifies a Fortran interface to a C function having an *ellipsis* parameter shall specify the fixed arguments as specified in section 3.4.1 and shall specify an additional final scalar dummy argument of type TYPE(C_VA_LIST) in the position of the *ellipsis*. Intent other than INTENT(IN) shall not be specified for this argument. The POINTER attribute shall not be specified for this argument.

The actual argument associated to the argument in the position of the *ellipsis* shall be a scalar expression of type TYPE(C_VA_LIST). To indicate an empty varying argument list, its value shall be C_VA_EMPTY. Otherwise its value shall be C_VA_EMPTY concatenated by OPERATOR(//) with the

list of required arguments. C_VA_EMPTY shall be the leftmost operand of this concatenation. All other operands shall be specified from left to right in order according to their respective positions in a C function reference.

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

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

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

```
EXTERNAL(C,"fcntl") INTERFACE
  INTEGER(c_int) FUNCTION fcntl ( fildes, cmd, va )
  USE iso_c, ONLY: c_int
  USE iso_c_stdarg_h, ONLY: c_va_list
  INTEGER(c_int), INTENT(IN)  :: fildes, va
  TYPE(c_va_list), INTENT(IN) :: va
```

With this explicit interface accessible, the function references

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

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

3.5 Access to C global data objects

Note

This functionality could be removed because one can always access a C global data object by way of a C function.

This section defines mechanisms to reference global data objects that are defined in C translation units from within Fortran program units.

To access a C data object of type T with external linkage from within Fortran, a Fortran variable with Fortran type corresponding to T (as specified in section 3.3 of this Technical Report) shall be declared in a module, and may then be accessed within the module and all other scoping units that contain a module reference for that module, subject to the rules for USE association.

To indicate that the storage for the Fortran variable is reserved by the C translation unit containing the definition of that C external variable, the EXTERNAL(C) attribute shall be specified with a *name-string* whose value is the identifier of the C object with the `extern` storage class. Because the data object is a global data object, the EXTERN(C) attribute for a module variable implies the SAVE attribute for that variable.

Note 3.50

For example, POSIX.1 requires the standard header <`errno.h`> to contain the declaration

```
extern int errno;
```

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

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

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

The following additional restrictions apply for module variables having the EXTERNAL(C) attribute:

- No *initialization* shall appear in the *entity-decl*.
- Neither ALLOCATABLE, PARAMETER, POINTER nor TARGET shall be specified.
- If the object has derived type, that type shall have the EXTERNAL(C) attribute.

- The object shall not have the LOGICAL type. If it has the CHARACTER type the length shall be one.

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

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

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

Note 3.51

Fortran prohibits a module variable to be *variable-name* in a *common-block-object* or as an *equivalence-object*, and module variables having the EXTERNAL(C) attribute are not exempted. However, the C `extern` variable is a global variable. Binding different module variables to the same C object effectively EQUIVALENCES them.

As with any global variable, the value of a module variable having the EXTERNAL(C) attribute may be changed by the execution of Fortran subprograms or C functions that have an external declaration of the identifier in the corresponding *name-string* in scope. The Fortran processor shall not assume that the value of such a module variable remains unchanged after a procedure reference. The value of a module variable having the EXTERNAL(C) attribute may also be changed by other means invisible to the Fortran program. The Fortran processor is not required to guard against such behavior.

Note 3.52

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

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

This section would be substantially different from the corresponding section of N1237. It has not been prepared. It will be prepared if there is interest in this alternative formulation of interoperability between Fortran and C.