

*D R A F T*

# Document for a Standard Message-Passing Interface

Message Passing Interface Forum

October 23, 2011

This work was supported in part by NSF and ARPA under NSF contract CDA-9115428 and Esprit under project HPC Standards (21111).

This is the result of a LaTeX run of a draft of a single chapter of the MPIF Final Report document.

## Chapter 13

# Deprecated Functions

### 13.1 Deprecated since MPI-2.0

The following function is deprecated and is superseded by `MPI_TYPE_CREATE_HVECTOR` in MPI-2.0. The language independent definition and the C binding of the deprecated function is the same as of the new function, except of the function name. Only the Fortran language binding is different.

`MPI_TYPE_HVECTOR( count, blocklength, stride, oldtype, newtype)`

IN	count	number of blocks (non-negative integer)
IN	blocklength	number of elements in each block (non-negative integer)
IN	stride	number of bytes between start of each block (integer)
IN	oldtype	old datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPI_Type_hvector(int count, int blocklength, MPI_Aint stride,  
                    MPI_Datatype oldtype, MPI_Datatype *newtype)
```

```
MPI_TYPE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)  
    INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR
```

The following function is deprecated and is superseded by `MPI_TYPE_CREATE_HINDEXED` in MPI-2.0. The language independent definition and the C binding of the deprecated function is the same as of the new function, except of the function name. Only the Fortran language binding is different.

```

1 MPI_TYPE_HINDEXED( count, array_of_blocklengths, array_of_displacements, oldtype, new-
2     type)
3     IN          count          number of blocks – also number of entries in
4                               array_of_displacements and array_of_blocklengths (non-
5                               negative integer)
6
7     IN          array_of_blocklengths  number of elements in each block (array of non-negative
8                               integers)
9
10    IN          array_of_displacements  byte displacement of each block (array of integer)
11
12    IN          oldtype              old datatype (handle)
13
14    OUT         newtype              new datatype (handle)

```

```

14 int MPI_Type_hindexed(int count, int *array_of_blocklengths,
15     MPI_Aint *array_of_displacements, MPI_Datatype oldtype,
16     MPI_Datatype *newtype)
17
18 MPI_TYPE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
19     OLDTYPE, NEWTYPE, IERROR)
20
21 INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
22     OLDTYPE, NEWTYPE, IERROR

```

The following function is deprecated and is superseded by MPI\_TYPE\_CREATE\_STRUCT in MPI-2.0. The language independent definition and the C binding of the deprecated function is the same as of the new function, except of the function name. Only the Fortran language binding is different.

```

27 MPI_TYPE_STRUCT(count, array_of_blocklengths, array_of_displacements, array_of_types,
28     newtype)
29
30    IN          count          number of blocks (integer) (non-negative integer) –
31                               also number of entries in arrays array_of_types,
32                               array_of_displacements and array_of_blocklengths
33
34    IN          array_of_blocklength  number of elements in each block (array of non-negative
35                               integer)
36
37    IN          array_of_displacements  byte displacement of each block (array of integer)
38
39    IN          array_of_types        type of elements in each block (array of handles to
40                               datatype objects)
41
42    OUT         newtype              new datatype (handle)
43
44 int MPI_Type_struct(int count, int *array_of_blocklengths,
45     MPI_Aint *array_of_displacements,
46     MPI_Datatype *array_of_types, MPI_Datatype *newtype)
47
48 MPI_TYPE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
49     ARRAY_OF_TYPES, NEWTYPE, IERROR)
50
51 INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
52     ARRAY_OF_TYPES(*), NEWTYPE, IERROR

```

The following function is deprecated and is superseded by `MPI_GET_ADDRESS` in MPI-2.0. The language independent definition and the C binding of the deprecated function is the same as of the new function, except of the function name. Only the Fortran language binding is different.

`MPI_ADDRESS(location, address)`

IN	location	location in caller memory (choice)
OUT	address	address of location (integer)

`int MPI_Address(void* location, MPI_Aint *address)`

`MPI_ADDRESS(LOCATION, ADDRESS, IERROR)`

`<type> LOCATION(*)`

`INTEGER ADDRESS, IERROR`

The following functions are deprecated and are superseded by `MPI_TYPE_GET_EXTENT` in MPI-2.0.

`MPI_TYPE_EXTENT(datatype, extent)`

IN	datatype	datatype (handle)
OUT	extent	datatype extent (integer)

`int MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent)`

`MPI_TYPE_EXTENT(DATATYPE, EXTENT, IERROR)`

`INTEGER DATATYPE, EXTENT, IERROR`

Returns the extent of a datatype, where extent is as defined on page 21.

The two functions below can be used for finding the lower bound and the upper bound of a datatype.

`MPI_TYPE_LB( datatype, displacement)`

IN	datatype	datatype (handle)
OUT	displacement	displacement of lower bound from origin, in bytes (integer)

`int MPI_Type_lb(MPI_Datatype datatype, MPI_Aint* displacement)`

`MPI_TYPE_LB( DATATYPE, DISPLACEMENT, IERROR)`

`INTEGER DATATYPE, DISPLACEMENT, IERROR`

```

1 MPI_TYPE_UB( datatype, displacement)
2     IN          datatype          datatype (handle)
3     OUT         displacement      displacement of upper bound from origin, in bytes (in-
4                                     teger)
5
6

```

```

7 int MPI_Type_ub(MPI_Datatype datatype, MPI_Aint* displacement)
8

```

```

9 MPI_TYPE_UB( DATATYPE, DISPLACEMENT, IERROR)
10     INTEGER DATATYPE, DISPLACEMENT, IERROR
11

```

The following function is deprecated and is superseded by MPI\_COMM\_CREATE\_KEYVAL in MPI-2.0. The language independent definition of the deprecated function is the same as that of the new function, except for the function name and a different behavior in the C/Fortran language interoperability, see Section 16.3.7 on page 527. The language bindings are modified.

```

17 MPI_KEYVAL_CREATE(copy_fn, delete_fn, keyval, extra_state)
18

```

```

19     IN          copy_fn          Copy callback function for keyval
20     IN          delete_fn       Delete callback function for keyval
21     OUT         keyval          key value for future access (integer)
22     IN          extra_state     Extra state for callback functions
23
24

```

```

25 int MPI_Keyval_create(MPI_Copy_function *copy_fn, MPI_Delete_function
26     *delete_fn, int *keyval, void* extra_state)
27

```

```

28 MPI_KEYVAL_CREATE(COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR)
29     EXTERNAL COPY_FN, DELETE_FN
30     INTEGER KEYVAL, EXTRA_STATE, IERROR
31

```

The copy\_fn function is invoked when a communicator is duplicated by MPI\_COMM\_DUP. copy\_fn should be of type MPI\_Copy\_function, which is defined as follows:

```

34
35 typedef int MPI_Copy_function(MPI_Comm oldcomm, int keyval,
36                               void *extra_state, void *attribute_val_in,
37                               void *attribute_val_out, int *flag)
38

```

A Fortran declaration for such a function is as follows:

```

39 SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
40     ATTRIBUTE_VAL_OUT, FLAG, IERR)
41     INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
42     ATTRIBUTE_VAL_OUT, IERR
43     LOGICAL FLAG
44

```

copy\_fn may be specified as MPI\_NULL\_COPY\_FN or MPI\_DUP\_FN from either C or FORTRAN; MPI\_NULL\_COPY\_FN is a function that does nothing other than returning flag = 0 and MPI\_SUCCESS. MPI\_DUP\_FN is a simple-minded copy function that sets flag =

1, returns the value of `attribute_val_in` in `attribute_val_out`, and returns `MPI_SUCCESS`. Note that `MPI_NULL_COPY_FN` and `MPI_DUP_FN` are also deprecated.

Analogous to `copy_fn` is a callback deletion function, defined as follows. The `delete_fn` function is invoked when a communicator is deleted by `MPI_COMM_FREE` or when a call is made explicitly to `MPI_ATTR_DELETE`. `delete_fn` should be of type `MPI_Delete_function`, which is defined as follows:

```
typedef int MPI_Delete_function(MPI_Comm comm, int keyval,
void *attribute_val, void *extra_state);
```

A Fortran declaration for such a function is as follows:

```
SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR)
  INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR
```

`delete_fn` may be specified as `MPI_NULL_DELETE_FN` from either C or FORTRAN; `MPI_NULL_DELETE_FN` is a function that does nothing, other than returning `MPI_SUCCESS`. Note that `MPI_NULL_DELETE_FN` is also deprecated.

The following function is deprecated and is superseded by `MPI_COMM_FREE_KEYVAL` in MPI-2.0. The language independent definition of the deprecated function is the same as of the new function, except of the function name. The language bindings are modified.

`MPI_KEYVAL_FREE(keyval)`

INOUT	keyval	Frees the integer key value (integer)
-------	--------	---------------------------------------

```
int MPI_Keyval_free(int *keyval)
```

```
MPI_KEYVAL_FREE(KEYVAL, IERROR)
```

```
  INTEGER KEYVAL, IERROR
```

The following function is deprecated and is superseded by `MPI_COMM_SET_ATTR` in MPI-2.0. The language independent definition of the deprecated function is the same as of the new function, except of the function name. The language bindings are modified.

`MPI_ATTR_PUT(comm, keyval, attribute_val)`

INOUT	comm	communicator to which attribute will be attached (handle)
IN	keyval	key value, as returned by MPI_KEYVAL_CREATE (integer)
IN	attribute_val	attribute value

```
int MPI_Attr_put(MPI_Comm comm, int keyval, void* attribute_val)
```

```
MPI_ATTR_PUT(COMM, KEYVAL, ATTRIBUTE_VAL, IERROR)
```

```
  INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR
```

The following function is deprecated and is superseded by `MPI_COMM_GET_ATTR` in MPI-2.0. The language independent definition of the deprecated function is the same as of the new function, except of the function name. The language bindings are modified.

1 MPI\_ATTR\_GET(comm, keyval, attribute\_val, flag)

2	IN	comm	communicator to which attribute is attached (handle)
3			
4	IN	keyval	key value (integer)
5	OUT	attribute_val	attribute value, unless flag = false
6	OUT	flag	true if an attribute value was extracted; false if no attribute is associated with the key
7			

9  
10 int MPI\_Attr\_get(MPI\_Comm comm, int keyval, void \*attribute\_val, int \*flag)

11 MPI\_ATTR\_GET(COMM, KEYVAL, ATTRIBUTE\_VAL, FLAG, IERROR)

12 INTEGER COMM, KEYVAL, ATTRIBUTE\_VAL, IERROR

13 LOGICAL FLAG

14  
15 The following function is deprecated and is superseded by MPI\_COMM\_DELETE\_ATTR  
16 in MPI-2.0. The language independent definition of the deprecated function is the same as  
17 of the new function, except of the function name. The language bindings are modified.

18  
19 MPI\_ATTR\_DELETE(comm, keyval)

20	INOUT	comm	communicator to which attribute is attached (handle)
21			
22	IN	keyval	The key value of the deleted attribute (integer)
23			

24 int MPI\_Attr\_delete(MPI\_Comm comm, int keyval)

25 MPI\_ATTR\_DELETE(COMM, KEYVAL, IERROR)

26 INTEGER COMM, KEYVAL, IERROR

27  
28 The following function is deprecated and is superseded by  
29 MPI\_COMM\_CREATE\_ERRHANDLER in MPI-2.0. The language independent definition  
30 of the deprecated function is the same as of the new function, except of the function name.  
31 The language bindings are modified.

32  
33 MPI\_ERRHANDLER\_CREATE( function, errhandler )

35	IN	function	user defined error handling procedure
36			
37	OUT	errhandler	MPI error handler (handle)
38			

39 int MPI\_Errhandler\_create(MPI\_Handler\_function \*function,  
40 MPI\_Errhandler \*errhandler)

41 MPI\_ERRHANDLER\_CREATE(FUNCTION, ERRHANDLER, IERROR)

42 EXTERNAL FUNCTION

43 INTEGER ERRHANDLER, IERROR

44  
45 Register the user routine function for use as an MPI exception handler. Returns in  
46 errhandler a handle to the registered exception handler.

47 In the C language, the user routine should be a C function of type MPI\_Handler\_function,  
48 which is defined as:



```
typedef void (MPI_Handler_function)(MPI_Comm *, int *, ...);
```

The first argument is the communicator in use, the second is the error code to be returned.

In the Fortran language, the user routine should be of the form:

```
SUBROUTINE HANDLER_FUNCTION(COMM, ERROR_CODE)
  INTEGER COMM, ERROR_CODE
```

The following function is deprecated and is superseded by `MPI_COMM_SET_ERRHANDLER` in MPI-2.0. The language independent definition of the deprecated function is the same as of the new function, except of the function name. The language bindings are modified.

```
MPI_ERRHANDLER_SET( comm, errhandler )
```

INOUT	comm	communicator to set the error handler for (handle)
IN	errhandler	new MPI error handler for communicator (handle)

```
int MPI_Errhandler_set(MPI_Comm comm, MPI_Errhandler errhandler)
```

```
MPI_ERRHANDLER_SET(COMM, ERRHANDLER, IERROR)
  INTEGER COMM, ERRHANDLER, IERROR
```

Associates the new error handler `errorhandler` with communicator `comm` at the calling process. Note that an error handler is always associated with the communicator.

The following function is deprecated and is superseded by `MPI_COMM_GET_ERRHANDLER` in MPI-2.0. The language independent definition of the deprecated function is the same as of the new function, except of the function name. The language bindings are modified.

```
MPI_ERRHANDLER_GET( comm, errhandler )
```

IN	comm	communicator to get the error handler from (handle)
OUT	errhandler	MPI error handler currently associated with communicator (handle)

```
int MPI_Errhandler_get(MPI_Comm comm, MPI_Errhandler *errhandler)
```

```
MPI_ERRHANDLER_GET(COMM, ERRHANDLER, IERROR)
  INTEGER COMM, ERRHANDLER, IERROR
```

Returns in `errhandler` (a handle to) the error handler that is currently associated with communicator `comm`.

## 13.2 Deprecated since MPI-2.2

The entire set of C++ language bindings have been deprecated.

*Rationale.* The C++ bindings add minimal functionality over the C bindings while incurring a significant amount of maintenance to the MPI specification. Since the C++ bindings are effectively a one-to-one mapping of the C bindings, it should be relatively easy to convert existing C++ MPI applications to use the MPI C bindings. Additionally, there are third party packages available that provide C++ class library functionality (i.e., C++-specific functionality layered on top of the MPI C bindings) that are likely more expressive and/or natural to C++ programmers and are not suitable for standardization in this specification. (*End of rationale.*)

The following function typedefs have been deprecated and are superseded by new names. Other than the typedef names, the function signatures are exactly the same; the names were updated to match conventions of other function typedef names.

Deprecated Name	New Name
<code>MPI_Comm_errhandler_fn</code>	<code>MPI_Comm_errhandler_function</code>
<code>MPI::Comm::Errhandler_fn</code>	<code>MPI::Comm::Errhandler_function</code>
<code>MPI_File_errhandler_fn</code>	<code>MPI_File_errhandler_function</code>
<code>MPI::File::Errhandler_fn</code>	<code>MPI::File::Errhandler_function</code>
<code>MPI_Win_errhandler_fn</code>	<code>MPI_Win_errhandler_function</code>
<code>MPI::Win::Errhandler_fn</code>	<code>MPI::Win::Errhandler_function</code>

## 13.3 Deprecated since MPI-3.0

### 13.3.1 Split Collective Data Access Routines

The entire split collective data access routines are deprecated and superseded by the immediate versions of the nonblocking collective I/O interfaces.

MPI provides a restricted form of “nonblocking collective” I/O operations for all data accesses using split collective data access routines. These routines are referred to as “split” collective routines because a single collective operation is split in two: a begin routine and an end routine. The begin routine begins the operation, much like a nonblocking data access (e.g., `MPI_FILE_IREAD`). The end routine completes the operation, much like the matching test or wait (e.g., `MPI_WAIT`). As with nonblocking data access operations, the user must not use the buffer passed to a begin routine while the routine is outstanding; the operation must be completed with an end routine before it is safe to free buffers, etc.

Split collective data access operations on a file handle `fh` are subject to the semantic rules given below.

- On any MPI process, each file handle may have at most one active split collective operation at any time.
- Begin calls are collective over the group of processes that participated in the collective open and follow the ordering rules for collective calls.
- End calls are collective over the group of processes that participated in the collective open and follow the ordering rules for collective calls. Each end call matches the preceding begin call for the same collective operation. When an “end” call is made, exactly one unmatched “begin” call for the same operation must precede it.

- An implementation is free to implement any split collective data access routine using the corresponding blocking collective routine when either the begin call (e.g., `MPI_FILE_READ_ALL_BEGIN`) or the end call (e.g., `MPI_FILE_READ_ALL_END`) is issued. The begin and end calls are provided to allow the user and MPI implementation to optimize the collective operation.
- Split collective operations do not match the corresponding regular collective operation. For example, in a single collective read operation, an `MPI_FILE_READ_ALL` on one process does not match an `MPI_FILE_READ_ALL_BEGIN`/`MPI_FILE_READ_ALL_END` pair on another process.
- Split collective routines must specify a buffer in both the begin and end routines. By specifying the buffer that receives data in the end routine, we can avoid many (though not all) of the problems described in “A Problem with Register Optimization,” Section 16.2.2, page 507.
- No collective I/O operations are permitted on a file handle concurrently with a split collective access on that file handle (i.e., between the begin and end of the access). That is

```

MPI_File_read_all_begin(fh, ...);
...
MPI_File_read_all(fh, ...);
...
MPI_File_read_all_end(fh, ...);

```

is erroneous.

- In a multithreaded implementation, any split collective begin and end operation called by a process must be called from the same thread. This restriction is made to simplify the implementation in the multithreaded case. (Note that we have already disallowed having two threads begin a split collective operation on the same file handle since only one split collective operation can be active on a file handle at any time.)

The arguments for these routines have the same meaning as for the equivalent collective versions (e.g., the argument definitions for `MPI_FILE_READ_ALL_BEGIN` and `MPI_FILE_READ_ALL_END` are equivalent to the arguments for `MPI_FILE_READ_ALL`). The begin routine (e.g., `MPI_FILE_READ_ALL_BEGIN`) begins a split collective operation that, when completed with the matching end routine (i.e., `MPI_FILE_READ_ALL_END`) produces the result as defined for the equivalent collective routine (i.e., `MPI_FILE_READ_ALL`).

For the purpose of consistency semantics (Section ??, page ??), a matched pair of split collective data access operations (e.g., `MPI_FILE_READ_ALL_BEGIN` and `MPI_FILE_READ_ALL_END`) compose a single data access.

```

1  MPI_FILE_READ_AT_ALL_BEGIN(fh, offset, buf, count, datatype)
2      IN      fh      file handle (handle)
3      IN      offset   file offset (integer)
4      OUT     buf      initial address of buffer (choice)
5      IN      count    number of elements in buffer (integer)
6      IN      datatype datatype of each buffer element (handle)
7
8
9
10 int MPI_File_read_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf,
11                               int count, MPI_Datatype datatype)
12
13 MPI_FILE_READ_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)
14     <type> BUF(*)
15     INTEGER FH, COUNT, DATATYPE, IERROR
16     INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
17
18 {void MPI::File::Read_at_all_begin(MPI::Offset offset, void* buf,
19     int count, const MPI::Datatype& datatype) (binding deprecated, see
20     Section 13.2) }
21
22 MPI_FILE_READ_AT_ALL_END(fh, buf, status)
23     IN      fh      file handle (handle)
24     OUT     buf      initial address of buffer (choice)
25     OUT     status   status object (Status)
26
27
28 int MPI_File_read_at_all_end(MPI_File fh, void *buf, MPI_Status *status)
29
30 MPI_FILE_READ_AT_ALL_END(FH, BUF, STATUS, IERROR)
31     <type> BUF(*)
32     INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
33
34 {void MPI::File::Read_at_all_end(void* buf, MPI::Status& status) (binding
35     deprecated, see Section 13.2) }
36
37 {void MPI::File::Read_at_all_end(void* buf) (binding deprecated, see Section 13.2)
38     }
39
40 MPI_FILE_WRITE_AT_ALL_BEGIN(fh, offset, buf, count, datatype)
41     INOUT    fh      file handle (handle)
42     IN      offset   file offset (integer)
43     IN      buf      initial address of buffer (choice)
44     IN      count    number of elements in buffer (integer)
45     IN      datatype datatype of each buffer element (handle)
46
47
48

```

```

int MPI_File_write_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf,
                                int count, MPI_Datatype datatype)
MPI_FILE_WRITE_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
{void MPI::File::Write_at_all_begin(MPI::Offset offset, const void* buf,
    int count, const MPI::Datatype& datatype) (binding deprecated, see
    Section 13.2) }

MPI_FILE_WRITE_AT_ALL_END(fh, buf, status)
    INOUT    fh                file handle (handle)
    IN        buf              initial address of buffer (choice)
    OUT       status           status object (Status)

int MPI_File_write_at_all_end(MPI_File fh, void *buf, MPI_Status *status)
MPI_FILE_WRITE_AT_ALL_END(FH, BUF, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
{void MPI::File::Write_at_all_end(const void* buf,
    MPI::Status& status) (binding deprecated, see Section 13.2) }
{void MPI::File::Write_at_all_end(const void* buf) (binding deprecated, see
    Section 13.2) }

MPI_FILE_READ_ALL_BEGIN(fh, buf, count, datatype)
    INOUT    fh                file handle (handle)
    OUT      buf              initial address of buffer (choice)
    IN       count            number of elements in buffer (integer)
    IN       datatype         datatype of each buffer element (handle)

int MPI_File_read_all_begin(MPI_File fh, void *buf, int count,
                             MPI_Datatype datatype)
MPI_FILE_READ_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, IERROR
{void MPI::File::Read_all_begin(void* buf, int count,
    const MPI::Datatype& datatype) (binding deprecated, see Section 13.2)
    }

```

```

1  MPI_FILE_READ_ALL_END(fh, buf, status)
2      INOUT    fh                file handle (handle)
3      OUT      buf                initial address of buffer (choice)
4      OUT      status            status object (Status)
5
6
7  int MPI_File_read_all_end(MPI_File fh, void *buf, MPI_Status *status)
8
9  MPI_FILE_READ_ALL_END(FH, BUF, STATUS, IERROR)
10     <type> BUF(*)
11     INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
12
13     {void MPI::File::Read_all_end(void* buf, MPI::Status& status) (binding
14         deprecated, see Section 13.2) }
15
16     {void MPI::File::Read_all_end(void* buf) (binding deprecated, see Section 13.2) }
17
18  MPI_FILE_WRITE_ALL_BEGIN(fh, buf, count, datatype)
19      INOUT    fh                file handle (handle)
20      IN       buf                initial address of buffer (choice)
21      IN       count              number of elements in buffer (integer)
22      IN       datatype            datatype of each buffer element (handle)
23
24
25  int MPI_File_write_all_begin(MPI_File fh, void *buf, int count,
26      MPI_Datatype datatype)
27
28  MPI_FILE_WRITE_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
29     <type> BUF(*)
30     INTEGER FH, COUNT, DATATYPE, IERROR
31
32     {void MPI::File::Write_all_begin(const void* buf, int count,
33         const MPI::Datatype& datatype) (binding deprecated, see Section 13.2)
34         }
35
36
37  MPI_FILE_WRITE_ALL_END(fh, buf, status)
38      INOUT    fh                file handle (handle)
39      IN       buf                initial address of buffer (choice)
40      OUT      status            status object (Status)
41
42
43  int MPI_File_write_all_end(MPI_File fh, void *buf, MPI_Status *status)
44
45  MPI_FILE_WRITE_ALL_END(FH, BUF, STATUS, IERROR)
46     <type> BUF(*)
47     INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
48

```

```
{void MPI::File::Write_all_end(const void* buf, MPI::Status& status) (binding
    deprecated, see Section 13.2) }
```

```
{void MPI::File::Write_all_end(const void* buf) (binding deprecated, see
    Section 13.2) }
```

MPI\_FILE\_READ\_ORDERED\_BEGIN(fh, buf, count, datatype)

INOUT	fh	file handle (handle)
OUT	buf	initial address of buffer (choice)
IN	count	number of elements in buffer (integer)
IN	datatype	datatype of each buffer element (handle)

```
int MPI_File_read_ordered_begin(MPI_File fh, void *buf, int count,
    MPI_Datatype datatype)
```

MPI\_FILE\_READ\_ORDERED\_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)

```
<type> BUF(*)
INTEGER FH, COUNT, DATATYPE, IERROR
```

```
{void MPI::File::Read_ordered_begin(void* buf, int count,
    const MPI::Datatype& datatype) (binding deprecated, see Section 13.2)
}
```

MPI\_FILE\_READ\_ORDERED\_END(fh, buf, status)

INOUT	fh	file handle (handle)
OUT	buf	initial address of buffer (choice)
OUT	status	status object (Status)

```
int MPI_File_read_ordered_end(MPI_File fh, void *buf, MPI_Status *status)
```

MPI\_FILE\_READ\_ORDERED\_END(FH, BUF, STATUS, IERROR)

```
<type> BUF(*)
INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
```

```
{void MPI::File::Read_ordered_end(void* buf, MPI::Status& status) (binding
    deprecated, see Section 13.2) }
```

```
{void MPI::File::Read_ordered_end(void* buf) (binding deprecated, see Section 13.2)
}
```

```

1  MPI_FILE_WRITE_ORDERED_BEGIN(fh, buf, count, datatype)
2      INOUT    fh                file handle (handle)
3      IN       buf                initial address of buffer (choice)
4      IN       count              number of elements in buffer (integer)
5      IN       datatype            datatype of each buffer element (handle)
6
7
8
9  int MPI_File_write_ordered_begin(MPI_File fh, void *buf, int count,
10                                  MPI_Datatype datatype)
11
12  MPI_FILE_WRITE_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
13      <type> BUF(*)
14      INTEGER FH, COUNT, DATATYPE, IERROR
15
16  {void MPI::File::Write_ordered_begin(const void* buf, int count,
17      const MPI::Datatype& datatype) (binding deprecated, see Section 13.2)
18      }
19
20  MPI_FILE_WRITE_ORDERED_END(fh, buf, status)
21      INOUT    fh                file handle (handle)
22      IN       buf                initial address of buffer (choice)
23      OUT      status              status object (Status)
24
25
26  int MPI_File_write_ordered_end(MPI_File fh, void *buf, MPI_Status *status)
27
28  MPI_FILE_WRITE_ORDERED_END(FH, BUF, STATUS, IERROR)
29      <type> BUF(*)
30      INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
31
32  {void MPI::File::Write_ordered_end(const void* buf,
33      MPI::Status& status) (binding deprecated, see Section 13.2) }
34
35  {void MPI::File::Write_ordered_end(const void* buf) (binding deprecated, see
36      Section 13.2) }
37
38
39
40
41
42
43
44
45
46
47
48

```



# Index

CONST:flag = 0, [4](#)  
CONST:flag = 1, [5](#)  
CONST:MPI::Aint, [1–4](#)  
CONST:MPI::Errhandler, [6, 7](#)  
CONST:MPI::File, [10–14](#)  
CONST:MPI::Offset, [10](#)  
CONST:MPI::Status, [10–14](#)  
CONST:MPI\_Aint, [1–4](#)  
CONST:MPI\_Errhandler, [6, 7](#)  
CONST:MPI\_File, [10–14](#)  
CONST:MPI\_Offset, [10](#)  
CONST:MPI\_Status, [10–14](#)  
CONST:MPI\_SUCCESS, [4, 5](#)  
copy\_fn, [4, 5](#)  
delete\_fn, [5](#)  
MPI\_ADDRESS(location, address), [3](#)  
MPI\_ATTR\_DELETE, [5](#)  
MPI\_ATTR\_DELETE(comm, keyval), [6](#)  
MPI\_ATTR\_GET(comm, keyval, attribute\_val,  
flag), [6](#)  
MPI\_ATTR\_PUT(comm, keyval, attribute\_val),  
[5](#)  
MPI\_COMM\_CREATE\_ERRHANDLER, [6](#)  
MPI\_COMM\_CREATE\_KEYVAL, [4](#)  
MPI\_COMM\_DELETE\_ATTR, [6](#)  
MPI\_COMM\_DUP, [4](#)  
MPI\_COMM\_FREE, [5](#)  
MPI\_COMM\_FREE\_KEYVAL, [5](#)  
MPI\_COMM\_GET\_ATTR, [5](#)  
MPI\_COMM\_GET\_ERRHANDLER, [7](#)  
MPI\_COMM\_SET\_ATTR, [5](#)  
MPI\_COMM\_SET\_ERRHANDLER, [7](#)  
MPI\_DUP\_FN, [4, 5](#)  
MPI\_ERRHANDLER\_CREATE( function, er-  
rhander ), [6](#)  
MPI\_ERRHANDLER\_GET( comm, errhan-  
dler ), [7](#)  
MPI\_ERRHANDLER\_SET( comm, errhan-  
dler ), [7](#)  
MPI\_FILE\_IREAD, [8](#)  
MPI\_FILE\_READ\_ALL, [9](#)  
MPI\_FILE\_READ\_ALL\_BEGIN, [9](#)  
MPI\_FILE\_READ\_ALL\_BEGIN(fh, buf, count,  
datatype), [11](#)  
MPI\_FILE\_READ\_ALL\_END, [9](#)  
MPI\_FILE\_READ\_ALL\_END(fh, buf, sta-  
tus), [12](#)  
MPI\_FILE\_READ\_AT\_ALL\_BEGIN(fh, off-  
set, buf, count, datatype), [10](#)  
MPI\_FILE\_READ\_AT\_ALL\_END(fh, buf, sta-  
tus), [10](#)  
MPI\_FILE\_READ\_ORDERED\_BEGIN(fh, buf,  
count, datatype), [13](#)  
MPI\_FILE\_READ\_ORDERED\_END(fh, buf,  
status), [13](#)  
MPI\_FILE\_WRITE\_ALL\_BEGIN(fh, buf, count,  
datatype), [12](#)  
MPI\_FILE\_WRITE\_ALL\_END(fh, buf, sta-  
tus), [12](#)  
MPI\_FILE\_WRITE\_AT\_ALL\_BEGIN(fh, off-  
set, buf, count, datatype), [10](#)  
MPI\_FILE\_WRITE\_AT\_ALL\_END(fh, buf,  
status), [11](#)  
MPI\_FILE\_WRITE\_ORDERED\_BEGIN(fh,  
buf, count, datatype), [14](#)  
MPI\_FILE\_WRITE\_ORDERED\_END(fh, buf,  
status), [14](#)  
MPI\_GET\_ADDRESS, [3](#)  
MPI\_KEYVAL\_CREATE, [5](#)  
MPI\_KEYVAL\_CREATE(copy\_fn, delete\_fn,  
keyval, extra\_state), [4](#)  
MPI\_KEYVAL\_FREE(keyval), [5](#)  
MPI\_NULL\_COPY\_FN, [4, 5](#)  
MPI\_NULL\_DELETE\_FN, [5](#)  
MPI\_TYPE\_CREATE\_HINDEXED, [1](#)  
MPI\_TYPE\_CREATE\_HVECTOR, [1](#)  
MPI\_TYPE\_CREATE\_STRUCT, [2](#)  
MPI\_TYPE\_EXTENT(datatype, extent), [3](#)  
MPI\_TYPE\_GET\_EXTENT, [3](#)

1 MPI\_TYPE\_HINDEXED( count, array\_of\_blocklengths,  
 2 array\_of\_displacements, oldtype, new-  
 3 type), [2](#)  
 4 MPI\_TYPE\_HVECTOR( count, blocklength,  
 5 stride, oldtype, newtype), [1](#)  
 6 MPI\_TYPE\_LB( datatype, displacement), [3](#)  
 7 MPI\_TYPE\_STRUCT(count, array\_of\_blocklengths,  
 8 array\_of\_displacements, array\_of\_types,  
 9 newtype), [2](#)  
 10 MPI\_TYPE\_UB( datatype, displacement), [4](#)  
 11 MPI\_WAIT, [8](#)  
 12  
 13 TYPEDEF:MPI\_Comm\_errhandler\_fn, [8](#)  
 14 TYPEDEF:MPI\_Comm\_errhandler\_function,  
 15 [8](#)  
 16 TYPEDEF:MPI\_Copy\_function, [4](#)  
 17 TYPEDEF:MPI\_Delete\_function, [5](#)  
 18 TYPEDEF:MPI\_File\_errhandler\_fn, [8](#)  
 19 TYPEDEF:MPI\_File\_errhandler\_function, [8](#)  
 20 TYPEDEF:MPI\_Handler\_function, [6](#)  
 21 TYPEDEF:MPI\_Win\_errhandler\_fn, [8](#)  
 22 TYPEDEF:MPI\_Win\_errhandler\_function, [8](#)