Extended interoperation with C

by Reinhold Bader, Leibniz Supercomputing Centre, Garching, Germany

Standardized support of C language interface use, as defined in Fortran 2003, is nowadays implemented in most Fortran compilers. However, there are significant limitations to this interoperability, particularly with respect to Fortran objects for which no C analog exists; furthermore, the language semantics as defined up to Fortran 2008 do not allow for a conforming implementation of the Message Passing Interface (MPI) which presently is the most prevalent parallelization approach on large-scale HPC systems. Therefore, a decision was made by the Fortran Standards Committee to develop an ISO/IEC Technical Specification (ISO/IEC TS 29113, referred to as "TS" in the following) that significantly extends the scope of the interoperation facilities; this work was completed in autumn 2012 and published by ISO shortly afterwards. This article provides an informal overview of the new facilities, which are also targeted for integration into the next release of the Fortran standard, with additional technical corrections if necessary.

Contents

Definition of a C descriptor for Fortran objects

Fortran objects that are not directly interoperable with C are, among others, dummy objects that are assumed-shape, deferred-shape, or have deferred character length. In order to provide access to such objects from C, a type definition for a C descriptor as well as macro definitions and accessor function prototypes are defined for the companion C compiler in a header file `ISO_Fortran_binding.h` which uses and reserves the name space `CFI_`. The descriptor's type name is `CFI_cdesc_t`, and an object of this type has the following members:

`void *base_addr;`	the C address of the first data item of the described Fortran object, which may be a scalar or an array. There are circumstances where this can be a null pointer (for example if the Fortran object is an unallocated allocatable entity).
`size_t elem_len;`	the size of the Fortran object in units of bytes. If the object is an array, the size is that of a single array element.
`int version;`	this is used by the implementation to achieve some level of compatibility between different compiler releases. The header file provides a macro `CFI_VERSION` that allows doing cross-checking.
`CFI_rank_t rank;`	for an array, the value is a non-negative integer that specifies the number of dimensions of the object. For a scalar, the value is zero.
`CFI_type_t type;`	takes on integer values that correspond to either an interoperable intrinsic type (for example, the value of the macro `CFI_type_float` would be encoded here if the corresponding Fortran object were declared `REAL(c_float)`), an interoperable structure type (`CFI_type_struct`), or an unknown type (`CFI_type_other`). All type specifier macros for interoperable types evaluate to positive values, those for non-interoperable types are negative. `CFI_type_other` is also negative and has a value distinct from all other type specifiers.
`CFI_attribute_t attribute;`	this integer value is used to distinguish between objects that have the `ALLOCATABLE` attribute, the `POINTER` attribute, or neither. Correspondingly, the header defines the macros `CFI_attribute_allocatable`, `CFI_attribute_pointer` and `CFI_attribute_other` with distinct values.
`CFI_dim_t dim[];`	this flexible array member encodes the lattice structure of the described Fortran object if it is an array. The number of array elements is equal to the rank of the array; each array element contains information about the corresponding dimension of the Fortran array.

The structure type **`CFI_dim_t`** has the following members:

`CFI_index_t lower_bound;`	the lower bound of the array object in the specified dimension. For nonpointer nonallocatable arrays, the value is zero; otherwise, it is determined by the preceding allocation or pointer assignment for the Fortran object.
`CFI_index_t extent;`	the number of array elements along the specified dimension.
`CFI_index_t sm;`	the stride multiplier describes the distance between subsequent array elements in the specified dimension in units of bytes.

All three members are of an integer type that is capable of representing the result of subtracting two pointers.

An object of type **`CFI_cdesc_t`** should for the most part be considered opaque, and if possible only be used as a parameter in invocation of the accessor functions. In some cases, read access to certain members is unavoidable. The following sections provide examples on how to use the descriptor, the macros and the accessor functions.

Assumed-shape arguments, array element addressing and contiguity

Consider the following interoperable interface that specifies assumed-shape dummy arguments:

```
subroutine example_1 ( a, x ) BIND(C)
  real(c_float) :: a(:,:)
  real(c_float), contiguous :: x(:)
end subroutine
```

The C prototype that matches the above interface reads

```
#include "ISO_Fortran_binding.h"

void example_1 ( CFI_cdesc_t *a, CFI_cdesc_t *x );
```

and the **`#include`** line that is always needed in C code using the TS' facilities will be taken for granted for the rest of this article. It is possible to write the implementation of the procedure either in Fortran or in C; for the purpose of comparing the functionality it is assumed in the following that both alternatives exist. For example, the Fortran code might want to loop through the array **`a`** as follows:

```
do k=1, ubound(a, 2)
  do i=1, ubound(a, 1)
    ... = a(i, k) * ...
  end do
end do
```

In order to access these array elements in the same manner in the C function body, the following code could be used:

```
for (k = 0; k < a->dim[1].extent; k++) {
  for (i = 0; i < a->dim[0].extent; i++) {
    CFI_index_t subscripts[2] = { i, k };
/* for allocatable or pointer objects, calculation of
   subscripts would also need to evaluate the non-trivial
   lower bounds */
    ... = *((float *) CFI_address(a, subscripts)) * ...;
  }
}
```

The function **`CFI_address()`** produces the C address of an array element designated by the provided subscript; the void-typed result pointer must be suitably cast. Validity of the subscript is here assured by extracting the array extents from the **`dim`** member of the C descriptor. Since the incoming array should be of rank 2, exactly two **`dim`** entries must exist in the descriptor here. Alternatively, it is also possible to start out from the first array element

```
float *a_ptr = (float *) a->base_addr;
```

and then use regular C pointer arithmetic to process the array; this is also likely to be more efficient. However, some care is required to avoid non-conforming accesses since the underlying Fortran object may be non-contiguous (for example, because the actual argument is a strided array section): If this approach is chosen, use must also be made of the stride multipliers stored in the C descriptor. A loop nest equivalent to the above one would then read

```
float *a_aux;
for (k = 0; k < a->dim[1].extent; k++) {
  a_aux = a_ptr;
  for (i = 0; i < a->dim[0].extent; i++) {
     ... = *a_ptr * ...;
     a_ptr += a->dim[0].sm / sizeof(float);
  }
  a_ptr = a_aux + a->dim[1].sm / sizeof(float);
}
```

This second version of the loop nest would not require referencing the lower bounds even if the descriptor were that for an allocatable or pointer object.

Because there exists no C language mechanism to perform type, rank or attribute checking of descriptor-based objects at compile time, but an implementation in C may be invoked also by another C function, it is considered good practice to provide suitable run time checking, at least for debugging. Specifically for the argument **`x`** of the procedure **`example_1`**, the following guard code might be used:

```
if (x->type != CFI_type_float || x->rank != 1) {
   printf("Argument X has wrong type or rank.\n");
   exit(1);
}
if (x->attribute != CFI_attribute_other) {
   printf("Argument X has wrong attribute.\n";
   exit(1);
}
if (! CFI_is_contiguous(x)) {
   printf("Argument X is not contiguous.\n");
   exit(1);
}
```

Checking exact attribute matching is appropriate for C-to-C calls in this example, particularly since the implementation relies on the lower bounds in the descriptor for **x** being zero. Also, in a call from Fortran it is assured that a compactified copy of a non-contiguous actual argument is created because the dummy argument **x** is declared with the **CONTIGUOUS** attribute, while for an invocation of the C implementation from C this will need to be done explicitly by the caller if (as presumably the case here) the implementation relies on the contiguity of the object. The function **CFI_is_contiguous()** returns an integer value 1 if the supplied descriptor is for a contiguous object, and 0 otherwise.

Optional arguments

A Fortran interface that uses optional arguments can be declared interoperable:

```
subroutine example_2 ( q, c ) BIND(C)
  integer(c_int), optional :: q(:)
  character(c_char), optional :: c
end subroutine
```

An implementation in C determines the presence of a parameter by checking whether it is a null pointer:

```
void example_2( CFI_cdesc_t *q, char *c ) {
  if (q) {
    ... /* process descriptor q */
  }
  if (c) {
    ... /* process character(c_char) entity c */
  }
}
```

Similarly, invocation of this procedure from C must use a null pointer to denote an absent parameter:

```
example_2( NULL, &y );
```

This presupposes that optional arguments interoperate with pointer types; it is therefore not permitted to specify both the **OPTIONAL** and the **VALUE** attributes in a **BIND(C)** interface.

Creating descriptors and Fortran objects in C

From the previous examples it is clear that occasionally the programmer will need to create descriptors within C that can be used as parameters in calls to interoperable procedures. For this purpose, the **`ISO_Fortran_binding.h`** header provides two items. The first is a parameterized macro **`CFI_CDESC_T`** that provides the storage needed to hold a C descriptor. For example, the statement

```
CFI_CDESC_T(3) R3A;
```

provides storage for an untyped entity that can be cast to a descriptor for an array of at most rank 3:

```
CFI_cdesc_t *r3a = (CFI_cdesc_t *) &R3A;
```

The second is a constructor function that establishes a descriptor by initializing all or most of its members. Its prototype is

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

and it must always be used to initialize a descriptor created within C. It cannot be applied to a descriptor the address of which has the same value as a C formal parameter that corresponds to a Fortran actual argument, or a C actual argument that is presently argument associated with a Fortran dummy argument; this rule prevents corruption of descriptors that have been created or initialized by the Fortran processor. It is required that the attribute of the created descriptor **exactly** match the attribute specified for the corresponding argument in the Fortran interface. This is different from Fortran where, for example, an allocatable actual argument may be associated with an assumed-shape dummy; as a consequence it will sometimes be necessary for implementations in C to create a second descriptor for the same data object.

To illustrate one of the many usage patterns for this function, a descriptor that can be used as the actual parameter for the entity **`a`** in the **`example_1`** procedure above would require the following preparations:

```
#define DIM1 56
#define DIM2 123

CFI_CDESC_T(2) A;
CFI_CDESC_T(1) X;
CFI_cdesc_t *a = (CFI_cdesc_t *) &A;
CFI_cdesc_t *x = (CFI_cdesc_t *) &X;
CFI_index_t extents[2];

extents = { DIM1, DIM2 }; /* dimensions of rank 2 array */
float *a_ptr = (float *) malloc(DIM1*DIM2*sizeof(float));
...                        /* initialize values of *a_ptr */
CFI_establish(  a, a_ptr,
                CFI_attribute_other,
                CFI_type_float,
                0, /* elem_len is ignored here */
                2, /* value for rank may not exceed 2 */
                extents );
...                        /* create rank 1 descriptor for X */
example_1( a, x );
```

The actual memory needed for the array data is here allocated by the C library function **`malloc()`**. The element length parameter is ignored in this example because the object is of intrinsic interoperable type. If **`CFI_establish()`** creates a fully defined object, it is always contiguous.

Dynamic memory management and pointer assignment

Creation of a C descriptor for an allocatable object will usually only be needed if the Fortran programming interface provides a "factory method" for an object of interoperable type. For example, for the type definition

```
type, BIND(C) :: qbody
  real(c_float) :: mass
  real(c_float) :: position(3)
end type
```

a procedure with interface

```
SUBROUTINE qbody_factory(this, fname) BIND(C)
  TYPE(qbody), ALLOCATABLE, INTENT(OUT) :: this(:,:)
  CHARACTER(c_char, len=*), INTENT(IN) :: fname
END SUBROUTINE
```

might be provided that constructs the object from data provided in an external file. This procedure could be used from C as follows:

```
typedef struct {
  float mass;
  float position[3];
} qbody;
char fname_ptr[] = 'InFrontOfMyHouse\0';
void qbody_factory(CFI_cdesc_t *this, CFI_cdesc_t *fname);

CFI_cdesc_t *pavement =
          (CFI_cdesc_t *) malloc(sizeof(CFI_CDESC_T(2)));
CFI_cdesc_t *fname =
          (CFI_cdesc_t *) malloc(sizeof(CFI_CDESC_T(0)));
CFI_establish(  pavement, NULL, CFI_attribute_allocatable,
                CFI_type_struct,
                sizeof(qbody),
                2, NULL );
CFI_establish(  fname, fname_ptr, CFI_attribute_other,
                CFI_type_char,
                strlen(fname_ptr),   /* A char has one byte */
                0, NULL );
qbody_factory ( pavement, fname );
...             /* process pavement */
CFI_deallocate( pavement );
free(pavement); free(fname);
```

When establishing the descriptor for an allocatable entity, null pointers are supplied for the base address and the extents, since both are determined later, when the allocation is performed. Automatic deallocation upon leaving the scoping unit is not supported for descriptors created in C, so eventual invocation of **`CFI_deallocate()`** is obligatory to prevent memory leaks from occurring. However, in this example it is possible to invoke **`qbody_factory()`** repeatedly without requiring intermediate deallocations; because the dummy argument is declared **`INTENT(OUT)`**, an implementation in Fortran will, if necessary, perform the deallocation upon entry to the procedure. The example also illustrates how an assumed-length scalar character string can be produced in C.

It is also possible to implement the factory method in C, because the **`ISO_Fortran_binding.h`** header defines a prototype

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

The **`elem_len`** parameter is ignored unless the descriptor is for a deferred-length string, in which case the allocation procedure needs to overwrite the value stored in the descriptor[1]. The size of the **`lower_bounds`** and **`upper_bounds`** arguments must be at least the rank of the described object. **`CFI_allocate()`** and **`CFI_deallocate()`** can also be used on a descriptor that has the attribute **`CFI_attribute_pointer`** (but not on one with the

[1] In fact, an interoperating allocatable or pointer character entity is obliged to be deferred-length.

attribute **CFI_attribute_other**), and such a descriptor can be established in an analogous manner as for the allocatable case. Because the C processor has no information about values of default-initialized type components that would need to be set by **CFI_allocate()**, using derived types with such default initialization as allocatable or pointer dummy arguments in BIND(C) interfaces is not permitted.

Finally, it is possible to perform the analog of a pointer assignment statement

```
real(c_float), target :: t(:)
real(c_float), pointer :: p(:)
p(3:) => t
```

in C; assuming that **t** and **p** are descriptors for an assumed-shape and a pointer object respectively, the statements

```
CFI_index_t lower_bounds[1] = { 3 };
status = CFI_setpointer(p, t, lower_bounds);
```

perform the desired pointer association. The result value of **CFI_setpointer()** should be checked in order to assure that no type or rank mismatch or other erroneous use has occurred; however **CFI_setpointer()** is not fully type-safe because it is likely that differing derived types with the same element size cannot be disambiguated.

Array sections and type components

The TS supports creation of two variants of array subobjects: Array sections and arrays that reference a type component of a derived type array.

First, assume that **arr** is a descriptor for an assumed-shape rank 3 array that has default lower bounds. The following code generates a descriptor for the rank 2 array section **arr(3:,4,::2)** by using a sequence of calls to **CFI_establish()** and **CFI_section()**:

```
CFI_cdesc_t *section =
          (CFI_cdesc_t *) malloc(sizeof(CFI_CDESC_T(2)));
CFI_index_t lower_bounds[3] = { 2, 3, 0 };
CFI_index_t upper_bounds[3] =
            { arr->dim[0].extent - 1, 3, arr->dim[2].extent - 1 };
CFI_index_t strides[3] = { 1, 0, 2 };
CFI_establish(  section, NULL, CFI_attribute_other,
                arr->type, arr->elem_len, 2, NULL );
CFI_section( section, arr, lower_bounds, upper_bounds, strides );
```

The invocation of **CFI_establish()** by itself in this case does not create a fully defined object; it must here be followed by invocation of **CFI_section()**. The **lower_bounds** and **upper_bounds** parameters of **CFI_section()** have obvious meanings; in the **strides** array, a value of zero indicates a subscript, and in this case the corresponding upper and lower bound entries must have the same value.

Next, consider the **pavement** object of type **qbody** from above. An array descriptor corresponding to the subobject **pavement(:)%position(1)** can be constructed in C by using a sequence of calls to **CFI_establish()** and **CFI_select_part()**:

```
CFI_cdesc_t *pos_1 =
          (CFI_cdesc_t *) malloc(sizeof(CFI_CDESC_T(2)));
CFI_establish(  pos_1, NULL, CFI_attribute_other,
                CFI_type_float, sizeof(float), 2, NULL );
size_t displacement = offsetof(qbody, position[0]);
size_t elem_len = 0;
CFI_select_part( pos_1, pavement, displacement, elem_len );
```

Based on the displacement of the selected member (which is supplied in units of bytes), the shifted base address is calculated for the result descriptor **pos_1** that then appears as the first actual argument in the call to **CFI_select_part()**; invocation of the latter again is necessary to obtain a fully defined descriptor. The **elem_len** parameter is only relevant if the result descriptor is of a character type, and is therefore ignored in this example.

If multiple descriptors are created that reference the same data or subsets of the same data, and the original descriptor becomes invalid or the object it describes becomes deallocated or undefined, all still existing descriptors derived from the original one will become invalid.

Assumed-rank entities

In an explicit interface, a Fortran dummy argument can be declared with assumed rank:

```
subroutine any_rank(a, ...)
  real, dimension(..) :: a
  ...
end subroutine
```

The matching actual argument in an invocation of the procedure can be an array of any rank, a scalar, or even an assumed-size array; this removes the need to define a generic and write large numbers of specific interfaces if all ranks must be supported for a procedure argument. However, apart from using inquiry intrinsics on such an object, nothing can be done with it within Fortran; in particular the data stored within the object cannot be accessed, and it is not possible to change the allocation status of the object if it is **ALLOCATABLE** or its association status if it has the **POINTER** attribute.

An assumed-rank dummy argument that appears in a **BIND(C)** interface is always mapped to its counterpart in the C prototype as a pointer to a C descriptor. On the level of static source inspection, there is no way to distinguish a descriptor for an assumed-rank entity from that for an assumed-shape or deferred-shape one. The C function body will need to be able to deal with all possible variants of incoming objects:

- if the actual argument is not contiguous and full generality is required, the C code might need to process a 15-fold nested loop, corresponding to the maximum rank supported by the Fortran standard. This can in some cases be avoided (at the cost of compactification overhead) by declaring the Fortran dummy argument **CONTIGUOUS** and adding a corresponding check in the C code. In **ISO_Fortran_binding.h**, the macro **CFI_MAX_RANK** specifies the maximum array rank supported by the implementation;

- if the actual argument is a scalar, the value for the rank stored in the descriptor will be zero; in any case – even if the object has the **ALLOCATABLE** or **POINTER** attribute – the value of the rank is not changeable.
- if the actual argument is an assumed size array, it is contiguous but its actual size cannot be inferred from information stored in the descriptor[2]: For the final dimension, the **extent** member of **dim** then has the value -1; In this case, the recommended programming practice is that either the interface will provide the actual size via a separate argument, or the C code should refuse processing of such an entity.

A new Fortran inquiry intrinsic **RANK** is supplied that allows identifying the actual rank of an assumed-rank object; it returns zero if it's effective argument is a scalar. The Fortran inquiry intrinsics **SHAPE**, **SIZE** and **UBOUND** have been extended to deal with the case of an assumed-size or scalar actual argument; specifically, **SIZE(A,DIM=RANK(A))** will return -1 for the assumed-size case, and **UBOUND(A,DIM=RANK(A))** will return **LBOUND(A, DIM=RANK(A))-2**. For scalar actual arguments, **SIZE** without a **DIM** argument will return 1 and invocation of the intrinsics **SIZE**, **LBOUND** and **UBOUND** with a **DIM** argument is not permitted because **DIM** must be positive and the functions return zero-sized arrays.

Assumed-type entities

A Fortran dummy argument can be declared to be of assumed type; the syntax used for such a declaration is **TYPE(*)** and the dummy argument is then considered to be unlimited polymorphic. Therefore, the interface must be explicit, and it is permitted to use a corresponding actual argument of any type (including assumed-type) in an invocation of the procedure. An assumed-type entity is not permitted to have the **ALLOCATABLE**, **CODIMENSION**, **INTENT(OUT)**, **POINTER** or **VALUE** attributes. Apart from appearing as an actual argument associated with an assumed-type dummy, or as an argument to the intrinsic functions **IS_CONTIGUOUS**, **LBOUND**, **PRESENT**, **RANK**, **SHAPE**, **SIZE**, **UBOUND**, and the intrinsic module procedure **C_LOC**, nothing can be done with such an object within Fortran.

Interoperation with C can be done in two ways, assuming the interface has the **BIND(C)** attribute:

1. The assumed-type entity is declared to be a scalar or an assumed-size array. In this case the corresponding parameter in the C prototype is a pointer to void;
2. The assumed-type entity is declared to be of assumed-shape or assumed-rank. In this case the corresponding parameter in the C prototype is a pointer of type **CFI_cdesc_t**.

The two cases can be illustrated by the well-known example of performing a blocking point-to-point send through the MPI interface. The prototype for the C call in the MPI standard is

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

[2] Some necessary restrictions on using descriptors for assumed-size entities as arguments to the accessor functions as well as in Fortran procedure invocations are presently missing in the TS.

Assuming that **`MPI_Datatype`** and **`MPI_Comm`** are interoperable struct types[3], one could write the following Fortran interface for this:

```
INTEGER(c_int) FUNCTION MPI_sendC( buf, count, datatype, dest, &
        tag, comm ) BIND(C, name="MPI_send")
  TYPE(*), DIMENSION(*), INTENT(IN) :: buf
  INTEGER(c_int), VALUE :: count, dest, tag
  TYPE(MPI_Datatype), VALUE :: datatype
  TYPE(MPI_Comm), VALUE :: comm
END FUNCTION MPI_sendC
```

This illustrates the first case above, where no further information about the argument **`buf`** is available to the programmer. Here are some examples of how to invoke the procedure from Fortran:

```
REAL(c_float) :: x(100)
INTEGER(c_int) :: y(10,10)
REAL(c_double) :: z
INTEGER(c_int) :: status, dest, tag

! Establish MPI_COMM_WORLD:
... MPI_Init(...)
! assign values to x, y, z, dest:
...
! send values in x, y, and z using MPI_sendC:
status = MPI_sendC(x, size(x,KIND=c_int), MPI_FLOAT, &
         dest, tag, MPI_COMM_WORLD)
status = MPI_sendC(y, size(y,KIND=c_int), MPI_INT, &
         dest, tag, MPI_COMM_WORLD)
status = MPI_sendC(z, 1_c_int, MPI_DOUBLE, &
         dest, tag, MPI_COMM_WORLD)
```

In these invocations, **`x`** and **`y`** are passed by address, and for **`y`** the sequence association rules allow this. The scalar **`z`** is treated as if it were a sequence associated array of size one[4].

```
status = MPI_sendC(y, size(y(:,1)), MPI_INT, &
         dest, tag, MPI_COMM_WORLD )
```

passes the first column of **`y`** (again by address), and

```
status = MPI_sendC(y(1,5), size(y(:,5:6)), MPI_INT, &
         dest, tag, MPI_COMM_WORLD)
```

passes the fifth and sixth columns of **`y`** using the sequence association rules. The above invocations always use contiguous objects for the **`buf`** argument; if this were not the case, the Fortran processor would be obliged to create a contiguous temporary array.

---

[3] Strictly, an explicit conversion between C and Fortran MPI handles is required. This will be disregarded in the context of this discussion.

[4] Allowing a scalar actual argument to match a **`type(*), dimension(*)`** dummy argument was added as a correction after the TS was published. Before that, it would have been necessary to write an array expression **`(/z/)`** in the invocation of the **`MPI_sendC()`** routine.

The Fortran interface for the **MPI_Send** routine, as defined in the **MPI_F08** module, illustrates the second case above:

```
SUBROUTINE MPI_Send( buf, count, datatype, dest, &
        tag, comm, ierror )
  TYPE(*), DIMENSION(..), INTENT(IN) :: buf
  INTEGER, INTENT(IN) :: count, dest, tag
  TYPE(MPI_Datatype), INTENT(IN):: datatype
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
END FUNCTION MPI_Send
```

Invocation of this procedure using the same objects as for the first case above could, for example, read

```
! send values in x, y, and z using MPI_Send:
CALL MPI_Send(x, size(x,KIND=c_int), MPI_FLOAT, &
         dest, tag, MPI_COMM_WORLD)
CALL MPI_Send(y(1::2,:), size(y(1::2,:),KIND=c_int), MPI_INT, &
         dest, tag, MPI_COMM_WORLD)
CALL MPI_Send(z, 1_c_int, MPI_DOUBLE, &
         dest, tag, MPI_COMM_WORLD)
```

The second call with a non-contiguous buffer object would, different from the first case, not cause creation of a temporary array copy[5].

If the Fortran default integer interoperates with a C integer type, adding **BIND(C)** to the above interface is permissible; this allows directly writing a C implementation:

```
void mpi_send( CFI_cdesc_t *buf, int count, MPI_Datatype datatype, \
             int dest, int tag, MPI_Comm comm, int *ierror ) {
   int ierror_local;
   MPI_Datatype disc_type;
   if (CFI_is_contiguous(buf)) {
     ierror_local = MPI_Send( buf->base_addr, count, datatype, \
                    dest, tag, comm );
   } else {
     ...  /* use descriptor information to construct disc_type
             from datatype */
     ierror_local = MPI_Send( buf->base_addr, count, disc_type, \
                    dest, tag, comm );
   }
   if (ierror != NULL) *ierror = ierror_local;
}
```

[5] This is especially important for the case of non-blocking communication (**MPI_Isend**, see also the later section on Extension of asynchronous processing) because the actually used communication buffer must persist until the completion procedure **MPI_Wait** has been executed.

For the non-contiguous case, the C code will need to create and commit a matching MPI data type before actually sending the message; the information required to invoke, for example, **`MPI_Type_create_subarray()`**, can be extracted from the descriptor.

Non-interoperable objects and functions

On the Fortran side, non-interoperable data entities may be of non-interoperable intrinsic type (for example, the 16 byte **`REAL`** type supported by some compilers, or the default **`LOGICAL`** type), of extensible type (specifically, types that have components with the **`POINTER`** or **`ALLOCATABLE`** attribute fall into this class), or polymorphic. On the C side, union types and structs with bit fields or flexible array components cannot directly interoperate with a Fortran object. An interoperable Fortran interface through which such objects should be passed can declare them to be of assumed type (the alternative, already supported in Fortran 2003, of declaring them as **`TYPE(c_ptr)`** is also feasible, but more cumbersome to use in procedure invocations); in most cases the companion processor will be obliged to treat such objects as an opaque handle, but sometimes it may be possible to access some or all data inside the object across the language boundary based on separately provided information about subobject type and storage association.

The TS provides extended support for such handle objects by permitting a non-interoperable array as an actual argument in the invocation of the **`C_LOC`** function, and a non-interoperable array pointer as the **`FPTR`** argument of the **`C_F_POINTER`** procedure.

Similarly, the TS extends support for handling non-interoperable procedures by allowing them as an actual argument in the invocation of the **`C_FUNLOC`** function, and pointers to a non-interoperable procedure as the **`FPTR`** argument of the **`C_F_PROCPOINTER`** procedure. This facility can, for example, be used to gain access to Fortran module procedures stored in shared libraries via the C library's dynamic linking loader ("Plug-in"):

```
PROGRAM test_dlfcn
  USE dlfcn; USE plugin
  IMPLICIT NONE
  INTEGER :: istat
  TYPE(dlfcn_handle) :: h
  TYPE(c_funptr) :: cp                    ! C address of procedure
  PROCEDURE(set), POINTER :: fp
  ! abstract interface set defined in plugin
  TYPE(ptype) :: obj
  ! ptype is a non-interoperable type defined in plugin
  h = dlfcn_open('./libplugin.so', rtld_now)
  cp = dlfcn_symbol(h, '__plugin_procs_MOD_set_val')
                       !  ^ gfortran name mangling
  CALL c_f_procpointer(cp, fp)            ! permitted by TS
  obj = ptype( [2] ); CALL fp(obj)
  WRITE(*, *) 'Module procedure plugin_procs::set_val output: ', &
              obj
  istat = dlfcn_close(h)                  ! fp now undefined
END PROGRAM test_dlfcn
```

Herein, the module **dlfcn** (whose implementation is not shown) provides a wrapper for the C library calls **dlopen()**, **dlsym()**, **dlclose()**. The remaining processor dependency results from the name mangling scheme used for module procedures. In combination with the submodule concept from Fortran 2008, this allows for greater flexibility by allowing the program to change between different implementations of the same set of module procedures at run time.

Some items that remain unsupported with respect to interoperation even with the TS' provisions in place are: Procedure dummy arguments cannot be specified in a **BIND(C)** interface. Functions that produce result values with the **POINTER** or **ALLOCATABLE** attribute cannot interoperate either; the same applies to global variables or type components that have these attributes. Also, C functions with variable argument lists remain excluded from interoperation.

Extension of asynchronous processing

Parallel programming with MPI supports a non-blocking data transfer mechannism. For example, point-to-point send (**MPI_Isend**) or receive (**MPI_Irecv**) operations may return even though the send buffer has not yet been transmitted to the matching receiver, or the receive buffer has not yet been fully defined. From the Fortran point of view, such an operation can be considered an **asynchronous initiation procedure**. Such a procedure typically returns a handle that must subsequently be used as an actual argument in invocation of an **asynchronous completion procedure**; once the latter has been successfully executed, the buffers can be assumed to be up-to-date. For MPI, this might be a call to **MPI_Wait** or **MPI_Test**. The purpose of using this style (if fully supported by the MPI implementation) is to improve performance by overlapping communication and computation.

In order to support this programming model on the language level, the TS has extended the semantics of the **ASYNCHRONOUS** attribute beyond the scope of asynchronous file I/O. In order to illustrate this, consider the following code:

```
REAL :: buf(100,100)
TYPE(MPI_Request) :: req
TYPE(MPI_Status) :: status
... ! Code that involves buf
BLOCK
  ASYNCHRONOUS :: buf
  CALL MPI_Irecv( buf, size(buf), MPI_REAL, src, tag, &
                  MPI_COMM_WORLD, req )
  ... ! Code that does not involve buf
  CALL MPI_Wait( req, status )
  ... ! Code that involves buf
END BLOCK
```

Giving an object the **ASYNCHRONOUS** attribute amounts to the following contract between the programmer and the processor's optimizer:

- The programmer has the obligation to not reference or define **`ASYNCHRONOUS`** objects between the invocations of the initiation and the completion procedure the object is involved in (in case the initiation procedure only writes data, this is relaxed to merely not defining such entities);
- The processor's optimizer has the obligation of not performing code movements across either the initiation or the completion procedure that reference or define the **`ASYNCHRONOUS`** objects (in the above situation, this might otherwise happen especially across **`MPI_Wait`** because **`buf`** does not appear as an argument in that call).

If the processor is aware of which procedures initiate or complete asynchronous operations, it can limit the impact on the optimization process to the identified code region; otherwise, it will be obliged to generally suppress code motion for definitions and references on **`ASYNCHRONOUS`** objects across any procedure call. The latter situation arises if dummy arguments are used as buffers involved in asynchronous processing, and initiation and completion occur in different scopes; it is then the programmer's responsibility to specify the **`ASYNCHRONOUS`** attribute wherever it is needed in the call stack, and to obey the restrictions throughout all impacted scopes.

Acknowledgments and further information

The author especially thanks Tobias Burnus for his implementation of the TS' facilities in the GCC gfortran compiler, which permitted implementing and testing the example programs in this article, and also uncovered a number of technical issues in the TS itself. Compilable source code for the examples is available at FIXME link.

FIXME: pointer to TS itself, fortran-dev branch of GCC, list of type specifiers (Appendix)