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The following function is deprecated and is superseded by MPI_TYPE_CREATE_HVECTOR in MPI-2.0.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The following function is deprecated and is superseded by MPI_TYPE_CREATE_HINDEXED in MPI-2.0.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.

MPI_TYPE_HINDEXED(count, array_of_blocklengths, array_of_displacements, oldtype, newtype)

IN	count	number of blocks – also number of entries in array_of_displacements and array_of_blocklengths (non-negative integer)
IN	array_of_blocklengths	number of elements in each block (array of non-negative integers)
IN	array_of_displacements	byte displacement of each block (array of integer)
IN	oldtype	old datatype (handle)
OUT	newtype	new datatype (handle)

```
int MPI_Type_hindexed(int count, int *array_of_blocklengths,
             MPI_Aint *array_of_displacements, MPI_Datatype oldtype,
             MPI_Datatype *newtype)
```

```
MPI_TYPE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
             OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
    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The following function is deprecated and is superseded by MPI_TYPE_CREATE_STRUCT in MPI-2.0.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.

MPI_TYPE_STRUCT(count, array_of_blocklengths, array_of_displacements, array_of_types, newtype)

IN	count	number of blocks (integer) (non-negative integer) – also number of entries in arrays array_of_types, array_of_displacements and array_of_blocklengths
IN	array_of_blocklength	number of elements in each block (array of non-negative integer)
IN	array_of_displacements	byte displacement of each block (array of integer)
IN	array_of_types	type of elements in each block (array of handles to datatype objects)
OUT	newtype	new datatype (handle)

```
int MPI_Type_struct(int count, int *array_of_blocklengths,
             MPI_Aint *array_of_displacements,
             MPI_Datatype *array_of_types, MPI_Datatype *newtype)
```

```
MPI_TYPE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
             ARRAY_OF_TYPES, NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
    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
```

MPI_TYPE_UB(datatype, displacement)

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

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

MPI_TYPE_UB( DATATYPE, DISPLACEMENT, IERROR)
    INTEGER DATATYPE, DISPLACEMENT, IERROR
```

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.

MPI_KEYVAL_CREATE(copy_fn, delete_fn, keyval, extra_state)

IN	copy_fn	Copy callback function for keyval
IN	delete_fn	Delete callback function for keyval
OUT	keyval	key value for future access (integer)
IN	extra_state	Extra state for callback functions

```
int MPI_Keyval_create(MPI_Copy_function *copy_fn, MPI_Delete_function
             *delete_fn, int *keyval, void* extra_state)

MPI_KEYVAL_CREATE(COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR)
    EXTERNAL COPY_FN, DELETE_FN
    INTEGER KEYVAL, EXTRA_STATE, IERROR
```

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:

```
typedef int MPI_Copy_function(MPI_Comm oldcomm, int keyval,
                              void *extra_state, void *attribute_val_in,
                              void *attribute_val_out, int *flag)
```

A Fortran declaration for such a function is as follows:

```
SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
             ATTRIBUTE_VAL_OUT, FLAG, IERR)
    INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
    ATTRIBUTE_VAL_OUT, IERR
    LOGICAL FLAG
```

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.

MPI_ATTR_GET(comm, keyval, attribute_val, flag)

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

```
int MPI_Attr_get(MPI_Comm comm, int keyval, void *attribute_val, int *flag)
```

```
MPI_ATTR_GET(COMM, KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR
    LOGICAL FLAG
```

The following function is deprecated and is superseded by MPI_COMM_DELETE_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_DELETE(comm, keyval)

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

```
int MPI_Attr_delete(MPI_Comm comm, int keyval)
```

```
MPI_ATTR_DELETE(COMM, KEYVAL, IERROR)
    INTEGER COMM, KEYVAL, IERROR
```

The following function is deprecated and is superseded by MPI_COMM_CREATE_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_CREATE(function, errhandler)

IN	function	user defined error handling procedure
OUT	errhandler	MPI error handler (handle)

```
int MPI_Errhandler_create(MPI_Handler_function *function,
             MPI_Errhandler *errhandler)
```

```
MPI_ERRHANDLER_CREATE(FUNCTION, ERRHANDLER, IERROR)
    EXTERNAL FUNCTION
    INTEGER ERRHANDLER, IERROR
```

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

In the C language, the user routine should be a C function of type MPI_Handler_function, 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
MPI_Comm_errhandler_fn	MPI_Comm_errhandler_function
MPI::Comm::Errhandler_fn	MPI::Comm::Errhandler_function
MPI_File_errhandler_fn	MPI_File_errhandler_function
MPI::File::Errhandler_fn	MPI::File::Errhandler_function
MPI_Win_errhandler_fn	MPI_Win_errhandler_function
MPI::Win::Errhandler_fn	MPI::Win:::Errhandler_function

ticket273.

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.

MPI_FILE_READ_AT_ALL_BEGIN(fh, offset, buf, count, datatype)

IN	fh	file handle (handle)
IN	offset	file offset (integer)
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_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf,
              int count, MPI_Datatype datatype)

MPI_FILE_READ_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::Read_at_all_begin(MPI::Offset offset, void* buf,
              int count, const MPI::Datatype& datatype)} (binding deprecated, see Section 13.2)
```

MPI_FILE_READ_AT_ALL_END(fh, buf, status)

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

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

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

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

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

MPI_FILE_WRITE_AT_ALL_BEGIN(fh, offset, buf, count, datatype)

INOUT	fh	file handle (handle)
IN	offset	file offset (integer)
IN	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_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)
              }
```

MPI_FILE_READ_ALL_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_all_end(MPI_File fh, void *buf, MPI_Status *status)

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

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

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

MPI_FILE_WRITE_ALL_BEGIN(fh, buf, count, datatype)

INOUT	fh	file handle (handle)
IN	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_write_all_begin(MPI_File fh, void *buf, int count,
              MPI_Datatype datatype)

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

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

MPI_FILE_WRITE_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_all_end(MPI_File fh, void *buf, MPI_Status *status)

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

{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)
              }
```

MPI_FILE_WRITE_ORDERED_BEGIN(fh, buf, count, datatype)

INOUT	fh	file handle (handle)
IN	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_write_ordered_begin(MPI_File fh, void *buf, int count,
             MPI_Datatype datatype)

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

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

MPI_FILE_WRITE_ORDERED_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_ordered_end(MPI_File fh, void *buf, MPI_Status *status)

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

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

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

Index