Chapter 1

Errata for MPI-3.0

This document was processed on January 1, 2015.

The known corrections to MPI-3.0 are listed in this document. All page and line numbers are for the official version of the MPI-3.0 document available from the MPI Forum home page at www.mpi-forum.org. Information on reporting mistakes in the MPI documents is also located on the MPI Forum home page.

- In all `mpi_f08` subroutine and function definitions in Chapters 3–17 and Annex A.3, in Example 5.21 on page 187 line 13, and in all `mpi_f08` `ABSTRACT INTERFACE` definitions (on page 183 line 47, page 268 lines 23 and 33, page 273 line 47, page 274 line 9, page 277 lines 12 and 21, page 344 line 22, page 346 line 12, page 347 line 36, page 475 lines 10 and 43, page 476 line 38, page 537 line 29, page 538 line 2, and page 678 line 11 through page 680 line 35), the `BIND(C)` must be removed.

  Note that a previous version of this errata *added* `BIND(C)` to a routine declaration. That change is now removed.

- Section 6.4.2, page 239 (MPI_Comm_idup) line 32 reads

  ```
  TYPE(MPI_Comm), INTENT(OUT) :: newcomm
  ```

  but should read

  ```
  TYPE(MPI_Comm), INTENT(OUT), ASYNCHRONOUS :: newcomm
  ```

- Section 6.4.4, page 249 (MPI_Comm_set_info) lines 20–21 read

  ```
  MPI_Comm_set_info(MPI_Comm comm, MPI_Info info) BIND(C)
  TYPE(MPI_Comm), INTENT(INOUT) :: comm
  ```

  but should read

  ```
  MPI_Comm_set_info(comm, info, ierror)
  TYPE(MPI_Comm), INTENT(IN) :: comm
  ```

- Section 8.1.1, page 336 (MPI_Get_library_version) line 17 reads

  ```
  MPI_Get_library_version(version, resulten, ierror) BIND(C)
  ```

but should read

```
MPI_Get_library_version(version, resultlen, ierror)
```

- Section 8.1.1, page 336 (MPI_Get_library_version) line 22 reads

```
MPI_GET_LIBRARY_VERSION(VERSION, RESULTEN, IERROR)
```

but should read

```
MPI_GET_LIBRARY_VERSION(VERSION, RESULTLEN, IERROR)
```

- Section 8.2, page 339 lines 44–47, page 407 lines 47 through page 408 line 2, page 409 lines 30–33, and page 411 lines 11–14 read

> If the Fortran compiler provides `TYPE(C_PTR)`, then the following interface must be provided in the `mpi` module and should be provided in `mpif.h` through overloading, i.e., with the same routine name as the routine with `INTEGER(KIND=MPI_ADDRESS_KIND) BASEPTR`, but with a different linker name:

but should read

> If the Fortran compiler provides `TYPE(C_PTR)`, then the following generic interface must be provided in the `mpi` module and should be provided in `mpif.h` through overloading, i.e., with the same routine name as the routine with `INTEGER(KIND=MPI_ADDRESS_KIND) BASEPTR`, but with a different specific procedure name:

- Section 8.2, page 340 lines 1–8, and Annex A.4.6, page 772, lines 38–46 read

```
INTERFACE MPI_ALLOC_MEM
    SUBROUTINE MPI_ALLOC_MEM_CPTR(SIZE, INFO, BASEPTR, IERROR)
        USE, INTRINSIC ::  ISO_C_BINDING, ONLY : C_PTR
        INTEGER ::  INFO, IERROR
        INTEGER(KIND=MPI_ADDRESS_KIND) ::  SIZE
        TYPE(C_PTR) ::  BASEPTR
    END SUBROUTINE
END INTERFACE
```

but should read

```
INTERFACE MPI_ALLOC_MEM
    SUBROUTINE MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
        IMPORT ::  MPI_ADDRESS_KIND
        INTEGER INFO, IERROR
        INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
    END SUBROUTINE
    SUBROUTINE MPI_ALLOC_MEM_CPTR(SIZE, INFO, BASEPTR, IERROR)
        USE, INTRINSIC ::  ISO_C_BINDING, ONLY : C_PTR
        IMPORT ::  MPI_ADDRESS_KIND
```

```
        INTEGER ::  INFO, IERROR
        INTEGER(KIND=MPI_ADDRESS_KIND) ::  SIZE
        TYPE(C_PTR) ::  BASEPTR
    END SUBROUTINE
END INTERFACE
```

- Section 8.2, page 340 (MPI_ALLOC_MEM) lines 10–11 read

  > The linker name base of this overloaded function is MPI_ALLOC_MEM_CPTR. The implied linker names are described in Section 17.1.5 on page 605.

  but should read

  > The base procedure name of this overloaded function is MPI_ALLOC_MEM_CPTR. The implied specific procedure names are described in Section 17.1.5 on page 605.

- Section 11.2.2, page 408, lines 4–12, and Annex A.4.9, page 777, lines 31–40 read

```
INTERFACE MPI_WIN_ALLOCATE
    SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &
    WIN, IERROR)
        USE, INTRINSIC ::  ISO_C_BINDING, ONLY : C_PTR
        INTEGER ::  DISP_UNIT, INFO, COMM, WIN, IERROR
        INTEGER(KIND=MPI_ADDRESS_KIND) ::  SIZE
        TYPE(C_PTR) ::  BASEPTR
    END SUBROUTINE
END INTERFACE
```

but should read

```
INTERFACE MPI_WIN_ALLOCATE
    SUBROUTINE MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &
    WIN, IERROR)
        IMPORT ::  MPI_ADDRESS_KIND
        INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
        INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
    END SUBROUTINE
    SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &
    WIN, IERROR)
        USE, INTRINSIC ::  ISO_C_BINDING, ONLY : C_PTR
        IMPORT ::  MPI_ADDRESS_KIND
        INTEGER ::  DISP_UNIT, INFO, COMM, WIN, IERROR
        INTEGER(KIND=MPI_ADDRESS_KIND) ::  SIZE
        TYPE(C_PTR) ::  BASEPTR
    END SUBROUTINE
END INTERFACE
```

- Section 11.2.2, page 408 (MPI_WIN_ALLOCATE) lines 14–15 read

> The linker name base of this overloaded function is MPI_WIN_ALLOCATE_CPTR. The implied linker names are described in Section 17.1.5 on page 605.

but should read

> The base procedure name of this overloaded function is MPI_WIN_ALLOCATE_CPTR. The implied specific procedure names are described in Section 17.1.5 on page 605.

- Section 11.2.2, Page 408, lines 24–26 read:

> The following info key is predefined:
>
> same_size — if set to true, then the implementation may assume that the argument size is identical on all processes.

That text should be deleted. Add the following text to page 406, after line 10:

> same_size — if set to true, then the implementation may assume that the argument size is identical on all processes.

- Section 11.2.3, page 409, lines 35–43, and Annex A.4.9, page 777, line 46 through page 778, line 6 read

```
INTERFACE MPI_WIN_ALLOCATE_SHARED
    SUBROUTINE MPI_WIN_ALLOCATE_SHARED_CPTR(SIZE, DISP_UNIT, INFO, COMM, &
    BASEPTR, WIN, IERROR)
        USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
        INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
        INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
        TYPE(C_PTR) :: BASEPTR
    END SUBROUTINE
END INTERFACE
```

but should read

```
INTERFACE MPI_WIN_ALLOCATE_SHARED
    SUBROUTINE MPI_WIN_ALLOCATE_SHARED(SIZE, DISP_UNIT, INFO, COMM, &
    BASEPTR, WIN, IERROR)
        IMPORT :: MPI_ADDRESS_KIND
        INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
        INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
    END SUBROUTINE
    SUBROUTINE MPI_WIN_ALLOCATE_SHARED_CPTR(SIZE, DISP_UNIT, INFO, COMM, &
    BASEPTR, WIN, IERROR)
        USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
        IMPORT :: MPI_ADDRESS_KIND
        INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
        INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
        TYPE(C_PTR) :: BASEPTR
    END SUBROUTINE
END INTERFACE
```

- Section 11.2.3, page 409 (MPI_WIN_ALLOCATE_SHARED) lines 44–46 read

  > The linker name base of this overloaded function is MPI_WIN_ALLOCATE_SHARED_CPTR. The implied linker names are described in Section 17.1.5 on page 605.

  but should read

  > The base procedure name of this overloaded function is MPI_WIN_ALLOCATE_SHARED_CPTR. The implied specific procedure names are described in Section 17.1.5 on page 605.

- Section 11.2.3, page 409, line 48: MPI_WIN_ALLOC should be changed to MPI_WIN_ALLOCATE.

- Section 11.2.3, page 411, lines 16–24, and Annex A.4.9, page 779, lines 12–20 read

```
INTERFACE MPI_WIN_SHARED_QUERY
    SUBROUTINE MPI_WIN_SHARED_QUERY_CPTR(WIN, RANK, SIZE, DISP_UNIT, &
    BASEPTR, IERROR)
        USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
        INTEGER :: WIN, RANK, DISP_UNIT, IERROR
        INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
        TYPE(C_PTR) :: BASEPTR
    END SUBROUTINE
END INTERFACE
```

  but should read

```
INTERFACE MPI_WIN_SHARED_QUERY
    SUBROUTINE MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, &
    BASEPTR, IERROR)
        IMPORT :: MPI_ADDRESS_KIND
        INTEGER WIN, RANK, DISP_UNIT, IERROR
        INTEGER (KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
    END SUBROUTINE
    SUBROUTINE MPI_WIN_SHARED_QUERY_CPTR(WIN, RANK, SIZE, DISP_UNIT, &
    BASEPTR, IERROR)
        USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
        IMPORT :: MPI_ADDRESS_KIND
        INTEGER :: WIN, RANK, DISP_UNIT, IERROR
        INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
        TYPE(C_PTR) :: BASEPTR
    END SUBROUTINE
END INTERFACE
```

- Section 11.2.3, page 411 (MPI_WIN_SHARED_QUERY_CPTR) lines 26–27 read

> The linker name base of this overloaded function is MPI_WIN_SHARED_QUERY_CPTR. The implied linker names are described in Section 17.1.5 on page 605.

but should read

> The base procedure name of this overloaded function is MPI_WIN_SHARED_QUERY_CPTR. The implied specific procedure names are described in Section 17.1.5 on page 605.

- Section 11.3.4, page 428, lines 15–18 read

  > Accumulate origin_count elements of type origin_datatype from the origin buffer (origin_addr) to the buffer at offset target_disp, in the target window specified by target_rank and win, using the operation op and return in the result buffer result_addr the content of the target buffer before the accumulation.

  but should say

  > Accumulate origin_count elements of type origin_datatype from the origin buffer (origin_addr) to the buffer at offset target_disp, in the target window specified by target_rank and win, using the operation op and return in the result buffer result_addr the content of the target buffer before the accumulation, specified by target_disp, target_count, and target_datatype. The data transferred from origin to target must fit, without truncation, in the target buffer. Likewise, the data copied from target to origin must fit, without truncation, in the result buffer.

- Section 11.3.4, page 428, line 30, add

  > When MPI_NO_OP is specified as the operation, the origin_addr, origin_count, and origin_datatype arguments are ignored.

  after

  > the origin and no operation is performed on the target buffer.

- Section 11.7.3, page 464, lines 16–20 read

  > While this ambiguity is unfortunate, it does not seem to affect many real codes. The MPI Forum decided not to decide which interpretation of the standard is the correct one, since the issue is very contentious, and a decision would have much impact on implementors but less impact on users.

  but should be

  > While this ambiguity is unfortunate, the MPI Forum decided not to define which interpretation of the standard is the correct one, since the issue is contentious.

- Section 11.8, example 11.21, page 469, in line 32 change

```
double **baseptr;
```

to

```
double *baseptr;
```

and in line 36, change

```
  MPI_COMM_WORLD, baseptr, &win);
```

to

```
  MPI_COMM_WORLD, &baseptr, &win);
```

- Section 14.2.1, page 555 (Profiling interface) lines 38–40 read

  > For Fortran, the different support methods cause several linker names. Therefore, several profiling routines (with these linker names) are needed for each Fortran MPI routine, as described in Section 17.1.5 on page 605.

  but should read

  > For Fortran, the different support methods cause several specific procedure names. Therefore, several profiling routines (with these specific procedure names) are needed for each Fortran MPI routine, as described in Section 17.1.5 on page 605.

- Section 14.2.7, page 560 (Profiling interface, Fortran support methods) lines 29–32 read

  > The different Fortran support methods and possible options for the support of subarrays (depending on whether the compiler can support `TYPE(*), DIMENSION(..)` choice buffers) imply different linker names for the same Fortran MPI routine. The rules and implications for the profiling interface are described in Section 17.1.5 on page 605.

  but should read

  > The different Fortran support methods and possible options for the support of subarrays (depending on whether the compiler can support `TYPE(*), DIMENSION(..)` choice buffers) imply different specific procedure names for the same Fortran MPI routine. The rules and implications for the profiling interface are described in Section 17.1.5 on page 605.

- Section 14.3, page 561, line 22 add

  > Variables and categories across connected processes with equivalent names are required to have the same meaning (see the definition of "equivalent" as related to strings in Section 14.3.3). Furthermore, enumerations with equivalent names across connected processes are required to have the same meaning, but are allowed to comprise different enumeration items. Enumeration items that have equivalent names across connected processes in enumerations with the same meaning must also have the same meaning. In

order for variables and categories to have the same meaning, routines in the tools information interface that return details for those variables and categories have requirements on what parameters must be identical. These requirements are specified in their respective sections.

- Section 14.3, page 561, lines 33–36 read

  Since the MPI tool information interface primarily focuses on tools and support libraries, MPI implementations are only required to provide C bindings for functions introduced in this section.

  but should read

  Since the MPI tool information interface primarily focuses on tools and support libraries, MPI implementations are only required to provide C bindings for functions and constants introduced in this section.

- Section 14.3, page 561, lines 43–45 read

  Further, there is no guarantee that the number of variables, variable indices, and variable names are the same across connected processes.

  but should read

  Further, there is no guarantee that the number of variables and variable indices are the same across connected processes.

- Section 14.3.3, page 563 line 34 add

  MPI implementations behave as if they have an internal character array that is copied to the output character array supplied by the user. Such output strings are defined to be equivalent if their notional source internal character arrays are identical (up to and including the null terminator), even if the output string is truncated due to a small input length parameter n.

- Section 14.3.5, line 36 add

  The use of the datatype MPI_CHAR in the MPI tool information interface implies a null-terminated character array, i.e., a string in the C language. If a variable has type MPI_CHAR, the value of the count parameter returned by MPI_T_CVAR_HANDLE_ALLOC and MPI_T_PVAR_HANDLE_ALLOC must be large enough to include any valid value, including its terminating null character. The contents of returned MPI_CHAR arrays are only defined from index 0 through the location of the first null character.

- Page 569, line 11 add

  If the name of a control variable is equivalent across connected processes, the following OUT parameters must be identical: verbosity, datatype, enumtype, bind, and scope. The returned description must be equivalent.

- Page 574, lines 10–16 read

> A performance variable in this class represents a value that is the fixed size of a resource. Values returned from variables in this class are non-negative and represented by one of the following datatypes: MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_UNSIGNED_LONG_LONG, MPI_DOUBLE. The starting value is the current utilization level of the resource at the time that the starting value is set. MPI implementations must ensure that variables of this class cannot overflow.

but should read

> A performance variable in this class represents a value that is the size of a resource. Values returned from variables in this class are non-negative and represented by one of the following datatypes: MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_UNSIGNED_LONG_LONG, MPI_DOUBLE. The starting value is the current size of the resource at the time that the starting value is set. MPI implementations must ensure that variables of this class cannot overflow.

- Page 574, lines 31–32 and 40–41 read

  > The starting value is the current utilization level of the resource at the time that the starting value is set.

  but should read

  > The starting value is the current utilization level of the resource at the time that the variable is started or reset.

- Page 577, line 32 add

  > If a performance variable has an equivalent name and has the same class across connected processes, the following OUT parameters must be identical: verbosity, varclass, datatype, enumtype, bind, readonly, continuous, and atomic. The returned description must be equivalent.

- Page 579, line 7 add:

  > For all routines in the rest of this section that take both handle and session as IN arguments, if the handle argument passed in is not associated with the session argument, MPI_T_ERR_INVALID_HANDLE is returned.

- Page 579, line 41 add the following after the word successfully:

  > (even if there are no non-continuous variables to be started)

- Page 580, line 13 add the following after the word successfully:

  > (even if there are no non-continuous variables to be stopped)

- Page 581, line 25 add the following after the word successfully:

  > (even if there are no valid handles or all are read-only)

- Page 585 line 21 of 3.0 reads:

  > The following function can be used to query the number of control variables, N.

  but should read

  > The following function can be used to query the number of categories, N.

- Page 586, line 34 add

  > If the name of a category is equivalent across connected processes, then the returned description must be equivalent.

- Page 589, lines 11–12 read

  > The enumeration index is invalid or has been deleted.

  > The enumeration index is invalid.

- Page 589, line 19 reads

  > The variable index is invalid or has been deleted.

  > The variable index is invalid.

- Section 17.1.1, page 598 (Fortran support, overview) lines 29–32 read

  > The Fortran interfaces of each MPI routine are shorthands. Section 17.1.5 defines the corresponding full interface specification together with the used linker names and implications for the profiling interface.

  but should read

  > The Fortran interfaces of each MPI routine are shorthands. Section 17.1.5 defines the corresponding full interface specification together with the specific procedure names and implications for the profiling interface.

- Section 17.1.2, page 599 (Fortran support through the `mpi_f08` module) lines 19–20 read

  > Define all MPI handles with uniquely named handle types (instead of `INTEGER` handles, as in the `mpi` module).

  but should read

  > Define the derived type MPI_Status, and define all MPI handles with uniquely named handle types (instead of `INTEGER` handles, as in the `mpi` module).

- Section 17.1.2, page 601 (Fortran support through the `mpi_f08` module) lines 11–15 read

> The INTERFACE construct in combination with BIND(C) allows the implementation of the Fortran mpi_f08 interface with a single set of portable wrapper routines written in C, which supports all desired features in the mpi_f08 interface. TS 29113 also has a provision for OPTIONAL arguments in BIND(C) interfaces.

but should be removed.

- Section 17.1.3 (mpi module), page 601 lines 33–35 read

> Provide explicit interfaces according to the Fortran routine interface specifications. This module therefore guarantees compile-time argument checking and allows positional and keyword-based argument lists.

but should read

> Provide explicit interfaces according to the Fortran routine interface specifications. This module therefore guarantees compile-time argument checking and allows positional and keyword-based argument lists. If an implementation is paired with a compiler that either does not support TYPE(*), DIMENSION(..) from TS 29113, or is otherwise unable to ignore the types of choice buffers, then the implementation must provide explicit interfaces only for MPI routines with no choice buffer arguments. See Section 17.1.6 on page 609 for more details.

- Both the last Advice to implementors in Section 17.1.4 (Fortran support through the mpif.h include file), page 604 line 29 through page 605 line 11, and the whole of Section 17.1.5 (Interface specification, linker names and the profiling interface), page 605 line 29 through page 609 line 31 are replaced with the following:

17.1.5 Interface Specifications, Procedure Names, and the Profiling Interface

The Fortran interface specification of each MPI routine specifies the routine name that must be called by the application program, and the names and types of the dummy arguments together with additional attributes. The Fortran standard allows a given Fortran interface to be implemented with several methods, e.g., within or outside of a module, with or without BIND(C), or the buffers with or without TS 29113. Such implementation decisions imply different binary interfaces and different specific procedure names. The requirements for several implementation schemes together with the rules for the specific procedure names and its implications for the profiling interface are specified within this section, but not the implementation details.

Rationale. This section was introduced in MPI-3.0 on Sep. 21, 2012. The major goals for implementing the three Fortran support methods have been:

- Portable implementation of the wrappers from the MPI Fortran interfaces to the MPI routines in C.
- Binary backward compatible implementation path when switching MPI_SUBARRAYS_SUPPORTED from .FALSE. to .TRUE..

- The Fortran PMPI interface need not be backward compatible, but a method must be included that a tools layer can use to examine the MPI library about the specific procedure names and interfaces used.
- No performance drawbacks.
- Consistency between all three Fortran support methods.
- Consistent with Fortran 2008 + TS 29113.

The design expected that all dummy arguments in the MPI Fortran interfaces are interoperable with C according to Fortran 2008 + TS 29113. This expectation was not fulfilled. The `LOGICAL` arguments are not interoperable with C, mainly because the internal representations for `.FALSE.` and `.TRUE.` are compiler dependent. The provided interface was mainly based on `BIND(C)` interfaces and therefore inconsistent with Fortran. To be consistent with Fortran, the `BIND(C)` had to be removed from the callback procedure interfaces and the predefined callbacks, e.g., MPI_COMM_DUP_FN. Non-`BIND(C)` procedures are also not interoperable with C, and therefore the `BIND(C)` had to be removed from all routines with `PROCEDURE` arguments, e.g., from MPI_OP_CREATE.

Therefore, this section was rewritten as an erratum to MPI-3.0. (*End of rationale.*)

A Fortran call to an MPI routine shall result in a call to a procedure with one of the specific procedure names and calling conventions, as described in Table 1.1 on page 13. Case is not significant in the names.

Note that for the deprecated routines in Section 15.1 on page 591, which are reported only in Annex A.4, scheme 2A is utilized in the `mpi` module and `mpif.h`, and also in the `mpi_f08` module.

To set MPI_SUBARRAYS_SUPPORTED to `.TRUE.` within a Fortran support method, it is required that all non-blocking and split-collective routines with buffer arguments are implemented according to 1B and 2B, i.e., with MPI_Xxxx_f08ts in the `mpi_f08` module, and with MPI_XXXX_FTS in the `mpi` module and the `mpif.h` include file.

The `mpi` and `mpi_f08` modules and the `mpif.h` include file will each correspond to exactly one implementation scheme from Table 1.1 on page 13. However, the MPI library may contain multiple implementation schemes from Table 1.1.

Advice to implementors. This may be desirable for backwards binary compatibility in the scope of a single MPI implementation, for example. (*End of advice to implementors.*)

Rationale. After a compiler provides the facilities from TS 29113, i.e., `TYPE(*), DIMENSION(..)`, it is possible to change the bindings within a Fortran support method to support subarrays without recompiling the complete application provided that the previous interfaces with their specific procedure names are still included in the library. Of course, only recompiled routines can benefit from the added facilities. There is no binary compatibility conflict because each interface uses its own specific procedure names and all interfaces use the same constants (except the value of MPI_SUBARRAYS_SUPPORTED and MPI_ASYNC_PROTECTS_NONBLOCKING) and type definitions. After a compiler also ensures that buffer arguments of nonblocking MPI operations can be protected through the `ASYNCHRONOUS` attribute, and the procedure declarations in the `mpi_f08` and `mpi` module and the `mpif.h` include file declare

No.	Specific procedure name	Calling convention
1A	MPI_Isend_f08	Fortran interface and arguments, as in Annex A.3, except that in routines with a choice buffer dummy argument, this dummy argument is implemented with non-standard extensions like `!$PRAGMA IGNORE_TKR`, which provides a call-by-reference argument without type, kind, and dimension checking.
1B	MPI_Isend_f08ts	Fortran interface and arguments, as in Annex A.3, but only for routines with one or more choice buffer dummy arguments; these dummy arguments are implemented with `TYPE(*), DIMENSION(..)`.
2A	MPI_ISEND	Fortran interface and arguments, as in Annex A.4, except that in routines with a choice buffer dummy argument, this dummy argument is implemented with non-standard extensions like `!$PRAGMA IGNORE_TKR`, which provides a call-by-reference argument without type, kind, and dimension checking.
2B	MPI_ISEND_FTS	Fortran interface and arguments, as in Annex A.4, but only for routines with one or more choice buffer dummy arguments; these dummy arguments are implemented with `TYPE(*), DIMENSION(..)`.

Table 1.1: Specific Fortran procedure names and related calling conventions. MPI_ISEND is used as an example. For routines without choice buffers, only 1A and 2A apply.

choice buffers with the ASYNCHRONOUS attribute, then the value of MPI_ASYNC_PROTECTS_NONBLOCKING can be switched to .TRUE. in the module definition and include file. (*End of rationale.*)

Advice to users. Partial recompilation of user applications when upgrading MPI implementations is a highly complex and subtle topic. Users are strongly advised to consult their MPI implementation's documentation to see exactly what is — and what is not — supported. (*End of advice to users.*)

Within the mpi_f08 and mpi modules and mpif.h, for all MPI procedures, a second procedure with the same calling conventions shall be supplied, except that the name is modified by prefixing with the letter "P", e.g., PMPI_Isend. The specific procedure names for these PMPI_Xxxx procedures must be different from the specific procedure names for the MPI_Xxxx procedures and are not specified by this standard.

A user-written or middleware profiling routine should provide the same specific Fortran procedure names and calling conventions, and therefore can interpose itself as the MPI library routine. The profiling routine can internally call the matching PMPI routine with any of its existing bindings, except for routines that have callback routine dummy arguments, choice buffer arguments, or that are attribute caching routines (MPI_{COMM|WIN|TYPE}_{SET|GET}_ATTR). In this case, the profiling software should invoke the corresponding PMPI routine using the same Fortran support method as used in the calling application program, because the C, mpi_f08 and mpi callback prototypes are different or the meaning of the choice buffer or attribute_val arguments are different.

Advice to users. Although for each support method and MPI routine (e.g., MPI_ISEND in mpi_f08), multiple routines may need to be provided to intercept the specific procedures in the MPI library (e.g., MPI_Isend_f08 and MPI_Isend_f08ts), each profiling routine itself uses only one support method (e.g., mpi_f08) and calls the real MPI routine through the one PMPI routine defined in this support method (i.e., PMPI_Isend in this example). (*End of advice to users.*)

Advice to implementors. If all of the following conditions are fulfilled:

- the handles in the mpi_f08 module occupy one Fortran numerical storage unit (same as an INTEGER handle),
- the internal argument passing mechanism used to pass an actual ierror argument to a non-optional ierror dummy argument is binary compatible to passing an actual ierror argument to an ierror dummy argument that is declared as OPTIONAL,
- the internal argument passing mechanism for ASYNCHRONOUS and non-ASYNCHRONOUS arguments is the same,
- the internal routine call mechanism is the same for the Fortran and the C compilers for which the MPI library is compiled,
- the compiler does not provide TS 29113,

then the implementor may use the same internal routine implementations for all Fortran support methods but with several different specific procedure names. If the accompanying Fortran compiler supports TS 29113, then the new routines are needed only for routines with choice buffer arguments. (*End of advice to implementors.*)

Advice to implementors. In the Fortran support method `mpif.h`, compile-time argument checking can be also implemented for all routines. For `mpif.h`, the argument names are not specified through the MPI standard, i.e., only positional argument lists are defined, and not key-word based lists. Due to the rule that `mpif.h` must be valid for fixed and free source form, the subroutine declaration is restricted to one line with 72 characters. To keep the argument lists short, each argument name can be shortened to a minimum of one character. With this, the two longest subroutine declaration statements are

```
      SUBROUTINE PMPI_Dist_graph_create_adjacent(a,b,c,d,e,f,g,h,i,j,k)
      SUBROUTINE PMPI_Rget_accumulate(a,b,c,d,e,f,g,h,i,j,k,l,m,n)
```

with 71 and 66 characters. With buffers implemented with TS 29113, the specific procedure names have an additional postfix. The longest of such interface definitions is

```
      INTERFACE  PMPI_Rget_accumulate
      SUBROUTINE PMPI_Rget_accumulate_fts(a,b,c,d,e,f,g,h,i,j,k,l,m,n)
```

with 70 characters. In principle, continuation lines would be possible in `mpif.h` (spaces in columns 73–131, & in column 132, and in column 6 of the continuation line) but this would not be valid if the source line length is extended with a compiler flag to 132 characters. Column 133 is also not available for the continuation character because lines longer than 132 characters are invalid with some compilers by default.

The longest specific procedure names are PMPI_Dist_graph_create_adjacent_f08 and PMPI_File_write_ordered_begin_f08ts both with 35 characters in the `mpi_f08` module.

For example, the interface specifications together with the specific procedure names can be implemented with

```
MODULE mpi_f08
  TYPE, BIND(C) :: MPI_Comm
    INTEGER :: MPI_VAL
  END TYPE MPI_Comm
  ...
  INTERFACE MPI_Comm_rank  ! (as defined in Chapter 6)
    SUBROUTINE MPI_Comm_rank_f08(comm, rank, ierror)
      IMPORT :: MPI_Comm
      TYPE(MPI_Comm),      INTENT(IN)  :: comm
      INTEGER,             INTENT(OUT) :: rank
      INTEGER, OPTIONAL,   INTENT(OUT) :: ierror
    END SUBROUTINE
  END INTERFACE
END MODULE mpi_f08

MODULE mpi
  INTERFACE MPI_Comm_rank  ! (as defined in Chapter 6)
    SUBROUTINE MPI_Comm_rank(comm, rank, ierror)
      INTEGER, INTENT(IN)  :: comm   ! The INTENT may be added although
      INTEGER, INTENT(OUT) :: rank   ! it is not defined in the
      INTEGER, INTENT(OUT) :: ierror ! official routine definition.
```

```
    END SUBROUTINE
  END INTERFACE
END MODULE mpi
```

And if interfaces are provided in `mpif.h`, they might look like this (outside of any module and in fixed source format):

```
!23456789012345678901234567890123456789012345678901234567890123456789012
      INTERFACE MPI_Comm_rank  ! (as defined in Chapter 6)
       SUBROUTINE MPI_Comm_rank(comm, rank, ierror)
        INTEGER, INTENT(IN)  :: comm   ! The argument names may be
        INTEGER, INTENT(OUT) :: rank   ! shortened so that the
        INTEGER, INTENT(OUT) :: ierror ! subroutine line fits to the
       END SUBROUTINE                  ! maximum of 72 characters.
      END INTERFACE
```

(*End of advice to implementors.*)

Advice to users. The following is an example of how a user-written or middleware profiling routine can be implemented:

```
SUBROUTINE MPI_Isend_f08ts(buf,count,datatype,dest,tag,comm,request,ierror)
  USE :: mpi_f08, my_noname => MPI_Isend_f08ts
  TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
  INTEGER,            INTENT(IN)       :: count, dest, tag
  TYPE(MPI_Datatype), INTENT(IN)       :: datatype
  TYPE(MPI_Comm),     INTENT(IN)       :: comm
  TYPE(MPI_Request),  INTENT(OUT)      :: request
  INTEGER, OPTIONAL,  INTENT(OUT)      :: ierror
    ! ... some code for the begin of profiling
  call PMPI_Isend (buf, count, datatype, dest, tag, comm, request, ierror)
    ! ... some code for the end of profiling
END SUBROUTINE MPI_Isend_f08ts
```

Note that this routine is used to intercept the existing specific procedure name MPI_Isend_f08ts in the MPI library. This routine must not be part of a module. This routine itself calls PMPI_Isend. The `USE` of the `mpi_f08` module is needed for definitions of handle types and the interface for PMPI_Isend. However, this module also contains an interface definition for the specific procedure name MPI_Isend_f08ts that conflicts with the definition of this profiling routine (i.e., the name is doubly defined). Therefore, the `USE` here specifically excludes the interface from the module by renaming the unused routine name in the `mpi_f08` module into "`my_noname`" in the scope of this routine. (*End of advice to users.*)

Advice to users. The PMPI interface allows intercepting MPI routines. For example, an additional MPI_ISEND profiling wrapper can be provided that is called by the application and internally calls PMPI_ISEND. There are two typical use cases: a profiling layer that is developed independently from the application and the MPI library, and profiling routines that are part of the application and have access to the application data. With MPI-3.0, new Fortran interfaces and implementation schemes were introduced that have several implications on how Fortran MPI routines are internally implemented and optimized. For profiling layers, these schemes imply that several internal interfaces with different specific procedure names may need to be intercepted, as shown in the example code above. Therefore, for wrapper routines that are part of a Fortran application, it may be more convenient to make the name shift within the application, i.e., to substitute the call to the MPI routine (e.g., MPI_ISEND) by a call to a user-written profiling wrapper with a new name (e.g., X_MPI_ISEND) and to call the Fortran MPI_ISEND from this wrapper, instead of using the PMPI interface. (*End of advice to users.*)

- Section 17.1.6, page 610 (MPI for different Fortran standard versions) line 27 reads

  > The routines are not `BIND(C)`.

  but should be removed.

- Section 17.1.6, page 610 (MPI for different Fortran standard versions) line 33 reads

  > The linker names are specified in Section 17.1.5 on page 605.

  but should read

  > The specific procedure names are specified in Section 17.1.5 on page 605.

- Section 17.1.6, page 611 (MPI for different Fortran standard versions) line 21 reads

  > `BIND(C, NAME='...')` interfaces.

  but should be removed.

- After Section 17.1.6, page 611 (MPI for different Fortran standard versions) line 26, which reads

  > arguments.

  the following list item should be added:

  > The ability to overload the operators `.EQ.` and `.NE.` to allow the comparison of derived types (used in MPI-3.0 for MPI handles).

- Section 17.1.6, page 611 (MPI for different Fortran standard versions) line 43 reads

  > The routines are not `BIND(C)`.

  but should be removed.

- Section 17.1.6, page 611 (MPI for different Fortran standard versions) line 47 reads

  > The linker names are specified in Section 17.1.5 on page 605.

  but should read

  > The specific procedure names are specified in Section 17.1.5 on page 605.

- Section 17.1.6, page 612 (MPI for different Fortran standard versions) lines 22–24 read

  - `OPTIONAL` dummy arguments are allowed in combination with `BIND(C)` interfaces.
  - `CHARACTER(LEN=*)` dummy arguments are allowed in combination with `BIND(C)` interfaces.

  but should be removed.

- Section 17.1.7, page 614 (Requirements on Fortran compilers) lines 25–47 read

  > All of these rules are valid independently of whether the MPI routine interfaces in the `mpi_f08` and `mpi` modules are internally defined with an `INTERFACE` or `CONTAINS` construct, and with or without `BIND(C)`, and also if `mpif.h` uses explicit interfaces.
  >
  > *Advice to implementors.* Some of these rules are already part of the Fortran 2003 standard if the MPI interfaces are defined without `BIND(C)`. Additional compiler support may be necessary if `BIND(C)` is used. Some of these additional requirements are defined in the Fortran TS 29113 [41]. Some of these requirements for MPI-3.0 are beyond the scope of TS 29113. (*End of advice to implementors.*)
  >
  > Further requirements apply if the MPI library internally uses `BIND(C)` routine interfaces (i.e., for a full implementation of `mpi_f08`):
  >
  > - Non-buffer arguments are `INTEGER`, `INTEGER(KIND=...)`, `CHARACTER(LEN=*)`, `LOGICAL`, and `BIND(C)` derived types (handles and status in `mpi_f08`), variables and arrays; function results are `DOUBLE PRECISION`. All these types must be valid as dummy arguments in the `BIND(C)` MPI routine interfaces. When compiling an MPI application, the compiler should not issue warnings indicating that these types may not be interoperable with an existing type in C. Some of these types are already valid in `BIND(C)` interfaces since Fortran 2003, some may be valid based on TS 29113 (e.g., `CHARACTER*(*)`).
  > - `OPTIONAL` dummy arguments are also valid within `BIND(C)` interfaces. This requirement is fulfilled if TS 29113 is fully supported by the compiler.

  but should read

  > All of these rules are valid for the `mpi_f08` and `mpi` modules and independently of whether `mpif.h` uses explicit interfaces.
  >
  > *Advice to implementors.* Some of these rules are already part of the Fortran 2003 standard, some of these requirements require the Fortran TS 29113 [41], and some of these requirements for MPI-3.0 are beyond the scope of TS 29113. (*End of advice to implementors.*)

- Annex A.1, page 674, line 31 reads

  > Fortran Type: INTEGER

  but should be deleted.

- Annex A.1, page 675, line 4 reads

Fortran Type: INTEGER

but should be deleted.

- Annex A.1, page 675, line 21 reads

  Fortran Type: INTEGER

  but should be deleted.

- Annex A.1, page 676, line 4 reads

  Fortran Type: INTEGER

  but should be deleted.

- Annex A.1.2, page 677 (Handle types in the `mpi_f08` and `mpi` modules) line 10 reads

  ```
  TYPE(MPI_Info)
  ```

  but should read

  ```
  TYPE(MPI_Info)
  TYPE(MPI_Message)
  ```

- Annex A.1.5 Info Keys, page 683, lines 17 and later, add (maintaining the sorted order):

  accumulate_ops
  accumulate_ordering
  alloc_shared_noncontig
  same_size

- Annex A.1.6 Info Values, page 684, beginning at line 1, add (maintaining the sorted order):

  rar
  raw
  same_op
  same_op_no_op
  war
  waw

- Annex A.2.11, page 700, line 46 reads

  ```
  int MPI_File_close(MPI_File *fh)
  ```

  but should read (add MPI_CONVERSION_FN_NULL before)

  ```
  int MPI_CONVERSION_FN_NULL(void *userbuf, MPI_Datatype datatype, int
  count, void *filebuf, MPI_Offset position, void *extra_state)

  int MPI_File_close(MPI_File *fh)
  ```

- Annex A.3.4, page 724 lines 15–40 read

```
MPI_COMM_DUP_FN(oldcomm, comm_keyval, extra_state, attribute_val_in,
attribute_val_out, flag, ierror) BIND(C)
    TYPE(MPI_Comm), INTENT(IN) ::  oldcomm
    INTEGER, INTENT(IN) ::  comm_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  extra_state,
    attribute_val_in
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::  attribute_val_out
    LOGICAL, INTENT(OUT) ::  flag
    INTEGER, INTENT(OUT) ::  ierror

MPI_COMM_NULL_COPY_FN(oldcomm, comm_keyval, extra_state,
attribute_val_in, attribute_val_out, flag, ierror) BIND(C)
    TYPE(MPI_Comm), INTENT(IN) ::  oldcomm
    INTEGER, INTENT(IN) ::  comm_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  extra_state,
    attribute_val_in
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::  attribute_val_out
    LOGICAL, INTENT(OUT) ::  flag
    INTEGER, INTENT(OUT) ::  ierror

MPI_COMM_NULL_DELETE_FN(comm, comm_keyval, attribute_val, extra_state,
ierror) BIND(C)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, INTENT(IN) ::  comm_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  attribute_val,
    extra_state
    INTEGER, INTENT(OUT) ::  ierror
```

but should read (without all `INTENT` information and `BIND(C)`)

```
MPI_COMM_DUP_FN(oldcomm, comm_keyval, extra_state, attribute_val_in,
attribute_val_out, flag, ierror)
    TYPE(MPI_Comm) ::  oldcomm
    INTEGER ::  comm_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND) ::  extra_state, attribute_val_in
    INTEGER(KIND=MPI_ADDRESS_KIND) ::  attribute_val_out
    LOGICAL ::  flag
    INTEGER ::  ierror

MPI_COMM_NULL_COPY_FN(oldcomm, comm_keyval, extra_state,
attribute_val_in, attribute_val_out, flag, ierror)
    TYPE(MPI_Comm) ::  oldcomm
    INTEGER ::  comm_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND) ::  extra_state, attribute_val_in
    INTEGER(KIND=MPI_ADDRESS_KIND) ::  attribute_val_out
    LOGICAL ::  flag
    INTEGER ::  ierror
```

```
MPI_COMM_NULL_DELETE_FN(comm, comm_keyval, attribute_val, extra_state,
ierror)
    TYPE(MPI_Comm) ::  comm
    INTEGER ::  comm_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND) ::  attribute_val, extra_state
    INTEGER ::  ierror
```

- Annex A.3.4, page 728 line 44 through page 729 line 22 reads

```
MPI_TYPE_DUP_FN(oldtype, type_keyval, extra_state, attribute_val_in,
attribute_val_out, flag, ierror) BIND(C)
    TYPE(MPI_Datatype), INTENT(IN) ::  oldtype
    INTEGER, INTENT(IN) ::  type_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  extra_state,
    attribute_val_in
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::  attribute_val_out
    LOGICAL, INTENT(OUT) ::  flag
    INTEGER, INTENT(OUT) ::  ierror

MPI_TYPE_NULL_COPY_FN(oldtype, type_keyval, extra_state,
attribute_val_in, attribute_val_out, flag, ierror) BIND(C)
    TYPE(MPI_Datatype), INTENT(IN) ::  oldtype
    INTEGER, INTENT(IN) ::  type_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  extra_state,
    attribute_val_in
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::  attribute_val_out
    LOGICAL, INTENT(OUT) ::  flag
    INTEGER, INTENT(OUT) ::  ierror

MPI_TYPE_NULL_DELETE_FN(datatype, type_keyval, attribute_val,
extra_state, ierror) BIND(C)
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER, INTENT(IN) ::  type_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  attribute_val,
    extra_state
    INTEGER, INTENT(OUT) ::  ierror
```

but should read (without all `INTENT` information and `BIND(C)`)

```
MPI_TYPE_DUP_FN(oldtype, type_keyval, extra_state, attribute_val_in,
attribute_val_out, flag, ierror)
    TYPE(MPI_Datatype) ::  oldtype
    INTEGER ::  type_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND) ::  extra_state, attribute_val_in
    INTEGER(KIND=MPI_ADDRESS_KIND) ::  attribute_val_out
    LOGICAL ::  flag
    INTEGER ::  ierror

MPI_TYPE_NULL_COPY_FN(oldtype, type_keyval, extra_state,
attribute_val_in, attribute_val_out, flag, ierror)
```

```
    TYPE(MPI_Datatype) :: oldtype
    INTEGER :: type_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in
    INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val_out
    LOGICAL :: flag
    INTEGER :: ierror

MPI_TYPE_NULL_DELETE_FN(datatype, type_keyval, attribute_val,
extra_state, ierror)
    TYPE(MPI_Datatype) :: datatype
    INTEGER :: type_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
    INTEGER :: ierror
```

- Annex A.3.4, page 730 lines 15–38 read

```
MPI_WIN_DUP_FN(oldwin, win_keyval, extra_state, attribute_val_in,
attribute_val_out, flag, ierror) BIND(C)
    INTEGER, INTENT(IN) :: oldwin, win_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state,
    attribute_val_in
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val_out
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, INTENT(OUT) :: ierror

MPI_WIN_NULL_COPY_FN(oldwin, win_keyval, extra_state,
attribute_val_in, attribute_val_out, flag, ierror) BIND(C)
    INTEGER, INTENT(IN) :: oldwin, win_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state,
    attribute_val_in
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val_out
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, INTENT(OUT) :: ierror

MPI_WIN_NULL_DELETE_FN(win, win_keyval, attribute_val, extra_state,
ierror) BIND(C)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, INTENT(IN) :: win_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val,
    extra_state
    INTEGER, INTENT(OUT) :: ierror
```

but should read (without all `INTENT` information, `BIND(C)`, and oldwin as `TYPE(MPI_Win)`)

```
MPI_WIN_DUP_FN(oldwin, win_keyval, extra_state, attribute_val_in,
attribute_val_out, flag, ierror)
    TYPE(MPI_Win) :: oldwin
    INTEGER :: win_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in
```

```
    INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val_out
    LOGICAL :: flag
    INTEGER :: ierror

MPI_WIN_NULL_COPY_FN(oldwin, win_keyval, extra_state,
attribute_val_in, attribute_val_out, flag, ierror)
    TYPE(MPI_Win) :: oldwin
    INTEGER :: win_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in
    INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val_out
    LOGICAL :: flag
    INTEGER :: ierror

MPI_WIN_NULL_DELETE_FN(win, win_keyval, attribute_val, extra_state,
ierror)
    TYPE(MPI_Win) :: win
    INTEGER :: win_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
    INTEGER :: ierror
```

- Annex A.3.11, page 747, line 36 reads

  ```
  MPI_File_close(fh, ierror) BIND(C)
  ```

  but should read (add MPI_CONVERSION_FN_NULL before, but without `BIND(C)`)

  ```
  MPI_CONVERSION_FN_NULL(userbuf, datatype, count, filebuf, position,
  extra_state, ierror)
      USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
      TYPE(C_PTR), VALUE :: userbuf, filebuf
      TYPE(MPI_Datatype) :: datatype
      INTEGER :: count, ierror
      INTEGER(KIND=MPI_OFFSET_KIND) :: position
      INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state

  MPI_File_close(fh, ierror)
  ```

- Annex A.4.11, page 780, line 22 reads

  ```
  MPI_FILE_CLOSE(FH, IERROR)
  ```

  but should read (add MPI_CONVERSION_FN_NULL before)

  ```
  MPI_CONVERSION_FN_NULL(USERBUF, DATATYPE, COUNT, FILEBUF, POSITION,
  EXTRA_STATE, IERROR)
      <TYPE> USERBUF(*), FILEBUF(*)
      INTEGER COUNT, DATATYPE, IERROR
      INTEGER(KIND=MPI_OFFSET_KIND) POSITION
      INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE

  MPI_FILE_CLOSE(FH, IERROR)
  ```