

D R A F T

Document for a Standard Message-Passing Interface

Message Passing Interface Forum

October 12, 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

I/O

13.1 Introduction

POSIX provides a model of a widely portable file system, but the portability and optimization needed for parallel I/O cannot be achieved with the POSIX interface.

The significant optimizations required for efficiency (e.g., grouping [7], collective buffering [1, 2, 8, 9, 10], and disk-directed I/O [6]) can only be implemented if the parallel I/O system provides a high-level interface supporting partitioning of file data among processes and a collective interface supporting complete transfers of global data structures between process memories and files. In addition, further efficiencies can be gained via support for asynchronous I/O, strided accesses, and control over physical file layout on storage devices (disks). The I/O environment described in this chapter provides these facilities.

Instead of defining I/O access modes to express the common patterns for accessing a shared file (broadcast, reduction, scatter, gather), we chose another approach in which data partitioning is expressed using derived datatypes. Compared to a limited set of predefined access patterns, this approach has the advantage of added flexibility and expressiveness.

13.1.1 Definitions

file An MPI file is an ordered collection of typed data items. MPI supports random or sequential access to any integral set of these items. A file is opened collectively by a group of processes. All collective I/O calls on a file are collective over this group.

displacement A file *displacement* is an absolute byte position relative to the beginning of a file. The displacement defines the location where a *view* begins. Note that a “file displacement” is distinct from a “typemap displacement.”

etype An *etype* (*elementary* datatype) is the unit of data access and positioning. It can be any MPI predefined or derived datatype. Derived etypes can be constructed using any of the MPI datatype constructor routines, provided all resulting typemap displacements are non-negative and monotonically nondecreasing. Data access is performed in etype units, reading or writing whole data items of type etype. Offsets are expressed as a count of etypes; file pointers point to the beginning of etypes. Depending on context, the term “etype” is used to describe one of three aspects of an elementary datatype: a particular MPI type, a data item of that type, or the extent of that type.

filetype A *filetype* is the basis for partitioning a file among processes and defines a template for accessing the file. A filetype is either a single etype or a derived MPI datatype constructed from multiple instances of the same etype. In addition, the extent of any hole in the filetype must be a multiple of the etype's extent. The displacements in the typemap of the filetype are not required to be distinct, but they must be non-negative and monotonically nondecreasing.

view A *view* defines the current set of data visible and accessible from an open file as an ordered set of etypes. Each process has its own view of the file, defined by three quantities: a displacement, an etype, and a filetype. The pattern described by a filetype is repeated, beginning at the displacement, to define the view. The pattern of repetition is defined to be the same pattern that `MPI_TYPE_CONTIGUOUS` would produce if it were passed the filetype and an arbitrarily large count. Figure 13.1 shows how the tiling works; note that the filetype in this example must have explicit lower and upper bounds set in order for the initial and final holes to be repeated in the view. Views can be changed by the user during program execution. The default view is a linear byte stream (displacement is zero, etype and filetype equal to `MPI_BYTE`).

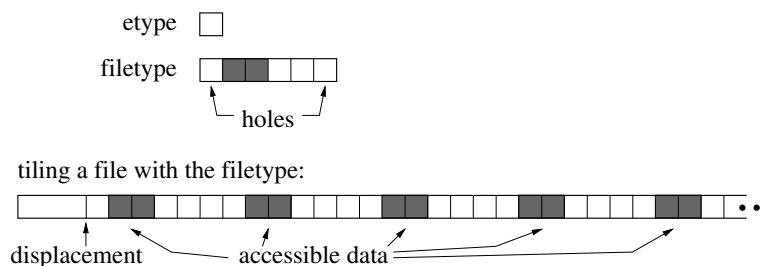


Figure 13.1: Etypes and filetypes

A group of processes can use complementary views to achieve a global data distribution such as a scatter/gather pattern (see Figure 13.2).

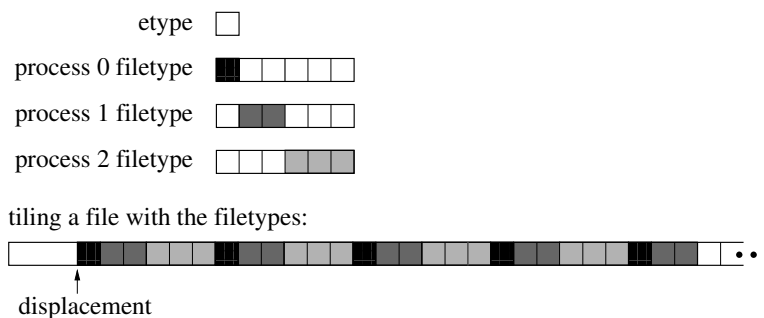


Figure 13.2: Partitioning a file among parallel processes

offset An *offset* is a position in the file relative to the current view, expressed as a count of etypes. Holes in the view's filetype are skipped when calculating this position. Offset 0 is the location of the first etype visible in the view (after skipping the displacement and any initial holes in the view). For example, an offset of 2 for process 1 in Figure 13.2 is the position of the 8th etype in the file after the displacement. An “explicit offset” is an offset that is used as a formal parameter in explicit data access routines.

file size and end of file The *size* of an MPI file is measured in bytes from the beginning of the file. A newly created file has a size of zero bytes. Using the size as an absolute displacement gives the position of the byte immediately following the last byte in the file. For any given view, the *end of file* is the offset of the first etype accessible in the current view starting after the last byte in the file.

file pointer A *file pointer* is an implicit offset maintained by MPI. “Individual file pointers” are file pointers that are local to each process that opened the file. A “shared file pointer” is a file pointer that is shared by the group of processes that opened the file.

file handle A *file handle* is an opaque object created by `MPI_FILE_OPEN` and freed by `MPI_FILE_CLOSE`. All operations on an open file reference the file through the file handle.

13.2 File Manipulation

13.2.1 Opening a File

`MPI_FILE_OPEN(comm, filename, amode, info, fh)`

IN	<code>comm</code>	communicator (handle)
IN	<code>filename</code>	name of file to open (string)
IN	<code>amode</code>	file access mode (integer)
IN	<code>info</code>	info object (handle)
OUT	<code>fh</code>	new file handle (handle)

```
int MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info info,
                  MPI_File *fh)
```

```
MPI_FILE_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERROR)
CHARACTER*(*) FILENAME
INTEGER COMM, AMODE, INFO, FH, IERROR
```

```
{static MPI::File MPI::File::Open(const MPI::Intracomm& comm,
    const char* filename, int amode, const MPI::Info& info) (binding
    deprecated, see Section 15.2) }
```

`MPI_FILE_OPEN` opens the file identified by the file name `filename` on all processes in the `comm` communicator group. `MPI_FILE_OPEN` is a collective routine: all processes must provide the same value for `amode`, and all processes must provide `filenames` that reference the same file. (Values for `info` may vary.) `comm` must be an intracommunicator; it is erroneous to pass an intercommunicator to `MPI_FILE_OPEN`. Errors in `MPI_FILE_OPEN` are raised using the default file error handler (see Section 13.7, page 61). A process can open a file independently of other processes by using the `MPI_COMM_SELF` communicator. The file handle returned, `fh`, can be subsequently used to access the file until the file is closed using `MPI_FILE_CLOSE`. Before calling `MPI_FINALIZE`, the user is required to close (via `MPI_FILE_CLOSE`) all files that were opened with `MPI_FILE_OPEN`. Note that the

communicator `comm` is unaffected by `MPI_FILE_OPEN` and continues to be usable in all MPI routines (e.g., `MPI_SEND`). Furthermore, the use of `comm` will not interfere with I/O behavior.

The format for specifying the file name in the `filename` argument is implementation dependent and must be documented by the implementation.

Advice to implementors. An implementation may require that `filename` include a string or strings specifying additional information about the file. Examples include the type of filesystem (e.g., a prefix of `ufs:`), a remote hostname (e.g., a prefix of `machine.univ.edu:`), or a file password (e.g., a suffix of `/PASSWORD=SECRET`). (*End of advice to implementors.*)

Advice to users. On some implementations of MPI, the file namespace may not be identical from all processes of all applications. For example, `"/tmp/foo"` may denote different files on different processes, or a single file may have many names, dependent on process location. The user is responsible for ensuring that a single file is referenced by the `filename` argument, as it may be impossible for an implementation to detect this type of namespace error. (*End of advice to users.*)

Initially, all processes view the file as a linear byte stream, and each process views data in its own native representation (no data representation conversion is performed). (POSIX files are linear byte streams in the native representation.) The file view can be changed via the `MPI_FILE_SET_VIEW` routine.

The following access modes are supported (specified in `amode`, a bit vector OR of the following integer constants):

- `MPI_MODE_RDONLY` — read only,
- `MPI_MODE_RDWR` — reading and writing,
- `MPI_MODE_WRONLY` — write only,
- `MPI_MODE_CREATE` — create the file if it does not exist,
- `MPI_MODE_EXCL` — error if creating file that already exists,
- `MPI_MODE_DELETE_ON_CLOSE` — delete file on close,
- `MPI_MODE_UNIQUE_OPEN` — file will not be concurrently opened elsewhere,
- `MPI_MODE_SEQUENTIAL` — file will only be accessed sequentially,
- `MPI_MODE_APPEND` — set initial position of all file pointers to end of file.

Advice to users. C/C++ users can use bit vector OR (`|`) to combine these constants; Fortran 90 users can use the bit vector `IOR` intrinsic. Fortran 77 users can use (non-portably) bit vector `IOR` on systems that support it. Alternatively, Fortran users can portably use integer addition to OR the constants (each constant should appear at most once in the addition.). (*End of advice to users.*)

Advice to implementors. The values of these constants must be defined such that the bitwise OR and the sum of any distinct set of these constants is equivalent. (*End of advice to implementors.*)

The modes `MPI_MODE_RDONLY`, `MPI_MODE_RDWR`, `MPI_MODE_WRONLY`, `MPI_MODE_CREATE`, and `MPI_MODE_EXCL` have identical semantics to their POSIX counterparts [5]. Exactly one of `MPI_MODE_RDONLY`, `MPI_MODE_RDWR`, or `MPI_MODE_WRONLY`, must be specified. It is erroneous to specify `MPI_MODE_CREATE` or `MPI_MODE_EXCL` in conjunction with `MPI_MODE_RDONLY`; it is erroneous to specify `MPI_MODE_SEQUENTIAL` together with `MPI_MODE_RDWR`.

The `MPI_MODE_DELETE_ON_CLOSE` mode causes the file to be deleted (equivalent to performing an `MPI_FILE_DELETE`) when the file is closed.

The `MPI_MODE_UNIQUE_OPEN` mode allows an implementation to optimize access by eliminating the overhead of file locking. It is erroneous to open a file in this mode unless the file will not be concurrently opened elsewhere.

Advice to users. For `MPI_MODE_UNIQUE_OPEN`, *not opened elsewhere* includes both inside and outside the MPI environment. In particular, one needs to be aware of potential external events which may open files (e.g., automated backup facilities). When `MPI_MODE_UNIQUE_OPEN` is specified, the user is responsible for ensuring that no such external events take place. (*End of advice to users.*)

The `MPI_MODE_SEQUENTIAL` mode allows an implementation to optimize access to some sequential devices (tapes and network streams). It is erroneous to attempt nonsequential access to a file that has been opened in this mode.

Specifying `MPI_MODE_APPEND` only guarantees that all shared and individual file pointers are positioned at the initial end of file when `MPI_FILE_OPEN` returns. Subsequent positioning of file pointers is application dependent. In particular, the implementation does not ensure that all writes are appended.

Errors related to the access mode are raised in the class `MPI_ERR_AMODE`.

The `info` argument is used to provide information regarding file access patterns and file system specifics (see Section 13.2.8, page 12). The constant `MPI_INFO_NULL` can be used when no `info` needs to be specified.

Advice to users. Some file attributes are inherently implementation dependent (e.g., file permissions). These attributes must be set using either the `info` argument or facilities outside the scope of MPI. (*End of advice to users.*)

Files are opened by default using nonatomic mode file consistency semantics (see Section 13.6.1, page 52). The more stringent atomic mode consistency semantics, required for atomicity of conflicting accesses, can be set using `MPI_FILE_SET_ATOMICITY`.

`MPI_FILE_IOPEN(comm, filename, amode, info, fh, request)`

IN	<code>comm</code>	communicator (handle)
IN	<code>filename</code>	name of file to open (string)
IN	<code>amode</code>	file access mode (integer)
IN	<code>info</code>	info object (handle)
OUT	<code>fh</code>	new file handle (handle)
OUT	<code>request</code>	request object (handle)

```

1  int MPI_File_iopen(MPI_Comm comm, char *filename, int amode, MPI_Info info,
2                      MPI_File *fh, MPI_Request *request)

```

```

3  MPI_FILE_IOPEN(COMM, FILENAME, AMODE, INFO, FH, REQUEST, IERROR)

```

```

4  CHARACTER*(*) FILENAME

```

```

5  INTEGER COMM, AMODE, INFO, FH, REQUEST, IERROR

```

```

6
7  MPI_FILE_IOPEN is a nonblocking version of the MPI_FILE_OPEN interface.

```

```

8  It is erroneous to access the file handle of an MPI_FILE_IOPEN or issue any I/O oper-
9  ation on that file handle, between the initiation and completion of the operation.

```

```

10
11  Advice to users. An MPI implementation may require threading support to imple-
12  ment a nonblocking open operation as well as other file manipulation routines. (End
13  of advice to users.)

```

13.2.2 Closing a File

```

14
15
16  MPI_FILE_CLOSE(fh)

```

```

17      INOUT    fh                      file handle (handle)

```

```

18
19  int MPI_File_close(MPI_File *fh)

```

```

20  MPI_FILE_CLOSE(FH, IERROR)

```

```

21  INTEGER FH, IERROR

```

```

22  {void MPI::File::Close() (binding deprecated, see Section 15.2) }

```

```

23
24  MPI_FILE_CLOSE first synchronizes file state (equivalent to performing an
25  MPI_FILE_SYNC), then closes the file associated with fh. The file is deleted if it was
26  opened with access mode MPI_MODE_DELETE_ON_CLOSE (equivalent to performing an
27  MPI_FILE_DELETE). MPI_FILE_CLOSE is a collective routine.

```

```

28
29  Advice to users. If the file is deleted on close, and there are other processes currently
30  accessing the file, the status of the file and the behavior of future accesses by these
31  processes are implementation dependent. (End of advice to users.)

```

```

32
33  The user is responsible for ensuring that all outstanding nonblocking requests and
34  split collective operations associated with fh made by a process have completed before that
35  process calls MPI_FILE_CLOSE.

```

```

36
37  The MPI_FILE_CLOSE routine deallocates the file handle object and sets fh to
38  MPI_FILE_NULL.

```

```

39
40  MPI_FILE_ICLOSE(fh, request)

```

```

41      INOUT    fh                      file handle (handle)

```

```

42      OUT      request                  request object (handle)

```



```
int MPI_File_icolse(MPI_File *fh, MPI_Request *request)
```

```
MPI_FILE_ICLOSE(FH, REQUEST, IERROR)
```

```
    INTEGER FH, REQUEST, IERROR
```

`MPI_FILE_ICLOSE` is a nonblocking version of the function `MPI_FILE_CLOSE`.

13.2.3 Deleting a File

```
MPI_FILE_DELETE(filename, info)
```

```
    IN      filename      name of file to delete (string)
```

```
    IN      info          info object (handle)
```

```
int MPI_File_delete(char *filename, MPI_Info info)
```

```
MPI_FILE_DELETE(FILENAME, INFO, IERROR)
```

```
    CHARACTER*(*) FILENAME
```

```
    INTEGER INFO, IERROR
```

```
{static void MPI::File::Delete(const char* filename,
                               const MPI::Info& info) (binding deprecated, see Section 15.2) }
```

`MPI_FILE_DELETE` deletes the file identified by the file name `filename`. If the file does not exist, `MPI_FILE_DELETE` raises an error in the class `MPI_ERR_NO_SUCH_FILE`.

The `info` argument can be used to provide information regarding file system specifics (see Section 13.2.8, page 12). The constant `MPI_INFO_NULL` refers to the null info, and can be used when no info needs to be specified.

If a process currently has the file open, the behavior of any access to the file (as well as the behavior of any outstanding accesses) is implementation dependent. In addition, whether an open file is deleted or not is also implementation dependent. If the file is not deleted, an error in the class `MPI_ERR_FILE_IN_USE` or `MPI_ERR_ACCESS` will be raised. Errors are raised using the default error handler (see Section 13.7, page 61).

```
MPI_FILE_IDELETE(filename, info, request)
```

```
    IN      filename      name of file to delete (string)
```

```
    IN      info          info object (handle)
```

```
    OUT     request       request object (handle)
```

```
int MPI_File_idelete(char *filename, MPI_Info info, MPI_Request *request)
```

```
MPI_FILE_IDELETE(FILENAME, INFO, REQUEST, IERROR)
```

```
    CHARACTER*(*) FILENAME
```

```
    INTEGER INFO, REQUEST, IERROR
```

`MPI_FILE_IDELETE` is a nonblocking version of the function `MPI_FILE_DELETE` interface.

13.2.4 Resizing a File

MPI_FILE_SET_SIZE(fh, size)

INOUT	fh	file handle (handle)
IN	size	size to truncate or expand file (integer)

```
int MPI_File_set_size(MPI_File fh, MPI_Offset size)
```

```
MPI_FILE_SET_SIZE(FH, SIZE, IERROR)
```

```
INTEGER FH, IERROR
```

```
INTEGER(KIND=MPI_OFFSET_KIND) SIZE
```

```
{void MPI::File::Set_size(MPI::Offset size) (binding deprecated, see Section 15.2)
}
```

MPI_FILE_SET_SIZE resizes the file associated with the file handle `fh`. `size` is measured in bytes from the beginning of the file. MPI_FILE_SET_SIZE is collective; all processes in the group must pass identical values for `size`.

If `size` is smaller than the current file size, the file is truncated at the position defined by `size`. The implementation is free to deallocate file blocks located beyond this position.

If `size` is larger than the current file size, the file size becomes `size`. Regions of the file that have been previously written are unaffected. The values of data in the new regions in the file (those locations with displacements between old file size and `size`) are undefined. It is implementation dependent whether the MPI_FILE_SET_SIZE routine allocates file space—use MPI_FILE_PREALLOCATE to force file space to be reserved.

MPI_FILE_SET_SIZE does not affect the individual file pointers or the shared file pointer. If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is erroneous to call this routine.

Advice to users. It is possible for the file pointers to point beyond the end of file after a MPI_FILE_SET_SIZE operation truncates a file. This is legal, and equivalent to seeking beyond the current end of file. (*End of advice to users.*)

All nonblocking requests and split collective operations on `fh` must be completed before calling MPI_FILE_SET_SIZE. Otherwise, calling MPI_FILE_SET_SIZE is erroneous. As far as consistency semantics are concerned, MPI_FILE_SET_SIZE is a write operation that conflicts with operations that access bytes at displacements between the old and new file sizes (see Section 13.6.1, page 52).

MPI_FILE_ISET_SIZE(fh, size, request)

INOUT	fh	file handle (handle)
IN	size	size to truncate or expand file (integer)
OUT	request	request object (handle)

```
int MPI_File_iset_size(MPI_File fh, MPI_Offset size, MPI_Request *request)
```

```

MPI_FILE_ISET_SIZE(FH, SIZE, REQUEST, IERROR)
    INTEGER FH, REQUEST, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) SIZE

```

`MPI_FILE_ISET_SIZE` is a nonblocking version of the `MPI_FILE_SET_SIZE` interface. As far as consistency semantics are concerned, similarly to its blocking counterpart, this routine is a write operation that conflicts with any operation that attempts to access bytes between the old and new size displacements.

13.2.5 Preallocating Space for a File

```

MPI_FILE_PREALLOCATE(fh, size)

```

INOUT	fh	file handle (handle)
IN	size	size to preallocate file (integer)

```

int MPI_File_preallocate(MPI_File fh, MPI_Offset size)

```

```

MPI_FILE_PREALLOCATE(FH, SIZE, IERROR)
    INTEGER FH, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) SIZE

```

```

{void MPI::File::Preallocate(MPI::Offset size) (binding deprecated, see
    Section 15.2) }

```

`MPI_FILE_PREALLOCATE` ensures that storage space is allocated for the first `size` bytes of the file associated with `fh`. `MPI_FILE_PREALLOCATE` is collective; all processes in the group must pass identical values for `size`. Regions of the file that have previously been written are unaffected. For newly allocated regions of the file, `MPI_FILE_PREALLOCATE` has the same effect as writing undefined data. If `size` is larger than the current file size, the file size increases to `size`. If `size` is less than or equal to the current file size, the file size is unchanged.

The treatment of file pointers, pending nonblocking accesses, and file consistency is the same as with `MPI_FILE_SET_SIZE`. If `MPI_MODE_SEQUENTIAL` mode was specified when the file was opened, it is erroneous to call this routine.

Advice to users. In some implementations, file preallocation may be expensive. (*End of advice to users.*)

```

MPI_FILE_IPREALLOCATE(fh, size, request)

```

INOUT	fh	file handle (handle)
IN	size	size to preallocate file (integer)
OUT	request	request object (handle)

```

int MPI_File_ipreallocate(MPI_File fh, MPI_Offset size,
    MPI_Request *request)

```

```

1 MPI_FILE_IPREALLOCATE(FH, SIZE, REQUEST, IERROR)
2     INTEGER FH, REQUEST, IERROR
3     INTEGER(KIND=MPI_OFFSET_KIND) SIZE

```

MPI_FILE_IPREALLOCATE is a nonblocking version of the MPI_FILE_PREALLOCATE interface.

13.2.6 Querying the Size of a File

```

10 MPI_FILE_GET_SIZE(fh, size)

```

12	IN	fh	file handle (handle)
13	OUT	size	size of the file in bytes (integer)

```

15 int MPI_File_get_size(MPI_File fh, MPI_Offset *size)

```

```

17 MPI_FILE_GET_SIZE(FH, SIZE, IERROR)
18     INTEGER FH, IERROR
19     INTEGER(KIND=MPI_OFFSET_KIND) SIZE

```

```

20 {MPI::Offset MPI::File::Get_size() const(binding deprecated, see Section 15.2) }

```

MPI_FILE_GET_SIZE returns, in size, the current size in bytes of the file associated with the file handle fh. As far as consistency semantics are concerned, MPI_FILE_GET_SIZE is a data access operation (see Section 13.6.1, page 52).

```

25 MPI_FILE_IGET_SIZE(fh, size, request)

```

28	IN	fh	file handle (handle)
29	OUT	size	size of the file in bytes (integer)
30	OUT	request	request object (handle)

```

32 int MPI_File_iget_size(MPI_File fh, MPI_Offset *size, MPI_Request *request)

```

```

33 MPI_FILE_IGET_SIZE(FH, SIZE, REQUEST, IERROR)
34     INTEGER FH, REQUEST, IERROR
35     INTEGER(KIND=MPI_OFFSET_KIND) SIZE

```

MPI_FILE_IGET_SIZE is a nonblocking version of the MPI_FILE_GET_SIZE interface.

13.2.7 Querying File Parameters

```

40 MPI_FILE_GET_GROUP(fh, group)

```

45	IN	fh	file handle (handle)
46	OUT	group	group which opened the file (handle)

```
int MPI_File_get_group(MPI_File fh, MPI_Group *group)
```

```
MPI_FILE_GET_GROUP(FH, GROUP, IERROR)
```

```
INTEGER FH, GROUP, IERROR
```

```
{MPI::Group MPI::File::Get_group() const(binding deprecated, see Section 15.2) }
```

MPI_FILE_GET_GROUP returns a duplicate of the group of the communicator used to open the file associated with fh. The group is returned in group. The user is responsible for freeing group.

```
MPI_FILE_GET_AMODE(fh, amode)
```

```
IN      fh      file handle (handle)
```

```
OUT     amode    file access mode used to open the file (integer)
```

```
int MPI_File_get_amode(MPI_File fh, int *amode)
```

```
MPI_FILE_GET_AMODE(FH, AMODE, IERROR)
```

```
INTEGER FH, AMODE, IERROR
```

```
{int MPI::File::Get_amode() const(binding deprecated, see Section 15.2) }
```

MPI_FILE_GET_AMODE returns, in amode, the access mode of the file associated with fh.

Example 13.1 In Fortran 77, decoding an amode bit vector will require a routine such as the following:

```
      SUBROUTINE BIT_QUERY(TEST_BIT, MAX_BIT, AMODE, BIT_FOUND)
!
!  TEST IF THE INPUT TEST_BIT IS SET IN THE INPUT AMODE
!  IF SET, RETURN 1 IN BIT_FOUND, 0 OTHERWISE
!
      INTEGER TEST_BIT, AMODE, BIT_FOUND, CP_AMODE, HIFOUND
      BIT_FOUND = 0
      CP_AMODE = AMODE
100  CONTINUE
      LBIT = 0
      HIFOUND = 0
      DO 20 L = MAX_BIT, 0, -1
          MATCHER = 2**L
          IF (CP_AMODE .GE. MATCHER .AND. HIFOUND .EQ. 0) THEN
              HIFOUND = 1
              LBIT = MATCHER
              CP_AMODE = CP_AMODE - MATCHER
          END IF
20  CONTINUE
      IF (HIFOUND .EQ. 1 .AND. LBIT .EQ. TEST_BIT) BIT_FOUND = 1
      IF (BIT_FOUND .EQ. 0 .AND. HIFOUND .EQ. 1 .AND. &
```

```

1      CP_AMODE .GT. 0) GO TO 100
2      END

```

This routine could be called successively to decode `amode`, one bit at a time. For example, the following code fragment would check for `MPI_MODE_RDONLY`.

```

6      CALL BIT_QUERY(MPI_MODE_RDONLY, 30, AMODE, BIT_FOUND)
7      IF (BIT_FOUND .EQ. 1) THEN
8          PRINT *, ' FOUND READ-ONLY BIT IN AMODE=', AMODE
9      ELSE
10         PRINT *, ' READ-ONLY BIT NOT FOUND IN AMODE=', AMODE
11     END IF

```

13.2.8 File Info

Hints specified via info (see Section 9, page 321) allow a user to provide information such as file access patterns and file system specifics to direct optimization. Providing hints may enable an implementation to deliver increased I/O performance or minimize the use of system resources. However, hints do not change the semantics of any of the I/O interfaces. In other words, an implementation is free to ignore all hints. Hints are specified on a per file basis, in `MPI_FILE_OPEN`, `MPI_FILE_DELETE`, `MPI_FILE_SET_VIEW`, and `MPI_FILE_SET_INFO`, via the opaque info object. When an info object that specifies a subset of valid hints is passed to `MPI_FILE_SET_VIEW` or `MPI_FILE_SET_INFO`, there will be no effect on previously set or defaulted hints that the info does not specify.

Advice to implementors. It may happen that a program is coded with hints for one system, and later executes on another system that does not support these hints. In general, unsupported hints should simply be ignored. Needless to say, no hint can be mandatory. However, for each hint used by a specific implementation, a default value must be provided when the user does not specify a value for this hint. (*End of advice to implementors.*)

`MPI_FILE_SET_INFO(fh, info)`

INOUT	fh	file handle (handle)
IN	info	info object (handle)

```
int MPI_File_set_info(MPI_File fh, MPI_Info info)
```

```
MPI_FILE_SET_INFO(FH, INFO, IERROR)
```

```
INTEGER FH, INFO, IERROR
```

```
{void MPI::File::Set_info(const MPI::Info& info) (binding deprecated, see
    Section 15.2) }
```

`MPI_FILE_SET_INFO` sets new values for the hints of the file associated with `fh`. `MPI_FILE_SET_INFO` is a collective routine. The info object may be different on each process, but any info entries that an implementation requires to be the same on all processes must appear with the same value in each process's info object.

Advice to users. Many info items that an implementation can use when it creates or opens a file cannot easily be changed once the file has been created or opened. Thus, an implementation may ignore hints issued in this call that it would have accepted in an open call. (*End of advice to users.*)

`MPI_FILE_GET_INFO(fh, info_used)`

IN	fh	file handle (handle)
OUT	info_used	new info object (handle)

`int MPI_File_get_info(MPI_File fh, MPI_Info *info_used)`

`MPI_FILE_GET_INFO(FH, INFO_USED, IERROR)`

`INTEGER FH, INFO_USED, IERROR`

`{MPI::Info MPI::File::Get_info() const}` (*binding deprecated, see Section 15.2*)

`MPI_FILE_GET_INFO` returns a new info object containing the hints of the file associated with `fh`. The current setting of all hints actually used by the system related to this open file is returned in `info_used`. If no such hints exist, a handle to a newly created info object is returned that contains no key/value pair. The user is responsible for freeing `info_used` via `MPI_INFO_FREE`.

Advice to users. The info object returned in `info_used` will contain all hints currently active for this file. This set of hints may be greater or smaller than the set of hints passed in to `MPI_FILE_OPEN`, `MPI_FILE_SET_VIEW`, and `MPI_FILE_SET_INFO`, as the system may not recognize some hints set by the user, and may recognize other hints that the user has not set. (*End of advice to users.*)

Reserved File Hints

Some potentially useful hints (info key values) are outlined below. The following key values are reserved. An implementation is not required to interpret these key values, but if it does interpret the key value, it must provide the functionality described. (For more details on “info,” see Section 9, page 321.)

These hints mainly affect access patterns and the layout of data on parallel I/O devices. For each hint name introduced, we describe the purpose of the hint, and the type of the hint value. The “[**SAME**]” annotation specifies that the hint values provided by all participating processes must be identical; otherwise the program is erroneous. In addition, some hints are context dependent, and are only used by an implementation at specific times (e.g., `file_perm` is only useful during file creation).

access_style (comma separated list of strings): This hint specifies the manner in which the file will be accessed until the file is closed or until the `access_style` key value is altered. The hint value is a comma separated list of the following: `read_once`, `write_once`, `read_mostly`, `write_mostly`, `sequential`, `reverse_sequential`, and `random`.

collective_buffering (boolean) [SAME**]:** This hint specifies whether the application may benefit from collective buffering. Collective buffering is an optimization performed

on collective accesses. Accesses to the file are performed on behalf of all processes in the group by a number of target nodes. These target nodes coalesce small requests into large disk accesses. Legal values for this key are `true` and `false`. Collective buffering parameters are further directed via additional hints: `cb_block_size`, `cb_buffer_size`, and `cb_nodes`.

cb_block_size (integer) [SAME]: This hint specifies the block size to be used for collective buffering file access. *Target nodes* access data in chunks of this size. The chunks are distributed among target nodes in a round-robin (CYCLIC) pattern.

cb_buffer_size (integer) [SAME]: This hint specifies the total buffer space that can be used for collective buffering on each target node, usually a multiple of `cb_block_size`.

cb_nodes (integer) [SAME]: This hint specifies the number of target nodes to be used for collective buffering.

chunked (comma separated list of integers) [SAME]: This hint specifies that the file consists of a multidimensional array that is often accessed by subarrays. The value for this hint is a comma separated list of array dimensions, starting from the most significant one (for an array stored in row-major order, as in C, the most significant dimension is the first one; for an array stored in column-major order, as in Fortran, the most significant dimension is the last one, and array dimensions should be reversed).

chunked_item (comma separated list of integers) [SAME]: This hint specifies the size of each array entry, in bytes.

chunked_size (comma separated list of integers) [SAME]: This hint specifies the dimensions of the subarrays. This is a comma separated list of array dimensions, starting from the most significant one.

filename (string): This hint specifies the file name used when the file was opened. If the implementation is capable of returning the file name of an open file, it will be returned using this key by `MPI_FILE_GET_INFO`. This key is ignored when passed to `MPI_FILE_OPEN`, `MPI_FILE_SET_VIEW`, `MPI_FILE_SET_INFO`, and `MPI_FILE_DELETE`.

file_perm (string) [SAME]: This hint specifies the file permissions to use for file creation. Setting this hint is only useful when passed to `MPI_FILE_OPEN` with an `amode` that includes `MPI_MODE_CREATE`. The set of legal values for this key is implementation dependent.

io_node_list (comma separated list of strings) [SAME]: This hint specifies the list of I/O devices that should be used to store the file. This hint is most relevant when the file is created.

nb_proc (integer) [SAME]: This hint specifies the number of parallel processes that will typically be assigned to run programs that access this file. This hint is most relevant when the file is created.

num_io_nodes (integer) [SAME]: This hint specifies the number of I/O devices in the system. This hint is most relevant when the file is created.

striping_factor (integer) [SAME]: This hint specifies the number of I/O devices that the file should be striped across, and is relevant only when the file is created.

striping_unit (integer) [SAME]: This hint specifies the suggested striping unit to be used for this file. The striping unit is the amount of consecutive data assigned to one I/O device before progressing to the next device, when striping across a number of devices. It is expressed in bytes. This hint is relevant only when the file is created.

13.3 File Views

MPI_FILE_SET_VIEW(fh, disp, etype, filetype, datarep, info)

INOUT	fh	file handle (handle)
IN	disp	displacement (integer)
IN	etype	elementary datatype (handle)
IN	filetype	filetype (handle)
IN	datarep	data representation (string)
IN	info	info object (handle)

```
int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype,
                     MPI_Datatype filetype, char *datarep, MPI_Info info)
```

```
MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR)
    INTEGER FH, ETYPE, FILETYPE, INFO, IERROR
    CHARACTER*(*) DATAREP
    INTEGER(KIND=MPI_OFFSET_KIND) DISP
```

```
{void MPI::File::Set_view(MPI::Offset disp, const MPI::Datatype& etype,
                          const MPI::Datatype& filetype, const char* datarep,
                          const MPI::Info& info) (binding deprecated, see Section 15.2) }
```

The **MPI_FILE_SET_VIEW** routine changes the process's view of the data in the file. The start of the view is set to **disp**; the type of data is set to **etype**; the distribution of data to processes is set to **filetype**; and the representation of data in the file is set to **datarep**. In addition, **MPI_FILE_SET_VIEW** resets the individual file pointers and the shared file pointer to zero. **MPI_FILE_SET_VIEW** is collective; the values for **datarep** and the extents of **etype** in the file data representation must be identical on all processes in the group; values for **disp**, **filetype**, and **info** may vary. The datatypes passed in **etype** and **filetype** must be committed.

The **etype** always specifies the data layout in the file. If **etype** is a portable datatype (see Section 2.4, page 11), the extent of **etype** is computed by scaling any displacements in the datatype to match the file data representation. If **etype** is not a portable datatype, no scaling is done when computing the extent of **etype**. The user must be careful when using nonportable **etypes** in heterogeneous environments; see Section 13.5.1, page 44 for further details.

If `MPI_MODE_SEQUENTIAL` mode was specified when the file was opened, the special displacement `MPI_DISPLACEMENT_CURRENT` must be passed in `disp`. This sets the displacement to the current position of the shared file pointer. `MPI_DISPLACEMENT_CURRENT` is invalid unless the amode for the file has `MPI_MODE_SEQUENTIAL` set.

Rationale. For some sequential files, such as those corresponding to magnetic tapes or streaming network connections, the *displacement* may not be meaningful. `MPI_DISPLACEMENT_CURRENT` allows the view to be changed for these types of files. (*End of rationale.*)

Advice to implementors. It is expected that a call to `MPI_FILE_SET_VIEW` will immediately follow `MPI_FILE_OPEN` in numerous instances. A high-quality implementation will ensure that this behavior is efficient. (*End of advice to implementors.*)

The `disp` displacement argument specifies the position (absolute offset in bytes from the beginning of the file) where the view begins.

Advice to users. `disp` can be used to skip headers or when the file includes a sequence of data segments that are to be accessed in different patterns (see Figure 13.3). Separate views, each using a different displacement and filetype, can be used to access each segment.

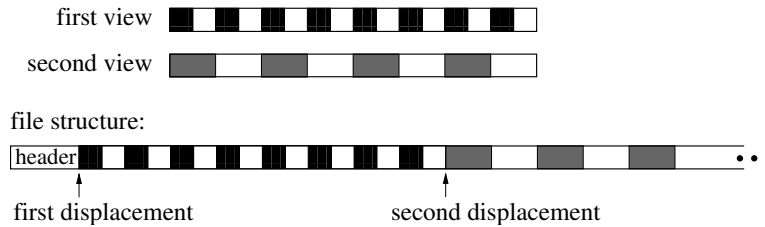


Figure 13.3: Displacements

(*End of advice to users.*)

An *etype* (*elementary* datatype) is the unit of data access and positioning. It can be any MPI predefined or derived datatype. Derived etypes can be constructed by using any of the MPI datatype constructor routines, provided all resulting typemap displacements are non-negative and monotonically nondecreasing. Data access is performed in etype units, reading or writing whole data items of type etype. Offsets are expressed as a count of etypes; file pointers point to the beginning of etypes.

Advice to users. In order to ensure interoperability in a heterogeneous environment, additional restrictions must be observed when constructing the **etype** (see Section 13.5, page 42). (*End of advice to users.*)

A filetype is either a single etype or a derived MPI datatype constructed from multiple instances of the same etype. In addition, the extent of any hole in the filetype must be a multiple of the etype's extent. These displacements are not required to be distinct, but they cannot be negative, and they must be monotonically nondecreasing.

If the file is opened for writing, neither the **etype** nor the **filetype** is permitted to contain overlapping regions. This restriction is equivalent to the “datatype used in a receive cannot

specify overlapping regions” restriction for communication. Note that `filetypes` from different processes may still overlap each other.

If `filetype` has holes in it, then the data in the holes is inaccessible to the calling process. However, the `disp`, `etype` and `filetype` arguments can be changed via future calls to `MPI_FILE_SET_VIEW` to access a different part of the file.

It is erroneous to use absolute addresses in the construction of the `etype` and `filetype`.

The `info` argument is used to provide information regarding file access patterns and file system specifics to direct optimization (see Section 13.2.8, page 12). The constant `MPI_INFO_NULL` refers to the null info and can be used when no info needs to be specified.

The `datarep` argument is a string that specifies the representation of data in the file. See the file interoperability section (Section 13.5, page 42) for details and a discussion of valid values.

The user is responsible for ensuring that all nonblocking requests and split collective operations on `fh` have been completed before calling `MPI_FILE_SET_VIEW`—otherwise, the call to `MPI_FILE_SET_VIEW` is erroneous.

`MPI_FILE_ISET_VIEW(fh, disp, etype, filetype, datarep, info, request)`

INOUT	<code>fh</code>	file handle (handle)
IN	<code>disp</code>	displacement (integer)
IN	<code>etype</code>	elementary datatype (handle)
IN	<code>filetype</code>	filetype (handle)
IN	<code>datarep</code>	data representation (string)
IN	<code>info</code>	info object (handle)
OUT	<code>request</code>	request object (handle)

```
int MPI_File_iset_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype,
                      MPI_Datatype filetype, char *datarep, MPI_Info info,
                      MPI_Request *request)
```

```
MPI_FILE_ISET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, REQUEST,
                   IERROR)
```

```
INTEGER FH, ETYPE, FILETYPE, INFO, REQUEST, IERROR
CHARACTER*(*) DATAREP
INTEGER(KIND=MPI_OFFSET_KIND) DISP
```

The `MPI_FILE_ISET_VIEW` is a nonblocking version of the `MPI_FILE_SET_VIEW` interface.

The user is responsible for ensuring that all nonblocking requests and split collective operations on `fh` have been completed before calling `MPI_FILE_ISET_VIEW`, and waiting for the completion of the operation before issuing any I/O requests to the file—otherwise, the call to `MPI_FILE_ISET_VIEW` is erroneous.

```

1 MPI_FILE_GET_VIEW(fh, disp, etype, filetype, datarep)
2     IN      fh                file handle (handle)
3     OUT     disp              displacement (integer)
4     OUT     etype             elementary datatype (handle)
5     OUT     filetype          filetype (handle)
6     OUT     datarep           data representation (string)
7
8
9
10 int MPI_File_get_view(MPI_File fh, MPI_Offset *disp, MPI_Datatype *etype,
11                      MPI_Datatype *filetype, char *datarep)
12
13 MPI_FILE_GET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, IERROR)
14     INTEGER FH, ETYPE, FILETYPE, IERROR
15     CHARACTER*(*) DATAREP
16     INTEGER(KIND=MPI_OFFSET_KIND) DISP
17
18 {void MPI::File::Get_view(MPI::Offset& disp, MPI::Datatype& etype,
19                          MPI::Datatype& filetype, char* datarep) const(binding deprecated,
20                          see Section 15.2) }

```

MPI_FILE_GET_VIEW returns the process's view of the data in the file. The current value of the displacement is returned in `disp`. The `etype` and `filetype` are new datatypes with typemaps equal to the typemaps of the current `etype` and `filetype`, respectively.

The data representation is returned in `datarep`. The user is responsible for ensuring that `datarep` is large enough to hold the returned data representation string. The length of a data representation string is limited to the value of `MPI_MAX_DATAREP_STRING`.

In addition, if a portable datatype was used to set the current view, then the corresponding datatype returned by `MPI_FILE_GET_VIEW` is also a portable datatype. If `etype` or `filetype` are derived datatypes, the user is responsible for freeing them. The `etype` and `filetype` returned are both in a committed state.

13.4 Data Access

13.4.1 Data Access Routines

Data is moved between files and processes by issuing read and write calls. There are three orthogonal aspects to data access: positioning (explicit offset *vs.* implicit file pointer), synchronism (blocking *vs.* nonblocking and split collective), and coordination (noncollective *vs.* collective). The following combinations of these data access routines, including two types of file pointers (individual and shared) are provided in Table 13.1.

POSIX `read()`/`fread()` and `write()`/`fwrite()` are blocking, noncollective operations and use individual file pointers. The MPI equivalents are `MPI_FILE_READ` and `MPI_FILE_WRITE`.

Implementations of data access routines may buffer data to improve performance. This does not affect reads, as the data is always available in the user's buffer after a read operation completes. For writes, however, the `MPI_FILE_SYNC` routine provides the only guarantee that data has been transferred to the storage device.

positioning	synchronism	coordination		1
		<i>noncollective</i>	<i>collective</i>	2
<i>explicit offsets</i>	<i>blocking</i>	MPI_FILE_READ_AT	MPI_FILE_READ_AT_ALL	3
		MPI_FILE_WRITE_AT	MPI_FILE_WRITE_AT_ALL	4
	<i>nonblocking & split collective</i>	MPI_FILE_IREAD_AT	MPI_FILE_READ_AT_ALL_BEGIN	5
		MPI_FILE_IWRITE_AT	MPI_FILE_WRITE_AT_ALL_BEGIN	6
<i>individual file pointers</i>	<i>blocking</i>	MPI_FILE_READ	MPI_FILE_READ_ALL	8
		MPI_FILE_WRITE	MPI_FILE_WRITE_ALL	9
	<i>nonblocking & split collective</i>	MPI_FILE_IREAD	MPI_FILE_READ_ALL_BEGIN	10
		MPI_FILE_IWRITE	MPI_FILE_WRITE_ALL_BEGIN	11
<i>shared file pointer</i>	<i>blocking</i>	MPI_FILE_READ_SHARED	MPI_FILE_READ_ORDERED	13
		MPI_FILE_WRITE_SHARED	MPI_FILE_WRITE_ORDERED	14
	<i>nonblocking & split collective</i>	MPI_FILE_IREAD_SHARED	MPI_FILE_READ_ORDERED_BEGIN	15
		MPI_FILE_IWRITE_SHARED	MPI_FILE_WRITE_ORDERED_BEGIN	16
			MPI_FILE_WRITE_ORDERED_END	17
				18

Table 13.1: Data access routines

Positioning

MPI provides three types of positioning for data access routines: explicit offsets, individual file pointers, and shared file pointers. The different positioning methods may be mixed within the same program and do not affect each other.

The data access routines that accept explicit offsets contain `_AT` in their name (e.g., `MPI_FILE_WRITE_AT`). Explicit offset operations perform data access at the file position given directly as an argument—no file pointer is used nor updated. Note that this is not equivalent to an atomic seek-and-read or seek-and-write operation, as no “seek” is issued. Operations with explicit offsets are described in Section 13.4.2, page 21.

The names of the individual file pointer routines contain no positional qualifier (e.g., `MPI_FILE_WRITE`). Operations with individual file pointers are described in Section 13.4.3, page 25. The data access routines that use shared file pointers contain `_SHARED` or `_ORDERED` in their name (e.g., `MPI_FILE_WRITE_SHARED`). Operations with shared file pointers are described in Section 13.4.4, page 31.

The main semantic issues with MPI-maintained file pointers are how and when they are updated by I/O operations. In general, each I/O operation leaves the file pointer pointing to the next data item after the last one that is accessed by the operation. In a nonblocking or split collective operation, the pointer is updated by the call that initiates the I/O, possibly before the access completes.

More formally,

$$new_file_offset = old_file_offset + \frac{elements(datatype)}{elements(etype)} \times count$$

where *count* is the number of *datatype* items to be accessed, *elements*(*X*) is the number of predefined datatypes in the typemap of *X*, and *old_file_offset* is the value of the implicit

offset before the call. The file position, *new_file_offset*, is in terms of a count of etypes relative to the current view.

Synchronism

MPI supports blocking and nonblocking I/O routines.

A *blocking* I/O call will not return until the I/O request is completed.

A *nonblocking* I/O call initiates an I/O operation, but does not wait for it to complete. Given suitable hardware, this allows the transfer of data out/in the user's buffer to proceed concurrently with computation. A separate *request complete* call (`MPI_WAIT`, `MPI_TEST`, or any of their variants) is needed to complete the I/O request, i.e., to confirm that the data has been read or written and that it is safe for the user to reuse the buffer. The nonblocking versions of the routines are named `MPI_FILE_IXXX`, where the *I* stands for immediate.

It is erroneous to access the local buffer of a nonblocking data access operation, or to use that buffer as the source or target of other communications, between the initiation and completion of the operation.

The split collective routines support a restricted form of “nonblocking” operations for collective data access (see Section 13.4.5, page 36).

Coordination

Every noncollective data access routine `MPI_FILE_XXX` has a collective counterpart. For most routines, this counterpart is `MPI_FILE_XXX_ALL` or a pair of `MPI_FILE_XXX_BEGIN` and `MPI_FILE_XXX_END`. The counterparts to the `MPI_FILE_XXX_SHARED` routines are `MPI_FILE_XXX_ORDERED`.

The completion of a noncollective call only depends on the activity of the calling process. However, the completion of a collective call (which must be called by all members of the process group) may depend on the activity of the other processes participating in the collective call. See Section 13.6.4, page 55, for rules on semantics of collective calls.

Collective operations may perform much better than their noncollective counterparts, as global data accesses have significant potential for automatic optimization.

Data Access Conventions

Data is moved between files and processes by calling read and write routines. Read routines move data from a file into memory. Write routines move data from memory into a file. The file is designated by a file handle, *fh*. The location of the file data is specified by an offset into the current view. The data in memory is specified by a triple: *buf*, *count*, and *datatype*. Upon completion, the amount of data accessed by the calling process is returned in a *status*.

An offset designates the starting position in the file for an access. The offset is always in etype units relative to the current view. Explicit offset routines pass *offset* as an argument (negative values are erroneous). The file pointer routines use implicit offsets maintained by MPI.

A data access routine attempts to transfer (read or write) *count* data items of type *datatype* between the user's buffer *buf* and the file. The *datatype* passed to the routine must be a committed datatype. The layout of data in memory corresponding to *buf*, *count*, *datatype* is interpreted the same way as in MPI communication functions; see Section 3.2.2 on page 27 and Section 4.1.11 on page 26. The data is accessed from those parts of the file specified by the current view (Section 13.3, page 15). The type signature of *datatype*

must match the type signature of some number of contiguous copies of the **etype** of the current view. As in a receive, it is erroneous to specify a **datatype** for reading that contains overlapping regions (areas of memory which would be stored into more than once).

The nonblocking data access routines indicate that MPI can start a data access and associate a request handle, **request**, with the I/O operation. Nonblocking operations are completed via **MPI_TEST**, **MPI_WAIT**, or any of their variants.

Data access operations, when completed, return the amount of data accessed in **status**.

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in subsections “Problems Due to Data Copying and Sequence Association,” and “A Problem with Register Optimization” in Section 16.2.2, pages 504 and 507. (*End of advice to users.*)

For blocking routines, **status** is returned directly. For nonblocking routines and split collective routines, **status** is returned when the operation is completed. The number of **datatype** entries and predefined elements accessed by the calling process can be extracted from **status** by using **MPI_GET_COUNT** and **MPI_GET_ELEMENTS**, respectively. The interpretation of the **MPI_ERROR** field is the same as for other operations — normally undefined, but meaningful if an MPI routine returns **MPI_ERR_IN_STATUS**. The user can pass (in C and Fortran) **MPI_STATUS_IGNORE** in the **status** argument if the return value of this argument is not needed. In C++, the **status** argument is optional. The **status** can be passed to **MPI_TEST_CANCELLED** to determine if the operation was cancelled. All other fields of **status** are undefined.

When reading, a program can detect the end of file by noting that the amount of data read is less than the amount requested. Writing past the end of file increases the file size. The amount of data accessed will be the amount requested, unless an error is raised (or a read reaches the end of file).

13.4.2 Data Access with Explicit Offsets

If **MPI_MODE_SEQUENTIAL** mode was specified when the file was opened, it is erroneous to call the routines in this section.

MPI_FILE_READ_AT(fh, offset, buf, count, datatype, status)

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)
OUT	status	status object (Status)

```
int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
                    MPI_Datatype datatype, MPI_Status *status)
```

```
MPI_FILE_READ_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
<type> BUF(*)
```



```

1      INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
2      INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
3
4      {void MPI::File::Read_at(MPI::Offset offset, void* buf, int count,
5          const MPI::Datatype& datatype, MPI::Status& status) (binding
6              deprecated, see Section 15.2) }
7
8      {void MPI::File::Read_at(MPI::Offset offset, void* buf, int count,
9          const MPI::Datatype& datatype) (binding deprecated, see Section 15.2)
10          }

```

MPI_FILE_READ_AT reads a file beginning at the position specified by offset.

MPI_FILE_READ_AT_ALL(fh, offset, buf, count, datatype, status)

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)
OUT	status	status object (Status)

```

23      int MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf,
24          int count, MPI_Datatype datatype, MPI_Status *status)
25
26      MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
27          <type> BUF(*)
28      INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
29      INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
30
31      {void MPI::File::Read_at_all(MPI::Offset offset, void* buf, int count,
32          const MPI::Datatype& datatype, MPI::Status& status) (binding
33              deprecated, see Section 15.2) }
34
35      {void MPI::File::Read_at_all(MPI::Offset offset, void* buf, int count,
36          const MPI::Datatype& datatype) (binding deprecated, see Section 15.2)
37          }

```

MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT interface.


```

MPI_FILE_WRITE_AT(fh, offset, buf, count, datatype, status)
    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)
    OUT      status            status object (Status)

int MPI_File_write_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
    MPI_Datatype datatype, MPI_Status *status)

MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

{void MPI::File::Write_at(MPI::Offset offset, const void* buf, int count,
    const MPI::Datatype& datatype, MPI::Status& status) (binding
    deprecated, see Section 15.2) }

{void MPI::File::Write_at(MPI::Offset offset, const void* buf, int count,
    const MPI::Datatype& datatype) (binding deprecated, see Section 15.2)
    }

    MPI_FILE_WRITE_AT writes a file beginning at the position specified by offset.

MPI_FILE_WRITE_AT_ALL(fh, offset, buf, count, datatype, status)
    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)
    OUT      status            status object (Status)

int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, void *buf,
    int count, MPI_Datatype datatype, MPI_Status *status)

MPI_FILE_WRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

{void MPI::File::Write_at_all(MPI::Offset offset, const void* buf,
    int count, const MPI::Datatype& datatype,
    MPI::Status& status) (binding deprecated, see Section 15.2) }

```

```

1  {void MPI::File::Write_at_all(MPI::Offset offset, const void* buf,
2      int count, const MPI::Datatype& datatype) (binding deprecated, see
3      Section 15.2) }

```

MPI_FILE_WRITE_AT_ALL is a collective version of the blocking MPI_FILE_WRITE_AT interface.

```

8  MPI_FILE_IREAD_AT(fh, offset, buf, count, datatype, request)

```

9	IN	fh	file handle (handle)
10			
11	IN	offset	file offset (integer)
12	OUT	buf	initial address of buffer (choice)
13			
14	IN	count	number of elements in buffer (integer)
15	IN	datatype	datatype of each buffer element (handle)
16	OUT	request	request object (handle)

```

18  int MPI_File_iread_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
19      MPI_Datatype datatype, MPI_Request *request)

```

```

21  MPI_FILE_IREAD_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)

```

```

22      <type> BUF(*)

```

```

23      INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR

```

```

24      INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

```

```

25  {MPI::Request MPI::File::Iread_at(MPI::Offset offset, void* buf, int count,
26      const MPI::Datatype& datatype) (binding deprecated, see Section 15.2)
27      }

```

MPI_FILE_IREAD_AT is a nonblocking version of the MPI_FILE_READ_AT interface.

```

32  MPI_FILE_IWRITE_AT(fh, offset, buf, count, datatype, request)

```

33	INOUT	fh	file handle (handle)
34	IN	offset	file offset (integer)
35			
36	IN	buf	initial address of buffer (choice)
37	IN	count	number of elements in buffer (integer)
38	IN	datatype	datatype of each buffer element (handle)
39	OUT	request	request object (handle)

```

42  int MPI_File_fwrite_at(MPI_File fh, MPI_Offset offset, void *buf,
43      int count, MPI_Datatype datatype, MPI_Request *request)

```

```

44  MPI_FILE_IWRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)

```

```

45      <type> BUF(*)

```

```

46      INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR

```

```

47      INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

```

```
{MPI::Request MPI::File::Iwrite_at(MPI::Offset offset, const void* buf,
    int count, const MPI::Datatype& datatype) (binding deprecated, see
    Section 15.2) }
```

MPI_FILE_IWRITE_AT is a nonblocking version of the MPI_FILE_WRITE_AT interface.

13.4.3 Data Access with Individual File Pointers

MPI maintains one individual file pointer per process per file handle. The current value of this pointer implicitly specifies the offset in the data access routines described in this section. These routines only use and update the individual file pointers maintained by MPI. The shared file pointer is not used nor updated.

The individual file pointer routines have the same semantics as the data access with explicit offset routines described in Section 13.4.2, page 21, with the following modification:

- the offset is defined to be the current value of the MPI-maintained individual file pointer.

After an individual file pointer operation is initiated, the individual file pointer is updated to point to the next etype after the last one that will be accessed. The file pointer is updated relative to the current view of the file.

If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is erroneous to call the routines in this section, with the exception of MPI_FILE_GET_BYTE_OFFSET.

MPI_FILE_READ(fh, buf, count, datatype, status)

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)
OUT	status	status object (Status)

```
int MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
    MPI_Status *status)
```

```
MPI_FILE_READ(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
```

```
<type> BUF(*)
```

```
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

```
{void MPI::File::Read(void* buf, int count, const MPI::Datatype& datatype,
    MPI::Status& status) (binding deprecated, see Section 15.2) }
```

```
{void MPI::File::Read(void* buf, int count,
    const MPI::Datatype& datatype) (binding deprecated, see Section 15.2)
}
```

MPI_FILE_READ reads a file using the individual file pointer.

Example 13.2 The following Fortran code fragment is an example of reading a file until the end of file is reached:

```

1  !   Read a preexisting input file until all data has been read.
2  !   Call routine "process_input" if all requested data is read.
3  !   The Fortran 90 "exit" statement exits the loop.
4
5      integer    bufsize, numread, totprocessed, status(MPI_STATUS_SIZE)
6      parameter (bufsize=100)
7      real       localbuffer(bufsize)
8
9      call MPI_FILE_OPEN( MPI_COMM_WORLD, 'myoldfile', &
10                          MPI_MODE_RDONLY, MPI_INFO_NULL, myfh, ierr )
11      call MPI_FILE_SET_VIEW( myfh, 0, MPI_REAL, MPI_REAL, 'native', &
12                              MPI_INFO_NULL, ierr )
13      totprocessed = 0
14      do
15          call MPI_FILE_READ( myfh, localbuffer, bufsize, MPI_REAL, &
16                              status, ierr )
17          call MPI_GET_COUNT( status, MPI_REAL, numread, ierr )
18          call process_input( localbuffer, numread )
19          totprocessed = totprocessed + numread
20          if ( numread < bufsize ) exit
21      enddo
22
23      write(6,1001) numread, bufsize, totprocessed
24 1001 format( "No more data:  read", I3, "and expected", I3, &
25             "Processed total of", I6, "before terminating job." )
26
27      call MPI_FILE_CLOSE( myfh, ierr )
28
29
30  MPI_FILE_READ_ALL(fh, buf, count, datatype, status)
31
32      INOUT    fh                file handle (handle)
33      OUT      buf              initial address of buffer (choice)
34      IN       count            number of elements in buffer (integer)
35      IN       datatype         datatype of each buffer element (handle)
36      OUT      status           status object (Status)
37
38
39  int MPI_File_read_all(MPI_File fh, void *buf, int count,
40                        MPI_Datatype datatype, MPI_Status *status)
41
42  MPI_FILE_READ_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
43      <type> BUF(*)
44      INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
45
46  {void MPI::File::Read_all(void* buf, int count,
47                            const MPI::Datatype& datatype, MPI::Status& status) (binding
48                            deprecated, see Section 15.2) }

```

```
{void MPI::File::Read_all(void* buf, int count,
    const MPI::Datatype& datatype) (binding deprecated, see Section 15.2)
}
```

MPI_FILE_READ_ALL is a collective version of the blocking MPI_FILE_READ interface.

MPI_FILE_WRITE(fh, buf, count, datatype, status)

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)
OUT	status	status object (Status)

```
int MPI_File_write(MPI_File fh, void *buf, int count,
    MPI_Datatype datatype, MPI_Status *status)
```

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

```
{void MPI::File::Write(const void* buf, int count,
    const MPI::Datatype& datatype, MPI::Status& status) (binding
    deprecated, see Section 15.2) }
```

```
{void MPI::File::Write(const void* buf, int count,
    const MPI::Datatype& datatype) (binding deprecated, see Section 15.2)
}
```

MPI_FILE_WRITE writes a file using the individual file pointer.

MPI_FILE_WRITE_ALL(fh, buf, count, datatype, status)

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)
OUT	status	status object (Status)

```
int MPI_File_write_all(MPI_File fh, void *buf, int count,
    MPI_Datatype datatype, MPI_Status *status)
```

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

```
{void MPI::File::Write_all(const void* buf, int count,
    const MPI::Datatype& datatype, MPI::Status& status) (binding
```

```

1      deprecated, see Section 15.2) }
2
3      {void MPI::File::Write_all(const void* buf, int count,
4          const MPI::Datatype& datatype) (binding deprecated, see Section 15.2)
5          }
6
7      MPI_FILE_WRITE_ALL is a collective version of the blocking MPI_FILE_WRITE inter-
8      face.
9
10     MPI_FILE_IREAD(fh, buf, count, datatype, request)
11
12     INOUT    fh                file handle (handle)
13
14     OUT      buf              initial address of buffer (choice)
15
16     IN       count            number of elements in buffer (integer)
17
18     IN       datatype          datatype of each buffer element (handle)
19
20     OUT      request           request object (handle)
21
22
23     int MPI_File_iread(MPI_File fh, void *buf, int count,
24         MPI_Datatype datatype, MPI_Request *request)
25
26     MPI_FILE_IREAD(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
27
28     <type> BUF(*)
29
30     INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
31
32     {MPI::Request MPI::File::Iread(void* buf, int count,
33         const MPI::Datatype& datatype) (binding deprecated, see Section 15.2)
34         }
35
36     MPI_FILE_IREAD is a nonblocking version of the MPI_FILE_READ interface.

```

Example 13.3 The following Fortran code fragment illustrates file pointer update semantics:

```

32
33     ! Read the first twenty real words in a file into two local
34     ! buffers. Note that when the first MPI_FILE_IREAD returns,
35     ! the file pointer has been updated to point to the
36     ! eleventh real word in the file.
37
38     integer    bufsize, req1, req2
39     integer, dimension(MPI_STATUS_SIZE) :: status1, status2
40     parameter (bufsize=10)
41     real       buf1(bufsize), buf2(bufsize)
42
43     call MPI_FILE_OPEN( MPI_COMM_WORLD, 'myoldfile', &
44         MPI_MODE_RDONLY, MPI_INFO_NULL, myfh, ierr )
45     call MPI_FILE_SET_VIEW( myfh, 0, MPI_REAL, MPI_REAL, 'native', &
46         MPI_INFO_NULL, ierr )
47     call MPI_FILE_IREAD( myfh, buf1, bufsize, MPI_REAL, &
48         req1, ierr )

```

```

call MPI_FILE_IREAD( myfh, buf2, bufsize, MPI_REAL, &      1
                     req2, ierr )                             2
                                                                3
call MPI_WAIT( req1, status1, ierr )                          4
call MPI_WAIT( req2, status2, ierr )                          5
                                                                6
call MPI_FILE_CLOSE( myfh, ierr )                             7
                                                                8
                                                                9
MPI_FILE_IWRITE(fh, buf, count, datatype, request)           10
                                                                11
INOUT    fh          file handle (handle)                    12
IN       buf          initial address of buffer (choice)     13
IN       count        number of elements in buffer (integer) 14
IN       datatype     datatype of each buffer element (handle) 15
OUT      request      request object (handle)                16
                                                                17
                                                                18
int MPI_File_irewrite(MPI_File fh, void *buf, int count,      19
                     MPI_Datatype datatype, MPI_Request *request) 20
                                                                21
MPI_FILE_IWRITE(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)    22
    <type> BUF(*)                                             23
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR              24
                                                                25
{MPI::Request MPI::File::Iwrite(const void* buf, int count,    26
    const MPI::Datatype& datatype) (binding deprecated, see Section 15.2) 27
    }                                                         28

```

MPI_FILE_IWRITE is a nonblocking version of the MPI_FILE_WRITE interface.

```

MPI_FILE_SEEK(fh, offset, whence)                             31
                                                                32
INOUT    fh          file handle (handle)                    33
IN       offset      file offset (integer)                   34
IN       whence      update mode (state)                     35
                                                                36
                                                                37
int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence) 38
                                                                39
MPI_FILE_SEEK(FH, OFFSET, WHENCE, IERROR)                     40
    INTEGER FH, WHENCE, IERROR                                 41
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET                      42
                                                                43
{void MPI::File::Seek(MPI::Offset offset, int whence) (binding deprecated, see 44
    Section 15.2) }                                           45

```

MPI_FILE_SEEK updates the individual file pointer according to *whence*, which has the following possible values:

- MPI_SEEK_SET: the pointer is set to offset

- MPI_SEEK_CUR: the pointer is set to the current pointer position plus offset
- MPI_SEEK_END: the pointer is set to the end of file plus offset

The offset can be negative, which allows seeking backwards. It is erroneous to seek to a negative position in the view.

MPI_FILE_GET_POSITION(fh, offset)

IN	fh	file handle (handle)
OUT	offset	offset of individual pointer (integer)

```
int MPI_File_get_position(MPI_File fh, MPI_Offset *offset)
```

```
MPI_FILE_GET_POSITION(FH, OFFSET, IERROR)
```

```
INTEGER FH, IERROR
```

```
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
```

```
{MPI::Offset MPI::File::Get_position() const(binding deprecated, see Section 15.2)
}
```

MPI_FILE_GET_POSITION returns, in **offset**, the current position of the individual file pointer in etype units relative to the current view.

Advice to users. The offset can be used in a future call to **MPI_FILE_SEEK** using **whence = MPI_SEEK_SET** to return to the current position. To set the displacement to the current file pointer position, first convert **offset** into an absolute byte position using **MPI_FILE_GET_BYTE_OFFSET**, then call **MPI_FILE_SET_VIEW** with the resulting displacement. (*End of advice to users.*)

MPI_FILE_GET_BYTE_OFFSET(fh, offset, disp)

IN	fh	file handle (handle)
IN	offset	offset (integer)
OUT	disp	absolute byte position of offset (integer)

```
int MPI_File_get_byte_offset(MPI_File fh, MPI_Offset offset,
                             MPI_Offset *disp)
```

```
MPI_FILE_GET_BYTE_OFFSET(FH, OFFSET, DISP, IERROR)
```

```
INTEGER FH, IERROR
```

```
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, DISP
```

```
{MPI::Offset MPI::File::Get_byte_offset(const MPI::Offset disp)
    const(binding deprecated, see Section 15.2) }
```

MPI_FILE_GET_BYTE_OFFSET converts a view-relative offset into an absolute byte position. The absolute byte position (from the beginning of the file) of **offset** relative to the current view of **fh** is returned in **disp**.

13.4.4 Data Access with Shared File Pointers

MPI maintains exactly one shared file pointer per collective `MPI_FILE_OPEN` (shared among processes in the communicator group). The current value of this pointer implicitly specifies the offset in the data access routines described in this section. These routines only use and update the shared file pointer maintained by MPI. The individual file pointers are not used nor updated.

The shared file pointer routines have the same semantics as the data access with explicit offset routines described in Section 13.4.2, page 21, with the following modifications:

- the offset is defined to be the current value of the MPI-maintained shared file pointer,
- the effect of multiple calls to shared file pointer routines is defined to behave as if the calls were serialized, and
- the use of shared file pointer routines is erroneous unless all processes use the same file view.

For the noncollective shared file pointer routines, the serialization ordering is not deterministic. The user needs to use other synchronization means to enforce a specific order.

After a shared file pointer operation is initiated, the shared file pointer is updated to point to the next etype after the last one that will be accessed. The file pointer is updated relative to the current view of the file.

Noncollective Operations

`MPI_FILE_READ_SHARED(fh, buf, count, datatype, status)`

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)
OUT	status	status object (Status)

```
int MPI_File_read_shared(MPI_File fh, void *buf, int count,
                        MPI_Datatype datatype, MPI_Status *status)
```

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

```
{void MPI::File::Read_shared(void* buf, int count,
    const MPI::Datatype& datatype, MPI::Status& status) (binding
    deprecated, see Section 15.2) }
```

```
{void MPI::File::Read_shared(void* buf, int count,
    const MPI::Datatype& datatype) (binding deprecated, see Section 15.2)
    }
```

`MPI_FILE_READ_SHARED` reads a file using the shared file pointer.

```

1  MPI_FILE_WRITE_SHARED(fh, buf, count, datatype, status)
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      OUT      status              status object (Status)
7
8
9
10 int MPI_File_write_shared(MPI_File fh, void *buf, int count,
11                          MPI_Datatype datatype, MPI_Status *status)
12
13 MPI_FILE_WRITE_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
14     <type> BUF(*)
15     INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
16
17 {void MPI::File::Write_shared(const void* buf, int count,
18                             const MPI::Datatype& datatype, MPI::Status& status) (binding
19                             deprecated, see Section 15.2) }
20
21 {void MPI::File::Write_shared(const void* buf, int count,
22                             const MPI::Datatype& datatype) (binding deprecated, see Section 15.2)
23                             }
24
25 MPI_FILE_WRITE_SHARED writes a file using the shared file pointer.
26
27
28 MPI_FILE_IREAD_SHARED(fh, buf, count, datatype, request)
29      INOUT    fh                file handle (handle)
30      OUT      buf                initial address of buffer (choice)
31      IN       count              number of elements in buffer (integer)
32      IN       datatype            datatype of each buffer element (handle)
33      OUT      request             request object (handle)
34
35
36
37 int MPI_File_iread_shared(MPI_File fh, void *buf, int count,
38                          MPI_Datatype datatype, MPI_Request *request)
39
40 MPI_FILE_IREAD_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
41     <type> BUF(*)
42     INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
43
44 {MPI::Request MPI::File::Iread_shared(void* buf, int count,
45                                     const MPI::Datatype& datatype) (binding deprecated, see Section 15.2)
46                                     }
47
48 MPI_FILE_IREAD_SHARED is a nonblocking version of the MPI_FILE_READ_SHARED
49 interface.

```

```

MPI_FILE_IWRITE_SHARED(fh, buf, count, datatype, request)
    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)
    OUT      request             request object (handle)

int MPI_File_iwrite_shared(MPI_File fh, void *buf, int count,
                           MPI_Datatype datatype, MPI_Request *request)

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

{MPI::Request MPI::File::Iwrite_shared(const void* buf, int count,
    const MPI::Datatype& datatype) (binding deprecated, see Section 15.2)
    }

```

MPI_FILE_IWRITE_SHARED is a nonblocking version of the MPI_FILE_WRITE_SHARED interface.

Collective Operations

The semantics of a collective access using a shared file pointer is that the accesses to the file will be in the order determined by the ranks of the processes within the group. For each process, the location in the file at which data is accessed is the position at which the shared file pointer would be after all processes whose ranks within the group less than that of this process had accessed their data. In addition, in order to prevent subsequent shared offset accesses by the same processes from interfering with this collective access, the call might return only after all the processes within the group have initiated their accesses. When the call returns, the shared file pointer points to the next etype accessible, according to the file view used by all processes, after the last etype requested.

Advice to users. There may be some programs in which all processes in the group need to access the file using the shared file pointer, but the program may not *require* that data be accessed in order of process rank. In such programs, using the shared ordered routines (e.g., MPI_FILE_WRITE_ORDERED rather than MPI_FILE_WRITE_SHARED) may enable an implementation to optimize access, improving performance. (*End of advice to users.*)

Advice to implementors. Accesses to the data requested by all processes do not have to be serialized. Once all processes have issued their requests, locations within the file for all accesses can be computed, and accesses can proceed independently from each other, possibly in parallel. (*End of advice to implementors.*)

```

1  MPI_FILE_READ_ORDERED(fh, buf, count, datatype, status)
2      INOUT    fh                file handle (handle)
3      OUT      buf                initial address of buffer (choice)
4
5      IN        count            number of elements in buffer (integer)
6      IN        datatype         datatype of each buffer element (handle)
7      OUT      status            status object (Status)
8
9
10 int MPI_File_read_ordered(MPI_File fh, void *buf, int count,
11                          MPI_Datatype datatype, MPI_Status *status)
12
13 MPI_FILE_READ_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
14     <type> BUF(*)
15     INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
16
17 {void MPI::File::Read_ordered(void* buf, int count,
18                               const MPI::Datatype& datatype, MPI::Status& status) (binding
19                               deprecated, see Section 15.2) }
20
21 {void MPI::File::Read_ordered(void* buf, int count,
22                               const MPI::Datatype& datatype) (binding deprecated, see Section 15.2)
23                               }
24
25 MPI_FILE_READ_ORDERED is a collective version of the MPI_FILE_READ_SHARED
26 interface.
27
28 MPI_FILE_WRITE_ORDERED(fh, buf, count, datatype, status)
29      INOUT    fh                file handle (handle)
30      IN        buf                initial address of buffer (choice)
31      IN        count            number of elements in buffer (integer)
32      IN        datatype         datatype of each buffer element (handle)
33      OUT      status            status object (Status)
34
35
36 int MPI_File_write_ordered(MPI_File fh, void *buf, int count,
37                          MPI_Datatype datatype, MPI_Status *status)
38
39 MPI_FILE_WRITE_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
40     <type> BUF(*)
41     INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
42
43 {void MPI::File::Write_ordered(const void* buf, int count,
44                               const MPI::Datatype& datatype, MPI::Status& status) (binding
45                               deprecated, see Section 15.2) }
46
47 {void MPI::File::Write_ordered(const void* buf, int count,
48                               const MPI::Datatype& datatype) (binding deprecated, see Section 15.2)
49                               }

```

MPI_FILE_WRITE_ORDERED is a collective version of the MPI_FILE_WRITE_SHARED interface.

Seek

If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is erroneous to call the following two routines (MPI_FILE_SEEK_SHARED and MPI_FILE_GET_POSITION_SHARED).

MPI_FILE_SEEK_SHARED(fh, offset, whence)

INOUT	fh	file handle (handle)
IN	offset	file offset (integer)
IN	whence	update mode (state)

int MPI_File_seek_shared(MPI_File fh, MPI_Offset offset, int whence)

MPI_FILE_SEEK_SHARED(FH, OFFSET, WHENCE, IERROR)

INTEGER FH, WHENCE, IERROR

INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

{void MPI::File::Seek_shared(MPI::Offset offset, int whence) (*binding deprecated, see Section 15.2*) }

MPI_FILE_SEEK_SHARED updates the shared file pointer according to whence, which has the following possible values:

- MPI_SEEK_SET: the pointer is set to offset
- MPI_SEEK_CUR: the pointer is set to the current pointer position plus offset
- MPI_SEEK_END: the pointer is set to the end of file plus offset

MPI_FILE_SEEK_SHARED is collective; all the processes in the communicator group associated with the file handle fh must call MPI_FILE_SEEK_SHARED with the same values for offset and whence.

The offset can be negative, which allows seeking backwards. It is erroneous to seek to a negative position in the view.

MPI_FILE_GET_POSITION_SHARED(fh, offset)

IN	fh	file handle (handle)
OUT	offset	offset of shared pointer (integer)

int MPI_File_get_position_shared(MPI_File fh, MPI_Offset *offset)

MPI_FILE_GET_POSITION_SHARED(FH, OFFSET, IERROR)

INTEGER FH, IERROR

INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

```

1 {MPI::Offset MPI::File::Get_position_shared() const(binding deprecated, see
2   Section 15.2) }
```

3 MPI_FILE_GET_POSITION_SHARED returns, in offset, the current position of the
4 shared file pointer in etype units relative to the current view.
5

6 *Advice to users.* The offset can be used in a future call to MPI_FILE_SEEK_SHARED
7 using whence = MPI_SEEK_SET to return to the current position. To set the displace-
8 ment to the current file pointer position, first convert offset into an absolute byte
9 position using MPI_FILE_GET_BYTE_OFFSET, then call MPI_FILE_SET_VIEW with
10 the resulting displacement. (*End of advice to users.*)
11

12 13.4.5 Split Collective Data Access Routines

13 MPI provides a restricted form of “nonblocking collective” I/O operations for all data ac-
14 cesses using split collective data access routines. These routines are referred to as “split”
15 collective routines because a single collective operation is split in two: a begin routine and
16 an end routine. The begin routine begins the operation, much like a nonblocking data access
17 (e.g., MPI_FILE_IREAD). The end routine completes the operation, much like the matching
18 test or wait (e.g., MPI_WAIT). As with nonblocking data access operations, the user must
19 not use the buffer passed to a begin routine while the routine is outstanding; the operation
20 must be completed with an end routine before it is safe to free buffers, etc.
21

22 Split collective data access operations on a file handle fh are subject to the semantic
23 rules given below.
24

- 25 • On any MPI process, each file handle may have at most one active split collective
26 operation at any time.
- 27 • Begin calls are collective over the group of processes that participated in the collective
28 open and follow the ordering rules for collective calls.
- 29 • End calls are collective over the group of processes that participated in the collective
30 open and follow the ordering rules for collective calls. Each end call matches the
31 preceding begin call for the same collective operation. When an “end” call is made,
32 exactly one unmatched “begin” call for the same operation must precede it.
- 33 • An implementation is free to implement any split collective data access routine using
34 the corresponding blocking collective routine when either the begin call (e.g.,
35 MPI_FILE_READ_ALL_BEGIN) or the end call (e.g., MPI_FILE_READ_ALL_END) is
36 issued. The begin and end calls are provided to allow the user and MPI implementation
37 to optimize the collective operation.
38
- 39 • Split collective operations do not match the corresponding regular collective opera-
40 tion. For example, in a single collective read operation, an MPI_FILE_READ_ALL
41 on one process does not match an MPI_FILE_READ_ALL_BEGIN/
42 MPI_FILE_READ_ALL_END pair on another process.
- 43 • Split collective routines must specify a buffer in both the begin and end routines.
44 By specifying the buffer that receives data in the end routine, we can avoid many
45 (though not all) of the problems described in “A Problem with Register Optimization,”
46 Section 16.2.2, page 507.
47
48

- 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 13.6.1, page 52), 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 15.2) }

```

```

1  MPI_FILE_READ_AT_ALL_END(fh, buf, status)
2      IN      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_at_all_end(MPI_File fh, void *buf, MPI_Status *status)
8
9  MPI_FILE_READ_AT_ALL_END(FH, BUF, STATUS, IERROR)
10     <type> BUF(*)
11     INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
12
13     {void MPI::File::Read_at_all_end(void* buf, MPI::Status& status) (binding
14         deprecated, see Section 15.2) }
15
16     {void MPI::File::Read_at_all_end(void* buf) (binding deprecated, see Section 15.2)
17         }
18
19  MPI_FILE_WRITE_AT_ALL_BEGIN(fh, offset, buf, count, datatype)
20      INOUT   fh                file handle (handle)
21      IN      offset            file offset (integer)
22      IN      buf                initial address of buffer (choice)
23      IN      count              number of elements in buffer (integer)
24      IN      datatype           datatype of each buffer element (handle)
25
26
27  int MPI_File_write_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf,
28      int count, MPI_Datatype datatype)
29
30  MPI_FILE_WRITE_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)
31     <type> BUF(*)
32     INTEGER FH, COUNT, DATATYPE, IERROR
33     INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
34
35     {void MPI::File::Write_at_all_begin(MPI::Offset offset, const void* buf,
36         int count, const MPI::Datatype& datatype) (binding deprecated, see
37         Section 15.2) }
38
39
40  MPI_FILE_WRITE_AT_ALL_END(fh, buf, status)
41      INOUT   fh                file handle (handle)
42      IN      buf                initial address of buffer (choice)
43      OUT     status             status object (Status)
44
45
46  int MPI_File_write_at_all_end(MPI_File fh, void *buf, MPI_Status *status)
47
48  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 15.2) }
{void MPI::File::Write_at_all_end(const void* buf) (binding deprecated, see
    Section 15.2) }

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 15.2)
    }

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 15.2) }
{void MPI::File::Read_all_end(void* buf) (binding deprecated, see Section 15.2) }

```

```

1  MPI_FILE_WRITE_ALL_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_all_begin(MPI_File fh, void *buf, int count,
10                               MPI_Datatype datatype)
11
12  MPI_FILE_WRITE_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
13      <type> BUF(*)
14      INTEGER FH, COUNT, DATATYPE, IERROR
15
16  {void MPI::File::Write_all_begin(const void* buf, int count,
17                                  const MPI::Datatype& datatype) (binding deprecated, see Section 15.2)
18      }
19
20  MPI_FILE_WRITE_ALL_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_all_end(MPI_File fh, void *buf, MPI_Status *status)
27
28  MPI_FILE_WRITE_ALL_END(FH, BUF, STATUS, IERROR)
29      <type> BUF(*)
30      INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
31
32  {void MPI::File::Write_all_end(const void* buf, MPI::Status& status) (binding deprecated, see Section 15.2) }
33
34  {void MPI::File::Write_all_end(const void* buf) (binding deprecated, see Section 15.2) }
35
36
37
38  MPI_FILE_READ_ORDERED_BEGIN(fh, buf, count, datatype)
39      INOUT    fh                file handle (handle)
40      OUT      buf                initial address of buffer (choice)
41      IN       count              number of elements in buffer (integer)
42      IN       datatype            datatype of each buffer element (handle)
43
44
45  int MPI_File_read_ordered_begin(MPI_File fh, void *buf, int count,
46                                  MPI_Datatype datatype)
47
48  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 15.2)
    }
}

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 15.2) }
{void MPI::File::Read_ordered_end(void* buf) (binding deprecated, see Section 15.2)
    }

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 15.2)
    }
}

```

```

1 MPI_FILE_WRITE_ORDERED_END(fh, buf, status)
2     INOUT    fh                file handle (handle)
3     IN       buf              initial address of buffer (choice)
4     OUT      status           status object (Status)
5
6
7 int MPI_File_write_ordered_end(MPI_File fh, void *buf, MPI_Status *status)
8
9 MPI_FILE_WRITE_ORDERED_END(FH, BUF, STATUS, IERROR)
10     <type> BUF(*)
11     INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
12
13 {void MPI::File::Write_ordered_end(const void* buf,
14     MPI::Status& status) (binding deprecated, see Section 15.2) }
15
16 {void MPI::File::Write_ordered_end(const void* buf) (binding deprecated, see
17     Section 15.2) }
18
19
20

```

13.5 File Interoperability

At the most basic level, file interoperability is the ability to read the information previously written to a file—not just the bits of data, but the actual information the bits represent. MPI guarantees full interoperability within a single MPI environment, and supports increased interoperability outside that environment through the external data representation (Section 13.5.2, page 46) as well as the data conversion functions (Section 13.5.3, page 46).

Interoperability within a single MPI environment (which could be considered “operability”) ensures that file data written by one MPI process can be read by any other MPI process, subject to the consistency constraints (see Section 13.6.1, page 52), provided that it would have been possible to start the two processes simultaneously and have them reside in a single MPI_COMM_WORLD. Furthermore, both processes must see the same data values at every absolute byte offset in the file for which data was written.

This single environment file interoperability implies that file data is accessible regardless of the number of processes.

There are three aspects to file interoperability:

- transferring the bits,
- converting between different file structures, and
- converting between different machine representations.

The first two aspects of file interoperability are beyond the scope of this standard, as both are highly machine dependent. However, transferring the bits of a file into and out of the MPI environment (e.g., by writing a file to tape) is required to be supported by all MPI implementations. In particular, an implementation must specify how familiar operations similar to POSIX `cp`, `rm`, and `mv` can be performed on the file. Furthermore, it is expected that the facility provided maintains the correspondence between absolute byte offsets (e.g., after possible file structure conversion, the data bits at byte offset 102 in the MPI environment are at byte offset 102 outside the MPI environment). As an example, a simple off-line conversion utility that transfers and converts files between the native file

system and the MPI environment would suffice, provided it maintained the offset coherence mentioned above. In a high-quality implementation of MPI, users will be able to manipulate MPI files using the same or similar tools that the native file system offers for manipulating its files.

The remaining aspect of file interoperability, converting between different machine representations, is supported by the typing information specified in the etype and filetype. This facility allows the information in files to be shared between any two applications, regardless of whether they use MPI, and regardless of the machine architectures on which they run.

MPI supports multiple data representations: “native,” “internal,” and “external32.” An implementation may support additional data representations. MPI also supports user-defined data representations (see Section 13.5.3, page 46). The “native” and “internal” data representations are implementation dependent, while the “external32” representation is common to all MPI implementations and facilitates file interoperability. The data representation is specified in the *datarep* argument to `MPI_FILE_SET_VIEW`.

Advice to users. MPI is not guaranteed to retain knowledge of what data representation was used when a file is written. Therefore, to correctly retrieve file data, an MPI application is responsible for specifying the same data representation as was used to create the file. (*End of advice to users.*)

“native” Data in this representation is stored in a file exactly as it is in memory. The advantage of this data representation is that data precision and I/O performance are not lost in type conversions with a purely homogeneous environment. The disadvantage is the loss of transparent interoperability within a heterogeneous MPI environment.

Advice to users. This data representation should only be used in a homogeneous MPI environment, or when the MPI application is capable of performing the data type conversions itself. (*End of advice to users.*)

Advice to implementors. When implementing read and write operations on top of MPI message-passing, the message data should be typed as `MPI_BYTE` to ensure that the message routines do not perform any type conversions on the data. (*End of advice to implementors.*)

“internal” This data representation can be used for I/O operations in a homogeneous or heterogeneous environment; the implementation will perform type conversions if necessary. The implementation is free to store data in any format of its choice, with the restriction that it will maintain constant extents for all predefined datatypes in any one file. The environment in which the resulting file can be reused is implementation-defined and must be documented by the implementation.

Rationale. This data representation allows the implementation to perform I/O efficiently in a heterogeneous environment, though with implementation-defined restrictions on how the file can be reused. (*End of rationale.*)

Advice to implementors. Since “external32” is a superset of the functionality provided by “internal,” an implementation may choose to implement “internal” as “external32.” (*End of advice to implementors.*)

“external32” This data representation states that read and write operations convert all data from and to the “external32” representation defined in Section 13.5.2, page 46. The data conversion rules for communication also apply to these conversions (see Section 3.3.2, page 25-27, of the MPI-1 document). The data on the storage medium is always in this canonical representation, and the data in memory is always in the local process’s native representation.

This data representation has several advantages. First, all processes reading the file in a heterogeneous MPI environment will automatically have the data converted to their respective native representations. Second, the file can be exported from one MPI environment and imported into any other MPI environment with the guarantee that the second environment will be able to read all the data in the file.

The disadvantage of this data representation is that data precision and I/O performance may be lost in data type conversions.

Advice to implementors. When implementing read and write operations on top of MPI message-passing, the message data should be converted to and from the “external32” representation in the client, and sent as type MPI_BYTE. This will avoid possible double data type conversions and the associated further loss of precision and performance. (*End of advice to implementors.*)

13.5.1 Datatypes for File Interoperability

If the file data representation is other than “native,” care must be taken in constructing etypes and filetypes. Any of the datatype constructor functions may be used; however, for those functions that accept displacements in bytes, the displacements must be specified in terms of their values in the file for the file data representation being used. MPI will interpret these byte displacements as is; no scaling will be done. The function MPI_FILE_GET_TYPE_EXTENT can be used to calculate the extents of datatypes in the file. For etypes and filetypes that are portable datatypes (see Section 2.4, page 11), MPI will scale any displacements in the datatypes to match the file data representation. Datatypes passed as arguments to read/write routines specify the data layout in memory; therefore, they must always be constructed using displacements corresponding to displacements in memory.

Advice to users. One can logically think of the file as if it were stored in the memory of a file server. The etype and filetype are interpreted as if they were defined at this file server, by the same sequence of calls used to define them at the calling process. If the data representation is “native”, then this logical file server runs on the same architecture as the calling process, so that these types define the same data layout on the file as they would define in the memory of the calling process. If the etype and filetype are portable datatypes, then the data layout defined in the file is the same as would be defined in the calling process memory, up to a scaling factor. The routine MPI_FILE_GET_FILE_EXTENT can be used to calculate this scaling factor. Thus, two equivalent, portable datatypes will define the same data layout in the file, even in a heterogeneous environment with “internal”, “external32”, or user defined data representations. Otherwise, the etype and filetype must be constructed so that their typemap and extent are the same on any architecture. This can be achieved if they have an explicit upper bound and lower bound (defined either using MPI_LB and

MPI_UB markers, or using MPI_TYPE_CREATE_RESIZED). This condition must also be fulfilled by any datatype that is used in the construction of the `etype` and `filetype`, if this datatype is replicated contiguously, either explicitly, by a call to MPI_TYPE_CONTIGUOUS, or implicitly, by a blocklength argument that is greater than one. If an `etype` or `filetype` is not portable, and has a typemap or extent that is architecture dependent, then the data layout specified by it on a file is implementation dependent.

File data representations other than “native” may be different from corresponding data representations in memory. Therefore, for these file data representations, it is important not to use hardwired byte offsets for file positioning, including the initial displacement that specifies the view. When a portable datatype (see Section 2.4, page 11) is used in a data access operation, any holes in the datatype are scaled to match the data representation. However, note that this technique only works when all the processes that created the file view build their etypes from the same predefined datatypes. For example, if one process uses an `etype` built from MPI_INT and another uses an `etype` built from MPI_FLOAT, the resulting views may be nonportable because the relative sizes of these types may differ from one data representation to another. (*End of advice to users.*)

MPI_FILE_GET_TYPE_EXTENT(fh, datatype, extent)

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

```
int MPI_File_get_type_extent(MPI_File fh, MPI_Datatype datatype,
                             MPI_Aint *extent)
```

```
MPI_FILE_GET_TYPE_EXTENT(FH, DATATYPE, EXTENT, IERROR)
    INTEGER FH, DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT
```

```
{MPI::Aint MPI::File::Get_type_extent(const MPI::Datatype& datatype)
    const(binding deprecated, see Section 15.2) }
```

Returns the extent of `datatype` in the file `fh`. This extent will be the same for all processes accessing the file `fh`. If the current view uses a user-defined data representation (see Section 13.5.3, page 46), MPI uses the `dtype_file_extent_fn` callback to calculate the extent.

Advice to implementors. In the case of user-defined data representations, the extent of a derived datatype can be calculated by first determining the extents of the predefined datatypes in this derived datatype using `dtype_file_extent_fn` (see Section 13.5.3, page 46). (*End of advice to implementors.*)

13.5.2 External Data Representation: “external32”

All MPI implementations are required to support the data representation defined in this section. Support of optional datatypes (e.g., `MPI_INTEGER2`) is not required.

All floating point values are in big-endian IEEE format [3] of the appropriate size. Floating point values are represented by one of three IEEE formats. These are the IEEE “Single,” “Double,” and “Double Extended” formats, requiring 4, 8 and 16 bytes of storage, respectively. For the IEEE “Double Extended” formats, MPI specifies a Format Width of 16 bytes, with 15 exponent bits, bias = +16383, 112 fraction bits, and an encoding analogous to the “Double” format. All integral values are in two’s complement big-endian format. Big-endian means most significant byte at lowest address byte. For C `_Bool`, Fortran `LOGICAL` and C++ `bool`, 0 implies false and nonzero implies true. C `float` `_Complex`, `double` `_Complex` and long `double` `_Complex` as well as Fortran `COMPLEX` and `DOUBLE COMPLEX` are represented by a pair of floating point format values for the real and imaginary components. Characters are in ISO 8859-1 format [4]. Wide characters (of type `MPI_WCHAR`) are in Unicode format [11].

All signed numerals (e.g., `MPI_INT`, `MPI_REAL`) have the sign bit at the most significant bit. `MPI_COMPLEX` and `MPI_DOUBLE_COMPLEX` have the sign bit of the real and imaginary parts at the most significant bit of each part.

According to IEEE specifications [3], the “NaN” (not a number) is system dependent. It should not be interpreted within MPI as anything other than “NaN.”

Advice to implementors. The MPI treatment of “NaN” is similar to the approach used in XDR (see <ftp://ds.internic.net/rfc/rfc1832.txt>). (*End of advice to implementors.*)

All data is byte aligned, regardless of type. All data items are stored contiguously in the file (if the file view is contiguous).

Advice to implementors. All bytes of `LOGICAL` and `bool` must be checked to determine the value. (*End of advice to implementors.*)

Advice to users. The type `MPI_PACKED` is treated as bytes and is not converted. The user should be aware that `MPI_PACK` has the option of placing a header in the beginning of the pack buffer. (*End of advice to users.*)

The size of the predefined datatypes returned from `MPI_TYPE_CREATE_F90_REAL`, `MPI_TYPE_CREATE_F90_COMPLEX`, and `MPI_TYPE_CREATE_F90_INTEGER` are defined in Section 16.2.5, page 515.

Advice to implementors. When converting a larger size integer to a smaller size integer, only the less significant bytes are moved. Care must be taken to preserve the sign bit value. This allows no conversion errors if the data range is within the range of the smaller size integer. (*End of advice to implementors.*)

Table 13.2 specifies the sizes of predefined datatypes in “external32” format.

13.5.3 User-Defined Data Representations

There are two situations that cannot be handled by the required representations:

Type	Length	Optional Type	Length
-----	-----	-----	-----
MPI_PACKED	1	MPI_INTEGER1	1
MPI_BYTE	1	MPI_INTEGER2	2
MPI_CHAR	1	MPI_INTEGER4	4
MPI_UNSIGNED_CHAR	1	MPI_INTEGER8	8
MPI_SIGNED_CHAR	1	MPI_INTEGER16	16
MPI_WCHAR	2		
MPI_SHORT	2	MPI_REAL2	2
MPI_UNSIGNED_SHORT	2	MPI_REAL4	4
MPI_INT	4	MPI_REAL8	8
MPI_UNSIGNED	4	MPI_REAL16	16
MPI_LONG	4		
MPI_UNSIGNED_LONG	4	MPI_COMPLEX4	2*2
MPI_LONG_LONG_INT	8	MPI_COMPLEX8	2*4
MPI_UNSIGNED_LONG_LONG	8	MPI_COMPLEX16	2*8
MPI_FLOAT	4	MPI_COMPLEX32	2*16
MPI_DOUBLE	8		
MPI_LONG_DOUBLE	16		
MPI_C_BOOL	4		
MPI_INT8_T	1		
MPI_INT16_T	2		
MPI_INT32_T	4		
MPI_INT64_T	8		
MPI_UINT8_T	1		
MPI_UINT16_T	2		
MPI_UINT32_T	4		
MPI_UINT64_T	8		
MPI_AINT	8		
MPI_OFFSET	8		
MPI_C_COMPLEX	2*4		
MPI_C_FLOAT_COMPLEX	2*4		
MPI_C_DOUBLE_COMPLEX	2*8		
MPI_C_LONG_DOUBLE_COMPLEX	2*16		
MPI_CHARACTER	1		
MPI_LOGICAL	4		
MPI_INTEGER	4		
MPI_REAL	4		
MPI_DOUBLE_PRECISION	8		
MPI_COMPLEX	2*4		
MPI_DOUBLE_COMPLEX	2*8		

Table 13.2: “external32” sizes of predefined datatypes

1. a user wants to write a file in a representation unknown to the implementation, and
2. a user wants to read a file written in a representation unknown to the implementation.

User-defined data representations allow the user to insert a third party converter into the I/O stream to do the data representation conversion.

```

MPI_REGISTER_DATAREP(datarep, read_conversion_fn, write_conversion_fn,
                      dtype_file_extent_fn, extra_state)

IN      datarep      data representation identifier (string)
IN      read_conversion_fn  function invoked to convert from file representation to
                             native representation (function)
IN      write_conversion_fn  function invoked to convert from native representation
                             to file representation (function)
IN      dtype_file_extent_fn  function invoked to get the extent of a datatype as
                             represented in the file (function)
IN      extra_state      extra state

int MPI_Register_datarep(char *datarep,
                        MPI_Datarep_conversion_function *read_conversion_fn,
                        MPI_Datarep_conversion_function *write_conversion_fn,
                        MPI_Datarep_extent_function *dtype_file_extent_fn,
                        void *extra_state)

MPI_REGISTER_DATAREP(DATAREP, READ_CONVERSION_FN, WRITE_CONVERSION_FN,
                     DTYPE_FILE_EXTENT_FN, EXTRA_STATE, IERROR)
CHARACTER*(*) DATAREP
EXTERNAL READ_CONVERSION_FN, WRITE_CONVERSION_FN, DTYPE_FILE_EXTENT_FN
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
INTEGER IERROR

{void MPI::Register_datarep(const char* datarep,
                           MPI::Datarep_conversion_function* read_conversion_fn,
                           MPI::Datarep_conversion_function* write_conversion_fn,
                           MPI::Datarep_extent_function* dtype_file_extent_fn,
                           void* extra_state) (binding deprecated, see Section 15.2) }
```

The call associates `read_conversion_fn`, `write_conversion_fn`, and `dtype_file_extent_fn` with the data representation identifier `datarep`. `datarep` can then be used as an argument to `MPI_FILE_SET_VIEW`, causing subsequent data access operations to call the conversion functions to convert all data items accessed between file data representation and native representation. `MPI_REGISTER_DATAREP` is a local operation and only registers the data representation for the calling MPI process. If `datarep` is already defined, an error in the error class `MPI_ERR_DUP_DATAREP` is raised using the default file error handler (see Section 13.7, page 61). The length of a data representation string is limited to the value of `MPI_MAX_DATAREP_STRING`. `MPI_MAX_DATAREP_STRING` must have a value of at least 64. No routines are provided to delete data representations and free the associated resources; it is not expected that an application will generate them in significant numbers.

Extent Callback

```

typedef int MPI_Datarep_extent_function(MPI_Datatype datatype,
                                         MPI_Aint *file_extent, void *extra_state);

SUBROUTINE DATAREP_EXTENT_FUNCTION(DATATYPE, EXTENT, EXTRA_STATE, IERROR)
    INTEGER DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, EXTRA_STATE

{typedef void MPI::Datarep_extent_function(const MPI::Datatype& datatype,
                                         MPI::Aint& file_extent, void* extra_state); (binding deprecated,
                                         see Section 15.2)}
```

The function `dtype_file_extent_fn` must return, in `file_extent`, the number of bytes required to store `datatype` in the file representation. The function is passed, in `extra_state`, the argument that was passed to the `MPI_REGISTER_DATAREP` call. MPI will only call 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);

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

{typedef void MPI::Datarep_conversion_function(void* userbuf,
                                             MPI::Datatype& datatype, int count, void* filebuf,
                                             MPI::Offset position, void* extra_state); (binding deprecated, see
                                             Section 15.2)}
```

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 write routines in Section 13.4, page 18, or `MPI_FILE_GET_TYPE_EXTENT` is called by the user. `dtype_file_extent_fn` will only be passed predefined datatypes employed by the user. The conversion functions will only be passed datatypes equivalent to those that the user has passed to one of the routines noted above.

The conversion functions must be reentrant. User defined data representations are restricted to use byte alignment for all types. Furthermore, it is erroneous for the conversion functions to call any collective routines or to free `datatype`.

The conversion functions should return an error code. If the returned error code has a value other than `MPI_SUCCESS`, the implementation will raise an error in the class `MPI_ERR_CONVERSION`.

13.5.4 Matching Data Representations

It is the user's responsibility to ensure that the data representation used to read data from a file is *compatible* with the data representation that was used to write that data to the file.

In general, using the same data representation name when writing and reading a file does not guarantee that the representation is compatible. Similarly, using different representation names on two different implementations may yield compatible representations.

Compatibility can be obtained when "external32" representation is used, although precision may be lost and the performance may be less than when "native" representation is used. Compatibility is guaranteed using "external32" provided at least one of the following conditions is met.

- The data access routines directly use types enumerated in Section 13.5.2, page 46, that are supported by all implementations participating in the I/O. The predefined type used to write a data item must also be used to read a data item.
- In the case of Fortran 90 programs, the programs participating in the data accesses obtain compatible datatypes using MPI routines that specify precision and/or range (Section 16.2.5, page 511).
- For any given data item, the programs participating in the data accesses use compatible predefined types to write and read the data item.

User-defined data representations may be used to provide an implementation compatibility with another implementation's "native" or "internal" representation.

Advice to users. Section 16.2.5, page 511, defines routines that support the use of matching datatypes in heterogeneous environments and contains examples illustrating their use. (*End of advice to users.*)

13.6 Consistency and Semantics

13.6.1 File Consistency

Consistency semantics define the outcome of multiple accesses to a single file. All file accesses in MPI are relative to a specific file handle created from a collective open. MPI provides three levels of consistency: sequential consistency among all accesses using a single file handle, sequential consistency among all accesses using file handles created from a single collective open with atomic mode enabled, and user-imposed consistency among accesses other than the above. Sequential consistency means the behavior of a set of operations will be as if the operations were performed in some serial order consistent with program order; each access appears atomic, although the exact ordering of accesses is unspecified. User-imposed consistency may be obtained using program order and calls to `MPI_FILE_SYNC`.

Let FH_1 be the set of file handles created from one particular collective open of the file *FOO*, and FH_2 be the set of file handles created from a different collective open of *FOO*. Note that nothing restrictive is said about FH_1 and FH_2 : the sizes of FH_1 and FH_2 may be different, the groups of processes used for each open may or may not intersect, the file handles in FH_1 may be destroyed before those in FH_2 are created, etc. Consider the following three cases: a single file handle (e.g., $fh_1 \in FH_1$), two file handles created from a single collective open (e.g., $fh_{1a} \in FH_1$ and $fh_{1b} \in FH_1$), and two file handles from different collective opens (e.g., $fh_1 \in FH_1$ and $fh_2 \in FH_2$).

For the purpose of consistency semantics, a matched pair (Section 13.4.5, page 36) of split collective data access operations (e.g., `MPI_FILE_READ_ALL_BEGIN` and `MPI_FILE_READ_ALL_END`) compose a single data access operation. Similarly, a non-blocking data access routine (e.g., `MPI_FILE_IREAD`) and the routine which completes the request (e.g., `MPI_WAIT`) also compose a single data access operation. For all cases below, these data access operations are subject to the same constraints as blocking data access operations.

Advice to users. For an `MPI_FILE_IREAD` and `MPI_WAIT` pair, the operation begins when `MPI_FILE_IREAD` is called and ends when `MPI_WAIT` returns. (*End of advice to users.*)

Assume that A_1 and A_2 are two data access operations. Let D_1 (D_2) be the set of absolute byte displacements of every byte accessed in A_1 (A_2). The two data accesses *overlap* if $D_1 \cap D_2 \neq \emptyset$. The two data accesses *conflict* if they overlap and at least one is a write access.

Let SEQ_{fh} be a sequence of file operations on a single file handle, bracketed by `MPI_FILE_SYNC`s on that file handle. (Both opening and closing a file implicitly perform an `MPI_FILE_SYNC`.) SEQ_{fh} is a “write sequence” if any of the data access operations in the sequence are writes or if any of the file manipulation operations in the sequence change the state of the file (e.g., `MPI_FILE_SET_SIZE` or `MPI_FILE_PREALLOCATE`). Given two sequences, SEQ_1 and SEQ_2 , we say they are not *concurrent* if one sequence is guaranteed to completely precede the other (temporally).

The requirements for guaranteeing sequential consistency among all accesses to a particular file are divided into the three cases given below. If any of these requirements are not met, then the value of all data in that file is implementation dependent.

Case 1: $fh_1 \in FH_1$ All operations on fh_1 are sequentially consistent if atomic mode is set. If nonatomic mode is set, then all operations on fh_1 are sequentially consistent if they are either nonconcurrent, nonconflicting, or both.

Case 2: $fh_{1a} \in FH_1$ and $fh_{1b} \in FH_1$ Assume A_1 is a data access operation using fh_{1a} , and A_2 is a data access operation using fh_{1b} . If for any access A_1 , there is no access A_2 that conflicts with A_1 , then MPI guarantees sequential consistency.

However, unlike POSIX semantics, the default MPI semantics for conflicting accesses do not guarantee sequential consistency. If A_1 and A_2 conflict, sequential consistency can be guaranteed by either enabling atomic mode via the `MPI_FILE_SET_ATOMICITY` routine, or meeting the condition described in Case 3 below.

Case 3: $fh_1 \in FH_1$ and $fh_2 \in FH_2$ Consider access to a single file using file handles from distinct collective opens. In order to guarantee sequential consistency, `MPI_FILE_SYNC` must be used (both opening and closing a file implicitly perform an `MPI_FILE_SYNC`).

Sequential consistency is guaranteed among accesses to a single file if for any write sequence SEQ_1 to the file, there is no sequence SEQ_2 to the file which is *concurrent* with SEQ_1 . To guarantee sequential consistency when there are write sequences, `MPI_FILE_SYNC` must be used together with a mechanism that guarantees nonconcurrency of the sequences.

See the examples in Section 13.6.10, page 57, for further clarification of some of these consistency semantics.

`MPI_FILE_SET_ATOMICITY(fh, flag)`

INOUT	fh	file handle (handle)
IN	flag	true to set atomic mode, false to set nonatomic mode (logical)

`int MPI_File_set_atomicity(MPI_File fh, int flag)`

`MPI_FILE_SET_ATOMICITY(FH, FLAG, IERROR)`

INTEGER FH, IERROR

LOGICAL FLAG

`{void MPI::File::Set_atomicity(bool flag) (binding deprecated, see Section 15.2) }`

Let FH be the set of file handles created by one collective open. The consistency semantics for data access operations using FH is set by collectively calling `MPI_FILE_SET_ATOMICITY` on FH . `MPI_FILE_SET_ATOMICITY` is collective; all processes in the group must pass identical values for `fh` and `flag`. If `flag` is true, atomic mode is set; if `flag` is false, nonatomic mode is set.

Changing the consistency semantics for an open file only affects new data accesses. All completed data accesses are guaranteed to abide by the consistency semantics in effect during their execution. Nonblocking data accesses and split collective operations that have not completed (e.g., via `MPI_WAIT`) are only guaranteed to abide by nonatomic mode consistency semantics.

Advice to implementors. Since the semantics guaranteed by atomic mode are stronger than those guaranteed by nonatomic mode, an implementation is free to adhere to the more stringent atomic mode semantics for outstanding requests. (*End of advice to implementors.*)

MPI_FILE_GET_ATOMICITY(fh, flag)

IN	fh	file handle (handle)
OUT	flag	true if atomic mode, false if nonatomic mode (logical)

int MPI_File_get_atomicity(MPI_File fh, int *flag)

MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR)

INTEGER FH, IERROR

LOGICAL FLAG

{bool MPI::File::Get_atomicity() const(*binding deprecated, see Section 15.2*) }

MPI_FILE_GET_ATOMICITY returns the current consistency semantics for data access operations on the set of file handles created by one collective open. If **flag** is true, atomic mode is enabled; if **flag** is false, nonatomic mode is enabled.

MPI_FILE_SYNC(fh)

INOUT	fh	file handle (handle)
-------	----	----------------------

int MPI_File_sync(MPI_File fh)

MPI_FILE_SYNC(FH, IERROR)

INTEGER FH, IERROR

{void MPI::File::Sync() (*binding deprecated, see Section 15.2*) }

Calling **MPI_FILE_SYNC** with **fh** causes all previous writes to **fh** by the calling process to be transferred to the storage device. If other processes have made updates to the storage device, then all such updates become visible to subsequent reads of **fh** by the calling process. **MPI_FILE_SYNC** may be necessary to ensure sequential consistency in certain cases (see above).

MPI_FILE_SYNC is a collective operation.

The user is responsible for ensuring that all nonblocking requests and split collective operations on **fh** have been completed before calling **MPI_FILE_SYNC**—otherwise, the call to **MPI_FILE_SYNC** is erroneous.

MPI_FILE_ISYNC(fh, request)

INOUT	fh	file handle (handle)
OUT	request	request object (handle)


```
int MPI_File_ismync(MPI_File fh, MPI_Request *request)
```

```
MPI_FILE_ISYNC(FH, REQUEST, IERROR)
```

```
INTEGER FH, REQUEST, IERROR
```

`MPI_FILE_ISYNC` is a nonblocking version of the `MPI_FILE_SYNC` interface.

13.6.2 Random Access vs. Sequential Files

MPI distinguishes ordinary random access files from sequential stream files, such as pipes and tape files. Sequential stream files must be opened with the `MPI_MODE_SEQUENTIAL` flag set in the amode. For these files, the only permitted data access operations are shared file pointer reads and writes. Filetypes and etypes with holes are erroneous. In addition, the notion of file pointer is not meaningful; therefore, calls to `MPI_FILE_SEEK_SHARED` and `MPI_FILE_GET_POSITION_SHARED` are erroneous, and the pointer update rules specified for the data access routines do not apply. The amount of data accessed by a data access operation will be the amount requested unless the end of file is reached or an error is raised.

Rationale. This implies that reading on a pipe will always wait until the requested amount of data is available or until the process writing to the pipe has issued an end of file. (*End of rationale.*)

Finally, for some sequential files, such as those corresponding to magnetic tapes or streaming network connections, writes to the file may be destructive. In other words, a write may act as a truncate (a `MPI_FILE_SET_SIZE` with size set to the current position) followed by the write.

13.6.3 Progress

The progress rules of MPI are both a promise to users and a set of constraints on implementors. In cases where the progress rules restrict possible implementation choices more than the interface specification alone, the progress rules take precedence.

All blocking routines must complete in finite time unless an exceptional condition (such as resource exhaustion) causes an error.

Nonblocking data access routines inherit the following progress rule from nonblocking point to point communication: a nonblocking write is equivalent to a nonblocking send for which a receive is eventually posted, and a nonblocking read is equivalent to a nonblocking receive for which a send is eventually posted.

Finally, an implementation is free to delay progress of collective routines until all processes in the group associated with the collective call have invoked the routine. Once all processes in the group have invoked the routine, the progress rule of the equivalent noncollective routine must be followed.

13.6.4 Collective File Operations

Collective file operations are subject to the same restrictions as collective communication operations. For a complete discussion, please refer to the semantics set forth in Section 5.13 on page 200.

Collective file operations are collective over a dup of the communicator used to open the file—this duplicate communicator is implicitly specified via the file handle argument.

Different processes can pass different values for other arguments of a collective routine unless specified otherwise.

13.6.5 Type Matching

The type matching rules for I/O mimic the type matching rules for communication with one exception: if `etype` is `MPI_BYTE`, then this matches any `datatype` in a data access operation. In general, the `etype` of data items written must match the `etype` used to read the items, and for each data access operation, the current `etype` must also match the type declaration of the data access buffer.

Advice to users. In most cases, use of `MPI_BYTE` as a wild card will defeat the file interoperability features of MPI. File interoperability can only perform automatic conversion between heterogeneous data representations when the exact datatypes accessed are explicitly specified. (*End of advice to users.*)

13.6.6 Miscellaneous Clarifications

Once an I/O routine completes, it is safe to free any opaque objects passed as arguments to that routine. For example, the `comm` and `info` used in an `MPI_FILE_OPEN`, or the `etype` and `filetype` used in an `MPI_FILE_SET_VIEW`, can be freed without affecting access to the file. Note that for nonblocking routines and split collective operations, the operation must be completed before it is safe to reuse data buffers passed as arguments.

As in communication, datatypes must be committed before they can be used in file manipulation or data access operations. For example, the `etype` and `filetype` must be committed before calling `MPI_FILE_SET_VIEW`, and the `datatype` must be committed before calling `MPI_FILE_READ` or `MPI_FILE_WRITE`.

13.6.7 MPI_Offset Type

`MPI_Offset` is an integer type of size sufficient to represent the size (in bytes) of the largest file supported by MPI. Displacements and offsets are always specified as values of type `MPI_Offset`.

In Fortran, the corresponding integer is an integer of kind `MPI_OFFSET_KIND`, defined in `mpif.h` and the `mpi` module.

In Fortran 77 environments that do not support `KIND` parameters, `MPI_Offset` arguments should be declared as an `INTEGER` of suitable size. The language interoperability implications for `MPI_Offset` are similar to those for addresses (see Section 16.3, page 519).

13.6.8 Logical vs. Physical File Layout

MPI specifies how the data should be laid out in a virtual file structure (the view), not how that file structure is to be stored on one or more disks. Specification of the physical file structure was avoided because it is expected that the mapping of files to disks will be system specific, and any specific control over file layout would therefore restrict program portability. However, there are still cases where some information may be necessary to optimize file layout. This information can be provided as *hints* specified via *info* when a file is created (see Section 13.2.8, page 12).

13.6.9 File Size

The size of a file may be increased by writing to the file after the current end of file. The size may also be changed by calling *MPI size changing* routines, such as `MPI_FILE_SET_SIZE`. A call to a size changing routine does not necessarily change the file size. For example, calling `MPI_FILE_PREALLOCATE` with a size less than the current size does not change the size.

Consider a set of bytes that has been written to a file since the most recent call to a size changing routine, or since `MPI_FILE_OPEN` if no such routine has been called. Let the *high byte* be the byte in that set with the largest displacement. The file size is the larger of

- One plus the displacement of the high byte.
- The size immediately after the size changing routine, or `MPI_FILE_OPEN`, returned.

When applying consistency semantics, calls to `MPI_FILE_SET_SIZE` and `MPI_FILE_PREALLOCATE` are considered writes to the file (which conflict with operations that access bytes at displacements between the old and new file sizes), and `MPI_FILE_GET_SIZE` is considered a read of the file (which overlaps with all accesses to the file).

Advice to users. Any sequence of operations containing the collective routines `MPI_FILE_SET_SIZE` and `MPI_FILE_PREALLOCATE` is a write sequence. As such, sequential consistency in nonatomic mode is not guaranteed unless the conditions in Section 13.6.1, page 52, are satisfied. (*End of advice to users.*)

File pointer update semantics (i.e., file pointers are updated by the amount accessed) are only guaranteed if file size changes are sequentially consistent.

Advice to users. Consider the following example. Given two operations made by separate processes to a file containing 100 bytes: an `MPI_FILE_READ` of 10 bytes and an `MPI_FILE_SET_SIZE` to 0 bytes. If the user does not enforce sequential consistency between these two operations, the file pointer may be updated by the amount requested (10 bytes) even if the amount accessed is zero bytes. (*End of advice to users.*)

13.6.10 Examples

The examples in this section illustrate the application of the MPI consistency and semantics guarantees. These address

- conflicting accesses on file handles obtained from a single collective open, and
- all accesses on file handles obtained from two separate collective opens.

The simplest way to achieve consistency for conflicting accesses is to obtain sequential consistency by setting atomic mode. For the code below, process 1 will read either 0 or 10 integers. If the latter, every element of `b` will be 5. If nonatomic mode is set, the results of the read are undefined.

```
/* Process 0 */
int i, a[10] ;
int TRUE = 1;
```

```

1  for ( i=0;i<10;i++)
2      a[i] = 5 ;
3
4
5  MPI_File_open( MPI_COMM_WORLD, "workfile",
6                  MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0 ) ;
7  MPI_File_set_view( fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
8  MPI_File_set_atomicity( fh0, TRUE ) ;
9  MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status) ;
10 /* MPI_Barrier( MPI_COMM_WORLD ) ; */
11
12 /* Process 1 */
13 int  b[10] ;
14 int  TRUE = 1;
15 MPI_File_open( MPI_COMM_WORLD, "workfile",
16                 MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1 ) ;
17 MPI_File_set_view( fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
18 MPI_File_set_atomicity( fh1, TRUE ) ;
19 /* MPI_Barrier( MPI_COMM_WORLD ) ; */
20 MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status) ;

```

A user may guarantee that the write on process 0 precedes the read on process 1 by imposing temporal order with, for example, calls to MPI_BARRIER.

Advice to users. Routines other than MPI_BARRIER may be used to impose temporal order. In the example above, process 0 could use MPI_SEND to send a 0 byte message, received by process 1 using MPI_RECV. (*End of advice to users.*)

Alternatively, a user can impose consistency with nonatomic mode set:

```

24
25
26
27
28
29
30 /* Process 0 */
31 int  i, a[10] ;
32 for ( i=0;i<10;i++)
33     a[i] = 5 ;
34
35 MPI_File_open( MPI_COMM_WORLD, "workfile",
36                 MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0 ) ;
37 MPI_File_set_view( fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
38 MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status ) ;
39 MPI_File_sync( fh0 ) ;
40 MPI_Barrier( MPI_COMM_WORLD ) ;
41 MPI_File_sync( fh0 ) ;
42
43 /* Process 1 */
44 int  b[10] ;
45 MPI_File_open( MPI_COMM_WORLD, "workfile",
46                 MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1 ) ;
47 MPI_File_set_view( fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
48 MPI_File_sync( fh1 ) ;

```

```

MPI_Barrier( MPI_COMM_WORLD ) ;
MPI_File_sync( fh1 ) ;
MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status ) ;

```

The “sync-barrier-sync” construct is required because:

- The barrier ensures that the write on process 0 occurs before the read on process 1.
- The first sync guarantees that the data written by all processes is transferred to the storage device.
- The second sync guarantees that all data which has been transferred to the storage device is visible to all processes. (This does not affect process 0 in this example.)

The following program represents an erroneous attempt to achieve consistency by eliminating the apparently superfluous second “sync” call for each process.

```

/* ----- THIS EXAMPLE IS ERRONEOUS ----- */
/* Process 0 */
int i, a[10] ;
for ( i=0;i<10;i++)
    a[i] = 5 ;

MPI_File_open( MPI_COMM_WORLD, "workfile",
               MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0 ) ;
MPI_File_set_view( fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status ) ;
MPI_File_sync( fh0 ) ;
MPI_Barrier( MPI_COMM_WORLD ) ;

/* Process 1 */
int b[10] ;
MPI_File_open( MPI_COMM_WORLD, "workfile",
               MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1 ) ;
MPI_File_set_view( fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
MPI_Barrier( MPI_COMM_WORLD ) ;
MPI_File_sync( fh1 ) ;
MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status ) ;

/* ----- THIS EXAMPLE IS ERRONEOUS ----- */

```

The above program also violates the MPI rule against out-of-order collective operations and will deadlock for implementations in which MPI_FILE_SYNC blocks.

Advice to users. Some implementations may choose to implement MPI_FILE_SYNC as a temporally synchronizing function. When using such an implementation, the “sync-barrier-sync” construct above can be replaced by a single “sync.” The results of using such code with an implementation for which MPI_FILE_SYNC is not temporally synchronizing is undefined. (*End of advice to users.*)

Asynchronous I/O

The behavior of asynchronous I/O operations is determined by applying the rules specified above for synchronous I/O operations.

The following examples all access a preexisting file “myfile.” Word 10 in myfile initially contains the integer 2. Each example writes and reads word 10.

First consider the following code fragment:

```
int a = 4, b, TRUE=1;
MPI_File_open( MPI_COMM_WORLD, "myfile",
               MPI_MODE_RDWR, MPI_INFO_NULL, &fh ) ;
MPI_File_set_view( fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
/* MPI_File_set_atomicity( fh, TRUE ) ; Use this to set atomic mode. */
MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]) ;
MPI_File_iread_at(fh, 10, &b, 1, MPI_INT, &reqs[1]) ;
MPI_Waitall(2, reqs, statuses) ;
```

For asynchronous data access operations, MPI specifies that the access occurs at any time between the call to the asynchronous data access routine and the return from the corresponding request complete routine. Thus, executing either the read before the write, or the write before the read is consistent with program order. If atomic mode is set, then MPI guarantees sequential consistency, and the program will read either 2 or 4 into b. If atomic mode is not set, then sequential consistency is not guaranteed and the program may read something other than 2 or 4 due to the conflicting data access.

Similarly, the following code fragment does not order file accesses:

```
int a = 4, b;
MPI_File_open( MPI_COMM_WORLD, "myfile",
               MPI_MODE_RDWR, MPI_INFO_NULL, &fh ) ;
MPI_File_set_view( fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
/* MPI_File_set_atomicity( fh, TRUE ) ; Use this to set atomic mode. */
MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]) ;
MPI_File_iread_at(fh, 10, &b, 1, MPI_INT, &reqs[1]) ;
MPI_Wait(&reqs[0], &status) ;
MPI_Wait(&reqs[1], &status) ;
```

If atomic mode is set, either 2 or 4 will be read into b. Again, MPI does not guarantee sequential consistency in nonatomic mode.

On the other hand, the following code fragment:

```
int a = 4, b;
MPI_File_open( MPI_COMM_WORLD, "myfile",
               MPI_MODE_RDWR, MPI_INFO_NULL, &fh ) ;
MPI_File_set_view( fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]) ;
MPI_Wait(&reqs[0], &status) ;
MPI_File_iread_at(fh, 10, &b, 1, MPI_INT, &reqs[1]) ;
MPI_Wait(&reqs[1], &status) ;
```

defines the same ordering as:

```

int a = 4, b;
MPI_File_open( MPI_COMM_WORLD, "myfile",
               MPI_MODE_RDWR, MPI_INFO_NULL, &fh ) ;
MPI_File_set_view( fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
MPI_File_write_at(fh, 10, &a, 1, MPI_INT, &status ) ;
MPI_File_read_at(fh, 10, &b, 1, MPI_INT, &status ) ;

```

Since

- nonconcurrent operations on a single file handle are sequentially consistent, and
- the program fragments specify an order for the operations,

MPI guarantees that both program fragments will read the value 4 into `b`. There is no need to set atomic mode for this example.

Similar considerations apply to conflicting accesses of the form:

```

MPI_File_write_all_begin(fh,...) ;
MPI_File_iread(fh,...) ;
MPI_Wait(fh,...) ;
MPI_File_write_all_end(fh,...) ;

```

Recall that constraints governing consistency and semantics are not relevant to the following:

```

MPI_File_write_all_begin(fh,...) ;
MPI_File_read_all_begin(fh,...) ;
MPI_File_read_all_end(fh,...) ;
MPI_File_write_all_end(fh,...) ;

```

since split collective operations on the same file handle may not overlap (see Section 13.4.5, page 36).

13.7 I/O Error Handling

By default, communication errors are fatal—`MPI_ERRORS_ARE_FATAL` is the default error handler associated with `MPI_COMM_WORLD`. I/O errors are usually less catastrophic (e.g., “file not found”) than communication errors, and common practice is to catch these errors and continue executing. For this reason, MPI provides additional error facilities for I/O.

Advice to users. MPI does not specify the state of a computation after an erroneous MPI call has occurred. A high-quality implementation will support the I/O error handling facilities, allowing users to write programs using common practice for I/O. (*End of advice to users.*)

Like communicators, each file handle has an error handler associated with it. The MPI I/O error handling routines are defined in Section 8.3, page 298.

When MPI calls a user-defined error handler resulting from an error on a particular file handle, the first two arguments passed to the file error handler are the file handle and the error code. For I/O errors that are not associated with a valid file handle (e.g., in

MPI_FILE_OPEN or MPI_FILE_DELETE), the first argument passed to the error handler is MPI_FILE_NULL,

I/O error handling differs from communication error handling in another important aspect. By default, the predefined error handler for file handles is MPI_ERRORS_RETURN. The default file error handler has two purposes: when a new file handle is created (by MPI_FILE_OPEN), the error handler for the new file handle is initially set to the default error handler, and I/O routines that have no valid file handle on which to raise an error (e.g., MPI_FILE_OPEN or MPI_FILE_DELETE) use the default file error handler. The default file error handler can be changed by specifying MPI_FILE_NULL as the fh argument to MPI_FILE_SET_ERRHANDLER. The current value of the default file error handler can be determined by passing MPI_FILE_NULL as the fh argument to MPI_FILE_GET_ERRHANDLER.

Rationale. For communication, the default error handler is inherited from MPI_COMM_WORLD. In I/O, there is no analogous “root” file handle from which default properties can be inherited. Rather than invent a new global file handle, the default file error handler is manipulated as if it were attached to MPI_FILE_NULL. (*End of rationale.*)

13.8 I/O Error Classes

The implementation dependent error codes returned by the I/O routines can be converted into the error classes defined in Table 13.3.

In addition, calls to routines in this chapter may raise errors in other MPI classes, such as MPI_ERR_TYPE.

13.9 Examples

13.9.1 Double Buffering with Split Collective I/O

This example shows how to overlap computation and output. The computation is performed by the function `compute_buffer()`.

```
/*=====
 *
 * Function:          double_buffer
 *
 * Synopsis:
 *   void double_buffer(
 *       MPI_File fh,                ** IN
 *       MPI_Datatype buftype,       ** IN
 *       int bufcount                 ** IN
 *   )
 *
 * Description:
 *   Performs the steps to overlap computation with a collective write
 *   by using a double-buffering technique.
 *
 */
```


		1
		2
		3
		4
		5
		6
		7
		8
		9
		10
MPI_ERR_FILE	Invalid file handle	11
MPI_ERR_NOT_SAME	Collective argument not identical on all	12
	processes, or collective routines called in	13
	a different order by different processes	14
MPI_ERR_AMODE	Error related to the <code>amode</code> passed to	15
	<code>MPI_FILE_OPEN</code>	16
MPI_ERR_UNSUPPORTED_DATAREP	Unsupported <code>datarep</code> passed to	17
	<code>MPI_FILE_SET_VIEW</code>	18
MPI_ERR_UNSUPPORTED_OPERATION	Unsupported operation, such as seeking on	19
	a file which supports sequential access only	20
MPI_ERR_NO_SUCH_FILE	File does not exist	21
MPI_ERR_FILE_EXISTS	File exists	22
MPI_ERR_BAD_FILE	Invalid file name (e.g., path name too long)	23
MPI_ERR_ACCESS	Permission denied	24
MPI_ERR_NO_SPACE	Not enough space	25
MPI_ERR_QUOTA	Quota exceeded	26
MPI_ERR_READ_ONLY	Read-only file or file system	27
MPI_ERR_FILE_IN_USE	File operation could not be completed, as	28
	the file is currently open by some process	29
MPI_ERR_DUP_DATAREP	Conversion functions could not be regis-	30
	tered because a data representation identi-	31
	fier that was already defined was passed to	32
	<code>MPI_REGISTER_DATAREP</code>	33
MPI_ERR_CONVERSION	An error occurred in a user supplied data	34
	conversion function.	35
MPI_ERR_IO	Other I/O error	36

Table 13.3: I/O Error Classes

```

1  * Parameters:
2  *      fh              previously opened MPI file handle
3  *      buftype         MPI datatype for memory layout
4  *                     (Assumes a compatible view has been set on fh)
5  *      bufcount        # buftype elements to transfer
6  *-----*/
7
8  /* this macro switches which buffer "x" is pointing to */
9  #define TOGGLE_PTR(x) (((x)==(buffer1)) ? (x=buffer2) : (x=buffer1))
10
11 void double_buffer( MPI_File fh, MPI_Datatype buftype, int bufcount)
12 {
13
14     MPI_Status status;      /* status for MPI calls */
15     float *buffer1, *buffer2; /* buffers to hold results */
16     float *compute_buf_ptr;  /* destination buffer */
17                             /* for computing */
18     float *write_buf_ptr;    /* source for writing */
19     int done;               /* determines when to quit */
20
21     /* buffer initialization */
22     buffer1 = (float *)
23             malloc(bufcount*sizeof(float)) ;
24     buffer2 = (float *)
25             malloc(bufcount*sizeof(float)) ;
26     compute_buf_ptr = buffer1 ; /* initially point to buffer1 */
27     write_buf_ptr   = buffer1 ; /* initially point to buffer1 */
28
29
30     /* DOUBLE-BUFFER prolog:
31      *   compute buffer1; then initiate writing buffer1 to disk
32      */
33     compute_buffer(compute_buf_ptr, bufcount, &done);
34     MPI_File_write_all_begin(fh, write_buf_ptr, bufcount, buftype);
35
36     /* DOUBLE-BUFFER steady state:
37      *   Overlap writing old results from buffer pointed to by write_buf_ptr
38      *   with computing new results into buffer pointed to by compute_buf_ptr.
39      *
40      *   There is always one write-buffer and one compute-buffer in use
41      *   during steady state.
42      */
43     while (!done) {
44         TOGGLE_PTR(compute_buf_ptr);
45         compute_buffer(compute_buf_ptr, bufcount, &done);
46         MPI_File_write_all_end(fh, write_buf_ptr, &status);
47         TOGGLE_PTR(write_buf_ptr);
48         MPI_File_write_all_begin(fh, write_buf_ptr, bufcount, buftype);

```

```

}
/* DOUBLE-BUFFER epilog:
 *   wait for final write to complete.
 */
MPI_File_write_all_end(fh, write_buf_ptr, &status);

/* buffer cleanup */
free(buffer1);
free(buffer2);
}

```

13.9.2 Subarray Filetype Constructor

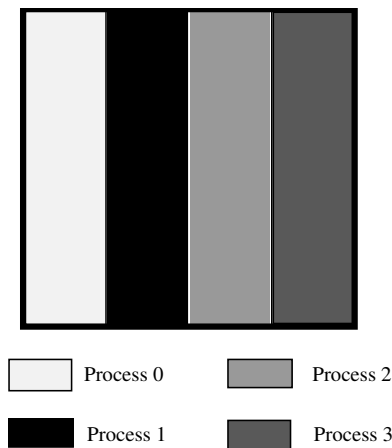


Figure 13.4: Example array file layout

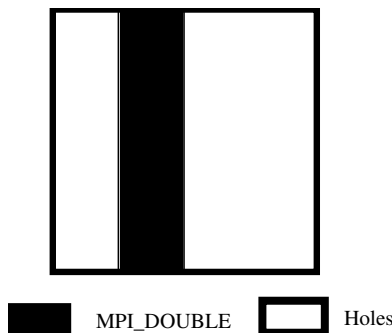


Figure 13.5: Example local array filetype for process 1

Assume we are writing out a 100x100 2D array of double precision floating point numbers that is distributed among 4 processes such that each process has a block of 25 columns (e.g., process 0 has columns 0-24, process 1 has columns 25-49, etc.; see Figure 13.4). To create the filetypes for each process one could use the following C program (see Section 4.1.3 on page 12):

```

1  double subarray[100][25];
2  MPI_Datatype filetype;
3  int sizes[2], subsizes[2], starts[2];
4  int rank;
5
6  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
7  sizes[0]=100; sizes[1]=100;
8  subsizes[0]=100; subsizes[1]=25;
9  starts[0]=0; starts[1]=rank*subsizes[1];
10
11 MPI_Type_create_subarray(2, sizes, subsizes, starts, MPI_ORDER_C,
12                          MPI_DOUBLE, &filetype);
13

```

Or, equivalently in Fortran:

```

15  double precision subarray(100,25)
16  integer filetype, rank, ierror
17  integer sizes(2), subsizes(2), starts(2)
18
19
20  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
21  sizes(1)=100
22  sizes(2)=100
23  subsizes(1)=100
24  subsizes(2)=25
25  starts(1)=0
26  starts(2)=rank*subsizes(2)
27
28  call MPI_TYPE_CREATE_SUBARRAY(2, sizes, subsizes, starts, &
29                               MPI_ORDER_FORTRAN, MPI_DOUBLE_PRECISION, &
30                               filetype, ierror)
31

```

The generated filetype will then describe the portion of the file contained within the process's subarray with holes for the space taken by the other processes. Figure 13.5 shows the filetype created for process 1.

Bibliography

- [1] Rajesh Bordawekar, Juan Miguel del Rosario, and Alok Choudhary. Design and evaluation of primitives for parallel I/O. In *Proceedings of Supercomputing '93*, pages 452–461, 1993. [13.1](#)
- [2] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 56–70, 1993. Also published in *Computer Architecture News* 21(5), December 1993, pages 31–38. [13.1](#)
- [3] Institute of Electrical and Electronics Engineers, New York. *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*, 1985. [13.5.2](#)
- [4] International Organization for Standardization, Geneva. *Information processing — 8-bit single-byte coded graphic character sets — Part 1: Latin alphabet No. 1*, 1987. [13.5.2](#)
- [5] International Organization for Standardization, Geneva. *Information technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language]*, December 1996. [13.2.1](#)
- [6] David Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74, November 1994. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994. [13.1](#)
- [7] Bill Nitzberg. Performance of the iPSC/860 Concurrent File System. Technical Report RND-92-020, NAS Systems Division, NASA Ames, December 1992. [13.1](#)
- [8] William J. Nitzberg. *Collective Parallel I/O*. PhD thesis, Department of Computer and Information Science, University of Oregon, December 1995. [13.1](#)
- [9] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*, December 1995. [13.1](#)
- [10] Rajeev Thakur and Alok Choudhary. An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays. *Scientific Programming*, 5(4):301–317, Winter 1996. [13.1](#)
- [11] *The Unicode Standard, Version 2.0*. Addison-Wesley, 1996. ISBN 0-201-48345-9. [13.5.2](#)

Index

CONST:access_style, 13
CONST:cb_block_size, 14
CONST:cb_buffer_size, 14
CONST:cb_nodes, 14
CONST:chunked, 14
CONST:chunked_item, 14
CONST:chunked_size, 14
CONST:collective_buffering, 13
CONST:external32, 43
CONST:false, 14
CONST:file_perm, 13, 14
CONST:filename, 14
CONST:internal, 43
CONST:io_node_list, 14
CONST:MPI::Aint, 45, 48
CONST:MPI::File, 3, 5, 6, 8–13, 15, 17, 18, 21–35, 37–41, 45, 53, 54
CONST:MPI::Group, 10
CONST:MPI::Info, 3, 5, 7, 12, 13, 15, 17
CONST:MPI::Offset, 8–10, 15, 17, 18, 21–24, 29, 30, 35, 37, 38, 49, 56
CONST:MPI::Request, 5–10, 17, 24, 28, 29, 32, 33, 54
CONST:MPI::Status, 21–23, 25–27, 31, 32, 34, 38–41
CONST:MPI_Aint, 45, 48
CONST:MPI_BYTE, 2, 43, 44, 55, 56
CONST:MPI_COMM_SELF, 3
CONST:MPI_COMM_WORLD, 42, 61, 62
CONST:MPI_COMPLEX, 46
CONST:MPI_CONVERSION_FN_NULL, 50
CONST:MPI_DISPLACEMENT_CURRENT, 16
CONST:MPI_DOUBLE_COMPLEX, 46
CONST:MPI_ERR_ACCESS, 7, 62
CONST:MPI_ERR_AMODE, 5, 62
CONST:MPI_ERR_BAD_FILE, 62
CONST:MPI_ERR_CONVERSION, 51, 62
CONST:MPI_ERR_DUP_DATAREP, 48, 62
CONST:MPI_ERR_FILE, 62
CONST:MPI_ERR_FILE_EXISTS, 62
CONST:MPI_ERR_FILE_IN_USE, 7, 62
CONST:MPI_ERR_IN_STATUS, 21
CONST:MPI_ERR_IO, 62
CONST:MPI_ERR_NO_SPACE, 62
CONST:MPI_ERR_NO_SUCH_FILE, 7, 62
CONST:MPI_ERR_NOT_SAME, 62
CONST:MPI_ERR_QUOTA, 62
CONST:MPI_ERR_READ_ONLY, 62
CONST:MPI_ERR_UNSUPPORTED_DATAREP, 62
CONST:MPI_ERR_UNSUPPORTED_OPERATION, 62
CONST:MPI_ERRORS_ARE_FATAL, 61
CONST:MPI_ERRORS_RETURN, 61
CONST:MPI_File, 3, 5, 6, 8–13, 15, 17, 18, 21–35, 37–41, 45, 53, 54
CONST:MPI_FILE_NULL, 6, 61, 62
CONST:MPI_FLOAT, 45
CONST:MPI_Group, 10
CONST:MPI_Info, 3, 5, 7, 12, 13, 15, 17
CONST:MPI_INFO_NULL, 5, 7, 17
CONST:MPI_INT, 45, 46
CONST:MPI_INTEGER2, 45
CONST:MPI_LB, 44
CONST:MPI_MAX_DATAREP_STRING, 18, 48
CONST:MPI_MODE_APPEND, 4, 5
CONST:MPI_MODE_CREATE, 4, 5, 14
CONST:MPI_MODE_DELETE_ON_CLOSE, 4–6
CONST:MPI_MODE_EXCL, 4, 5
CONST:MPI_MODE_RDONLY, 4, 5, 12
CONST:MPI_MODE_RDWR, 4, 5
CONST:MPI_MODE_SEQUENTIAL, 4, 5, 8, 9, 16, 21, 25, 35, 55
CONST:MPI_MODE_UNIQUE_OPEN, 4, 5
CONST:MPI_MODE_WRONLY, 4, 5
CONST:MPI_Offset, 8–10, 15, 17, 18, 21–24, 29, 30, 35, 37, 38, 49, 56, 56

CONST:MPI_OFFSET_KIND, 56	MPI_FILE_GET_ATOMICHITY, 54	1
CONST:MPI_PACKED, 46	MPI_FILE_GET_ATOMICHITY(fh, flag), 54	2
CONST:MPI_REAL, 46	MPI_FILE_GET_BYTE_OFFSET, 25 , 30 ,	3
CONST:MPI_Request, 5–10 , 17 , 24 , 28 , 29 ,	36	4
32 , 33 , 54	MPI_FILE_GET_BYTE_OFFSET(fh, offset,	5
CONST:MPI_SEEK_CUR, 29 , 35	disp), 30	6
CONST:MPI_SEEK_END, 29 , 35	MPI_FILE_GET_ERRHANDLER, 62	7
CONST:MPI_SEEK_SET, 29 , 30 , 35	MPI_FILE_GET_GROUP, 11	8
CONST:MPI_Status, 21–23 , 25–27 , 31 , 32 ,	MPI_FILE_GET_GROUP(fh, group), 10	9
34 , 38–41	MPI_FILE_GET_INFO, 13 , 14	10
CONST:MPI_STATUS_IGNORE, 21	MPI_FILE_GET_INFO(fh, info_used), 13	11
CONST:MPI_SUCCESS, 51	MPI_FILE_GET_POSITION, 30	12
CONST:MPI_UB, 44	MPI_FILE_GET_POSITION(fh, offset), 30	13
CONST:MPI_WCHAR, 46	MPI_FILE_GET_POSITION_SHARED, 35 ,	14
CONST:native, 43	55	15
CONST:nb_proc, 14	MPI_FILE_GET_POSITION_SHARED(fh, off-	16
CONST:num_io_nodes, 14	set), 35	17
CONST:random, 13	MPI_FILE_GET_SIZE, 10 , 57	18
CONST:read_mostly, 13	MPI_FILE_GET_SIZE(fh, size), 10	19
CONST:read_once, 13	MPI_FILE_GET_TYPE_EXTENT, 44 , 51	20
CONST:reverse_sequential, 13	MPI_FILE_GET_TYPE_EXTENT(fh, datatype,	21
CONST:sequential, 13	extent), 45	22
CONST:striping_factor, 15	MPI_FILE_GET_VIEW, 18	23
CONST:striping_unit, 15	MPI_FILE_GET_VIEW(fh, disp, etype, file-	24
CONST:true, 14	type, datarep), 18	25
CONST:write_mostly, 13	MPI_FILE_ICLOSE, 7	26
CONST:write_once, 13	MPI_FILE_ICLOSE(fh, request), 6	27
	MPI_FILE_IDELETE, 7	28
EXAMPLES:MPI_FILE_CLOSE, 25 , 28	MPI_FILE_IDELETE(filename, info, request),	29
EXAMPLES:MPI_FILE_GET_AMODE, 11	7	30
EXAMPLES:MPI_FILE_IREAD, 28	MPI_FILE_IGET_SIZE, 10	31
EXAMPLES:MPI_FILE_OPEN, 25 , 28	MPI_FILE_IGET_SIZE(fh, size, request), 10	32
EXAMPLES:MPI_FILE_READ, 25	MPI_FILE_IOPEN, 6	33
EXAMPLES:MPI_FILE_SET_ATOMICHITY,	MPI_FILE_IOPEN(comm, filename, amode,	34
57	info, fh, request), 5	35
EXAMPLES:MPI_FILE_SET_VIEW, 25 , 28	MPI_FILE_IPREALLOCATE, 10	36
EXAMPLES:MPI_FILE_SYNC, 58	MPI_FILE_IPREALLOCATE(fh, size, request),	37
EXAMPLES:MPI_TYPE_CREATE_SUBARRAY,	9	38
64	MPI_FILE_IREAD, 19 , 28 , 36 , 52	39
EXAMPLES:MPI_WAIT, 28	MPI_FILE_IREAD(fh, buf, count, datatype,	40
	request), 28	41
MPI_BARRIER, 58	MPI_FILE_IREAD_AT, 19 , 24	42
MPI_FILE_CLOSE, 3 , 6 , 7	MPI_FILE_IREAD_AT(fh, offset, buf, count,	43
MPI_FILE_CLOSE(fh), 6	datatype, request), 24	44
MPI_FILE_DELETE, 5–7 , 12 , 14 , 61	MPI_FILE_IREAD_SHARED, 19 , 32	45
MPI_FILE_DELETE(filename, info), 7	MPI_FILE_IREAD_SHARED(fh, buf, count,	46
MPI_FILE_GET_AMODE, 11	datatype, request), 32	47
MPI_FILE_GET_AMODE(fh, amode), 11	MPI_FILE_ISET_SIZE, 9	48

- MPI_FILE_ISET_SIZE(fh, size, request), [8](#)
 MPI_FILE_ISET_VIEW, [17](#)
 MPI_FILE_ISET_VIEW(fh, disp, etype, file-
 type, datarep, info, request), [17](#)
 MPI_FILE_ISYNC, [54](#)
 MPI_FILE_ISYNC(fh, request), [54](#)
 MPI_FILE_IWRITE, [19](#), [29](#)
 MPI_FILE_IWRITE(fh, buf, count, datatype, request), [29](#)
 MPI_FILE_IWRITE_AT, [19](#), [24](#)
 MPI_FILE_IWRITE_AT(fh, offset, buf, count, datatype, request), [24](#)
 MPI_FILE_IWRITE_SHARED, [19](#), [33](#)
 MPI_FILE_IWRITE_SHARED(fh, buf, count, datatype, request), [33](#)
 MPI_FILE_OPEN, [3–6](#), [12–14](#), [16](#), [30](#), [56](#), [57](#), [61](#), [62](#)
 MPI_FILE_OPEN(comm, filename, amode, info, fh), [3](#)
 MPI_FILE_PREALLOCATE, [8–10](#), [52](#), [56](#), [57](#)
 MPI_FILE_PREALLOCATE(fh, size), [9](#)
 MPI_FILE_READ, [18](#), [19](#), [25](#), [27](#), [28](#), [56](#), [57](#)
 MPI_FILE_READ(fh, buf, count, datatype, status), [25](#)
 MPI_FILE_READ_ALL, [19](#), [27](#), [36](#), [37](#)
 MPI_FILE_READ_ALL(fh, buf, count, datatype, status), [26](#)
 MPI_FILE_READ_ALL_BEGIN, [19](#), [36](#), [37](#), [52](#)
 MPI_FILE_READ_ALL_BEGIN(fh, buf, count, datatype), [39](#)
 MPI_FILE_READ_ALL_END, [19](#), [36](#), [37](#), [52](#)
 MPI_FILE_READ_ALL_END(fh, buf, status), [39](#)
 MPI_FILE_READ_AT, [19](#), [22](#), [24](#)
 MPI_FILE_READ_AT(fh, offset, buf, count, datatype, status), [21](#)
 MPI_FILE_READ_AT_ALL, [19](#), [22](#)
 MPI_FILE_READ_AT_ALL(fh, offset, buf, count, datatype, status), [22](#)
 MPI_FILE_READ_AT_ALL_BEGIN, [19](#)
 MPI_FILE_READ_AT_ALL_BEGIN(fh, offset, buf, count, datatype), [37](#)
 MPI_FILE_READ_AT_ALL_END, [19](#)
 MPI_FILE_READ_AT_ALL_END(fh, buf, status), [38](#)
 MPI_FILE_READ_ORDERED, [19](#), [34](#)
 MPI_FILE_READ_ORDERED(fh, buf, count, datatype, status), [34](#)
 MPI_FILE_READ_ORDERED_BEGIN, [19](#)
 MPI_FILE_READ_ORDERED_BEGIN(fh, buf, count, datatype), [40](#)
 MPI_FILE_READ_ORDERED_END, [19](#)
 MPI_FILE_READ_ORDERED_END(fh, buf, status), [41](#)
 MPI_FILE_READ_SHARED, [19](#), [31](#), [32](#), [34](#)
 MPI_FILE_READ_SHARED(fh, buf, count, datatype, status), [31](#)
 MPI_FILE_SEEK, [29](#), [30](#)
 MPI_FILE_SEEK(fh, offset, whence), [29](#)
 MPI_FILE_SEEK_SHARED, [35](#), [36](#), [55](#)
 MPI_FILE_SEEK_SHARED(fh, offset, whence), [35](#)
 MPI_FILE_SET_ATOMICITY, [5](#), [53](#)
 MPI_FILE_SET_ATOMICITY(fh, flag), [53](#)
 MPI_FILE_SET_ERRHANDLER, [62](#)
 MPI_FILE_SET_INFO, [12–14](#)
 MPI_FILE_SET_INFO(fh, info), [12](#)
 MPI_FILE_SET_SIZE, [8](#), [9](#), [52](#), [55–57](#)
 MPI_FILE_SET_SIZE(fh, size), [8](#)
 MPI_FILE_SET_VIEW, [4](#), [12–17](#), [30](#), [36](#), [43](#), [48](#), [56](#), [62](#)
 MPI_FILE_SET_VIEW(fh, disp, etype, file-type, datarep, info), [15](#)
 MPI_FILE_SYNC, [6](#), [18](#), [52–54](#), [59](#)
 MPI_FILE_SYNC(fh), [54](#)
 MPI_FILE_WRITE, [18](#), [19](#), [27–29](#), [56](#)
 MPI_FILE_WRITE(fh, buf, count, datatype, status), [27](#)
 MPI_FILE_WRITE_ALL, [19](#), [28](#)
 MPI_FILE_WRITE_ALL(fh, buf, count, datatype, status), [27](#)
 MPI_FILE_WRITE_ALL_BEGIN, [19](#)
 MPI_FILE_WRITE_ALL_BEGIN(fh, buf, count, datatype), [40](#)
 MPI_FILE_WRITE_ALL_END, [19](#)
 MPI_FILE_WRITE_ALL_END(fh, buf, status), [40](#)
 MPI_FILE_WRITE_AT, [19](#), [23](#), [24](#)
 MPI_FILE_WRITE_AT(fh, offset, buf, count, datatype, status), [22](#)
 MPI_FILE_WRITE_AT_ALL, [19](#), [23](#)
 MPI_FILE_WRITE_AT_ALL(fh, offset, buf, count, datatype, status), [23](#)

MPI_FILE_WRITE_AT_ALL_BEGIN, 19	1
MPI_FILE_WRITE_AT_ALL_BEGIN(fh, off-	2
set, buf, count, datatype), 38	3
MPI_FILE_WRITE_AT_ALL_END, 19	4
MPI_FILE_WRITE_AT_ALL_END(fh, buf,	5
status), 38	6
MPI_FILE_WRITE_ORDERED, 19 , 33 , 34	7
MPI_FILE_WRITE_ORDERED(fh, buf, count,	8
datatype, status), 34	9
MPI_FILE_WRITE_ORDERED_BEGIN, 19	10
MPI_FILE_WRITE_ORDERED_BEGIN(fh,	11
buf, count, datatype), 41	12
MPI_FILE_WRITE_ORDERED_END, 19	13
MPI_FILE_WRITE_ORDERED_END(fh, buf,	14
status), 41	15
MPI_FILE_WRITE_SHARED, 19 , 32–34	16
MPI_FILE_WRITE_SHARED(fh, buf, count,	17
datatype, status), 32	18
MPI_FINALIZE, 3	19
MPI_GET_COUNT, 21	20
MPI_GET_ELEMENTS, 21	21
MPI_INFO_FREE, 13	22
MPI_PACK, 46 , 49	23
MPI_RECV, 58	24
MPI_REGISTER_DATAREP, 48–50 , 62	25
MPI_REGISTER_DATAREP(datarep, read_conversion_fn,	26
write_conversion_fn,	27
dtype_file_extent_fn, extra_state), 48	28
MPI_SEND, 4 , 58	29
MPI_TEST, 20 , 21	30
MPI_TEST_CANCELLED, 21	31
MPI_TYPE_CONTIGUOUS, 2 , 44	32
MPI_TYPE_CREATE_F90_COMPLEX, 46	33
MPI_TYPE_CREATE_F90_INTEGER, 46	34
MPI_TYPE_CREATE_F90_REAL, 46	35
MPI_TYPE_CREATE_RESIZED, 44	36
MPI_UNPACK, 49	37
MPI_WAIT, 20 , 21 , 36 , 52 , 53	38
read_conversion_fn, 49 , 50	39
TYPEDEF:MPI_Datarep_conversion_function(void *user-	41
buf, MPI_Datatype datatype, int count,	42
void *filebuf, MPI_Offset position,	43
void *extra_state), 49	44
TYPEDEF:MPI_Datarep_extent_function(MPI_Datatype datatype,	45
MPI_Aint *file_extent, void *extra_state),	46
48	47
	48