this routine with predefined datatypes employed by the user.

Datarep Conversion Functions

```
typedef int MPI_Datarep_conversion_function(void *userbuf,
             MPI_Datatype datatype, int count, void *filebuf,
             MPI_Offset position, void *extra_state);
```

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Datarep_conversion_function(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
```

```
SUBROUTINE DATAREP_CONVERSION_FUNCTION(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
```

The function read_conversion_fn must convert from file data representation to native representation. Before calling this routine, MPI allocates and fills filebuf with count contiguous data items. The type of each data item matches the corresponding entry for the predefined datatype in the type signature of datatype. The function is passed, in extra_state, the argument that was passed to the MPI_REGISTER_DATAREP call. The function must copy all count data items from filebuf to userbuf in the distribution described by datatype, converting each data item from file representation to native representation. datatype will be equivalent to the datatype that the user passed to the read function. If the size of datatype is less than the size of the count data items, the conversion function must treat datatype as being contiguously tiled over the userbuf. The conversion function must begin storing converted data at the location in userbuf specified by position into the (tiled) datatype.

Advice to users. Although the conversion functions have similarities to MPI_PACK and MPI_UNPACK, one should note the differences in the use of the arguments count and position. In the conversion functions, count is a count of data items (i.e., count of typemap entries of datatype), and position is an index into this typemap. In MPI_PACK, incount refers to the number of whole datatypes, and position is a number of bytes. (*End of advice to users.*)

Advice to implementors. A converted read operation could be implemented as follows:

1. Get file extent of all data items
2. Allocate a filebuf large enough to hold all count data items
3. Read data from file into filebuf

4. Call read_conversion_fn to convert data and place it into userbuf
5. Deallocate filebuf

(*End of advice to implementors.*)

If MPI cannot allocate a buffer large enough to hold all the data to be converted from a read operation, it may call the conversion function repeatedly using the same datatype and userbuf, and reading successive chunks of data to be converted in filebuf. For the first call (and in the case when all the data to be converted fits into filebuf), MPI will call the function with position set to zero. Data converted during this call will be stored in the userbuf according to the first count data items in datatype. Then in subsequent calls to the conversion function, MPI will increment the value in position by the count of items converted in the previous call, and the userbuf pointer will be unchanged.

Rationale. Passing the conversion function a position and one datatype for the transfer allows the conversion function to decode the datatype only once and cache an internal representation of it on the datatype. Then on subsequent calls, the conversion function can use the position to quickly find its place in the datatype and continue storing converted data where it left off at the end of the previous call. (*End of rationale.*)

Advice to users. Although the conversion function may usefully cache an internal representation on the datatype, it should not cache any state information specific to an ongoing conversion operation, since it is possible for the same datatype to be used concurrently in multiple conversion operations. (*End of advice to users.*)

The function write_conversion_fn must convert from native representation to file data representation. Before calling this routine, MPI allocates filebuf of a size large enough to hold count contiguous data items. The type of each data item matches the corresponding entry for the predefined datatype in the type signature of datatype. The function must copy count data items from userbuf in the distribution described by datatype, to a contiguous distribution in filebuf, converting each data item from native representation to file representation. If the size of datatype is less than the size of count data items, the conversion function must treat datatype as being contiguously tiled over the userbuf.

The function must begin copying at the location in userbuf specified by position into the (tiled) datatype. datatype will be equivalent to the datatype that the user passed to the write function. The function is passed, in extra_state, the argument that was passed to the MPI_REGISTER_DATAREP call.

The predefined constant MPI_CONVERSION_FN_NULL may be used as either write_conversion_fn or read_conversion_fn. In that case, MPI will not attempt to invoke write_conversion_fn or read_conversion_fn, respectively, but will perform the requested data access using the native data representation.

An MPI implementation must ensure that all data accessed is converted, either by using a filebuf large enough to hold all the requested data items or else by making repeated calls to the conversion function with the same datatype argument and appropriate values for position.

An implementation will only invoke the callback routines in this section (read_conversion_fn, write_conversion_fn, and dtype_file_extent_fn) when one of the read or

Topologies
C type: `const int` (or unnamed `enum`)
Fortran type: `INTEGER`
MPI_GRAPH
MPI_CART
MPI_DIST_GRAPH

Predefined functions

C/Fortran name C type / Fortran type with `mpi` module / Fortran type with `mpi_f08` module
MPI_COMM_NULL_COPY_FN `MPI_Comm_copy_attr_function` / `COMM_COPY_ATTR_FUNCTION` / `PROCEDURE(MPI_Comm_copy_attr_function)` [1])
MPI_COMM_DUP_FN `MPI_Comm_copy_attr_function` / `COMM_COPY_ATTR_FUNCTION` / `PROCEDURE(MPI_Comm_copy_attr_function)` [1])
MPI_COMM_NULL_DELETE_FN `MPI_Comm_delete_attr_function` / `COMM_DELETE_ATTR_FUNCTION` / `PROCEDURE(MPI_Comm_delete_attr_function)` [1])
MPI_WIN_NULL_COPY_FN `MPI_Win_copy_attr_function` / `WIN_COPY_ATTR_FUNCTION` / `PROCEDURE(MPI_Win_copy_attr_function)` [1])
MPI_WIN_DUP_FN `MPI_Win_copy_attr_function` / `WIN_COPY_ATTR_FUNCTION` / `PROCEDURE(MPI_Win_copy_attr_function)` [1])
MPI_WIN_NULL_DELETE_FN `MPI_Win_delete_attr_function` / `WIN_DELETE_ATTR_FUNCTION` / `PROCEDURE(MPI_Win_delete_attr_function)` [1])
MPI_TYPE_NULL_COPY_FN `MPI_Type_copy_attr_function` / `TYPE_COPY_ATTR_FUNCTION` / `PROCEDURE(MPI_Type_copy_attr_function)` [1])
MPI_TYPE_DUP_FN `MPI_Type_copy_attr_function` / `TYPE_COPY_ATTR_FUNCTION` / `PROCEDURE(MPI_Type_copy_attr_function)` [1])
MPI_TYPE_NULL_DELETE_FN `MPI_Type_delete_attr_function` / `TYPE_DELETE_ATTR_FUNCTION` / `PROCEDURE(MPI_Type_delete_attr_function)` [1])
MPI_CONVERSION_FN_NULL `MPI_Datarep_conversion_function` / `DATAREP_CONVERSION_FUNCTION` / `PROCEDURE(MPI_Datarep_conversion_function)` [1])

[1] See the advice to implementors (on page 270) and advice to users (on page 270) on the predefined Fortran functions MPI_COMM_NULL_COPY_FN, ... in Section 6.7.2.

```
int MPI_Win_free(MPI_Win *win)

int MPI_Win_get_group(MPI_Win win, MPI_Group *group)

int MPI_Win_get_info(MPI_Win win, MPI_Info *info_used)

int MPI_Win_lock_all(int assert, MPI_Win win)

int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)

int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)

int MPI_Win_set_info(MPI_Win win, MPI_Info info)

int MPI_Win_shared_query(MPI_Win win, int rank, MPI_Aint *size,
             int *disp_unit, void *baseptr)

int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)

int MPI_Win_sync(MPI_Win win)

int MPI_Win_test(MPI_Win win, int *flag)

int MPI_Win_unlock_all(MPI_Win win)

int MPI_Win_unlock(int rank, MPI_Win win)

int MPI_Win_wait(MPI_Win win)
```

A.2.10 External Interfaces C Bindings

```
int MPI_Grequest_complete(MPI_Request request)

int MPI_Grequest_start(MPI_Grequest_query_function *query_fn,
             MPI_Grequest_free_function *free_fn,
             MPI_Grequest_cancel_function *cancel_fn, void *extra_state,
             MPI_Request *request)

int MPI_Init_thread(int *argc, char ***argv, int required, int *provided)

int MPI_Is_thread_main(int *flag)

int MPI_Query_thread(int *provided)

int MPI_Status_set_cancelled(MPI_Status *status, int flag)

int MPI_Status_set_elements(MPI_Status *status, MPI_Datatype datatype,
             int count)

int MPI_Status_set_elements_x(MPI_Status *status, MPI_Datatype datatype,
             MPI_Count count)
```

A.2.11 I/O C Bindings

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

#389

```
    INTEGER, INTENT(IN) ::  count
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Status_set_elements_x(status, datatype, count, ierror)
    TYPE(MPI_Status), INTENT(INOUT) ::  status
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER(KIND = MPI_COUNT_KIND), INTENT(IN) ::  count
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

A.3.11 I/O Fortran 2008 Bindings

#389

```
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)
    TYPE(MPI_File), INTENT(INOUT) ::  fh
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_File_delete(filename, info, ierror)
    CHARACTER(LEN=*), INTENT(IN) ::  filename
    TYPE(MPI_Info), INTENT(IN) ::  info
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_File_get_amode(fh, amode, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    INTEGER, INTENT(OUT) ::  amode
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_File_get_atomicity(fh, flag, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    LOGICAL, INTENT(OUT) ::  flag
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_File_get_byte_offset(fh, offset, disp, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) ::  offset
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) ::  disp
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_File_get_group(fh, group, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    TYPE(MPI_Group), INTENT(OUT) ::  group
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_File_get_info(fh, info_used, ierror)
```

```
    INTEGER REQUEST, IERROR
    EXTERNAL QUERY_FN, FREE_FN, CANCEL_FN
    INTEGER (KIND=MPI_ADDRESS_KIND) EXTRA_STATE

MPI_INIT_THREAD(REQUIRED, PROVIDED, IERROR)
    INTEGER REQUIRED, PROVIDED, IERROR

MPI_IS_THREAD_MAIN(FLAG, IERROR)
    LOGICAL FLAG
    INTEGER IERROR

MPI_QUERY_THREAD(PROVIDED, IERROR)
    INTEGER PROVIDED, IERROR

MPI_STATUS_SET_CANCELLED(STATUS, FLAG, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), IERROR
    LOGICAL FLAG

MPI_STATUS_SET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR

MPI_STATUS_SET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR
    INTEGER (KIND=MPI_COUNT_KIND) COUNT
```

A.4.11 I/O Fortran Bindings

```
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
```

#389

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

MPI_FILE_DELETE(FILENAME, INFO, IERROR)
    CHARACTER*(*) FILENAME
    INTEGER INFO, IERROR

MPI_FILE_GET_AMODE(FH, AMODE, IERROR)
    INTEGER FH, AMODE, IERROR

MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR)
    INTEGER FH, IERROR
    LOGICAL FLAG

MPI_FILE_GET_BYTE_OFFSET(FH, OFFSET, DISP, IERROR)
    INTEGER FH, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, DISP

MPI_FILE_GET_GROUP(FH, GROUP, IERROR)
```

Annex B

Change-Log

This annex summarizes changes from the previous version of the MPI standard to the version presented by this document. Only significant changes (i.e., clarifications and new features) that might either require implementation effort in the MPI libraries or change the understanding of MPI from a user's perspective are presented. Editorial modifications, formatting, typo corrections and minor clarifications are not shown.

B.1 Changes from Version 3.0 to Version 3.1

B.1.1 Fixes to Errata in Previous Versions of MPI

1. Sections 2.6.4 and 4.1.5 on pages 20 and 101.
   MPI-3.0 Sections 2.6.4 and 4.1.5 on pages 19 and 102.
   The use of the intrinsic operators "+" and "-" for absolute addresses is substituted by MPI_AINT_ADD and MPI_AINT_DIFF. InC, they can be implemented as macros.

2. Section 7.6 on page 314.
   MPI-3.0 Sections 7.6, on pages 314.
   In the case of virtual general graph topolgies (created with MPI_CART_CREATE), the use of neighborhood collective communication is restricted to adjacency matrices with the number of edges between any two processes is defined to be the same for both processes (i.e., with a symmetric adjacency matrix).

3. Section 11.2.2 on page 405.
   MPI-3.0 Section 11.2.2 page 407.
   The same_size info key can be used with all window flavors.

4. Section 11.3.4 on page 423.
   MPI-3.0 Section 11.3.4 page 424.
   Origin buffer arguments to MPI_GET_ACCUMULATE are ignored when the MPI_NO_OP operation is used.

5. Section 11.3.4 on page 423.
   MPI-3.0 Section 11.3.4 page 424.
   Clarify the roles of origin, result, and target communication parameters in MPI_GET_ACCUMULATE.

6. Annexes A.2, A.3, and A.4 on pages 690, 712, and 761.
   MPI-3.0 Annexes A.2, A.3, and A.4.
   The predefined callback MPI_CONVERSION_FN_NULL was added to all three annexes. #389

7. Section ?? on page ??.
   MPI-3.0 Section xxx page nnn.
   xxx

B.1.2 Changes in MPI-3.0

1. Section ?? on page ??.
   xxx

B.2 Changes from Version 2.2 to Version 3.0

B.2.1 Fixes to Errata in Previous Versions of MPI

1. Sections 2.6.2 and 2.6.3 on pages 19 and 19, and
   MPI-2.2 Section 2.6.2 on page 17, lines 41-42, Section 2.6.3 on page 18, lines 15-16, and Section 2.6.4 on page 18, lines 40-41.
   This is an MPI-2 erratum: The scope for the reserved prefix `MPI_` and the C++ namespace `MPI` is now any name as originally intended in MPI-1.

2. Sections 3.2.2, 5.9.2, 13.6.2 Table 13.2, and Annex A.1.1 on pages 25, 176, 538, and 667, and
   MPI-2.2 Sections 3.2.2, 5.9.2, 13.5.2 Table 13.2, 16.1.16 Table 16.1, and Annex A.1.1 on pages 27, 164, 433, 472 and 513
   This is an MPI-2.2 erratum: New named predefined datatypes MPI_CXX_BOOL, MPI_CXX_FLOAT_COMPLEX, MPI_CXX_DOUBLE_COMPLEX, and MPI_CXX_LONG_DOUBLE_COMPLEX were added in C and Fortran corresponding to the C++ types `bool`, `std::complex<float>`, `std::complex<double>`, and `std::complex<long double>`. These datatypes also correspond to the deprecated C++ predefined datatypes MPI::BOOL, MPI::COMPLEX, MPI::DOUBLE_COMPLEX, and MPI::LONG_DOUBLE_COMPLEX, which were removed in MPI-3.0. The non-standard C++ types `Complex<...>` were substituted by the standard types `std::complex<...>`.

3. Sections 5.9.2 on pages 176 and MPI-2.2 Section 5.9.2, page 165, line 47.
   This is an MPI-2.2 erratum: MPI_C_COMPLEX was added to the "Complex" reduction group.

4. Section 7.5.5 on page 302, and
   MPI-2.2, Section 7.5.5 on page 257, C++ interface on page 264, line 3.
   This is an MPI-2.2 erratum: The argument rank was removed and in/outdegree are now defined as `int& indegree` and `int& outdegree` in the C++ interface of MPI_DIST_GRAPH_NEIGHBORS_COUNT.

5. Section 13.6.2, Table 13.2 on page 538, and
   MPI-2.2, Section 13.5.3, Table 13.2 on page 433.