MPI: A Message-Passing Interface Standard

Version 3.0 ticket0.

Message Passing Interface Forum

Draft July 4th, 2011

Contents

List of Figures

List of Tables

Chapter 1

MPI Environmental Management

This chapter discusses routines for getting and, where appropriate, setting various parameters that relate to the MPI implementation and the execution environment (such as error handling). The procedures for entering and leaving the MPI execution environment are also described here.

1.1 Implementation Information

1.1.1 Version Inquiries

In order to cope with changes to the MPI Standard, there are both compile-time and run-time ways to determine which version of the standard is in use in the environment one is using.

The "version" will be represented by two separate integers, for the version and subversion: In C and C++,

```
#define MPI_VERSION    2
#define MPI_SUBVERSION 2
```

in Fortran,

```
INTEGER MPI_VERSION, MPI_SUBVERSION
PARAMETER (MPI_VERSION    = 2)
PARAMETER (MPI_SUBVERSION = 2)
```

For runtime determination,

MPI_GET_VERSION(version, subversion)

OUT	version	version number (integer)
OUT	subversion	subversion number (integer)

```
int MPI_Get_version(int *version, int *subversion)
```

```
MPI_GET_VERSION(VERSION, SUBVERSION, IERROR)
    INTEGER VERSION, SUBVERSION, IERROR
```

{void MPI::Get_version(int& version, int& subversion) *(binding deprecated, see Section ??)* }

ticket204. ticket0-new. []Valid (MPI_VERSION, MPI_SUBVERSION) pairs in this and previous versions of the MPI standard are [](3,0), (2,2), (2,1), (2,0), and (1,2).

ticket204. ticket204. [ ] MPI_GET_LIBRARY_VERSION(version, resultlen)

OUT	version	version string (string)
OUT	resultlen	Length (in printable characters) of the result returned in version (integer)

```
int MPI_Get_library_version(char *version, int *resultlen)
```

```
MPI_GET_LIBRARY_VERSION(VERSION, RESULTEN, IERROR)
    CHARACTER*(*) VERSION
    INTEGER RESULTLEN,IERROR
```

This routine returns a string representing the version of the MPI library. The version argument is a character string for maximum flexibility.

> *Advice to implementors.* An implementation of MPI should return a different string for every change to its source code or build that could be visible to the user. (*End of advice to implementors.*)

The argument version must represent storage that is MPI_MAX_LIBRARY_VERSION_STRING characters long. MPI_GET_LIBRARY_VERSION may write up to this many characters into version.

The number of characters actually written is returned in the output argument, resultlen. In C, a null character is additionally stored at name[resultlen]. The resultlen cannot be larger than MPI_MAX_LIBRARY_VERSION_STRING - 1. In Fortran, version is padded on the right with blank characters. The resultlen cannot be larger than MPI_MAX_LIBRARY_VERSION_STRING.

ticket204. ticket204. MPI_GET_VERSION [is one]and MPI_GET_LIBRARY_VERSION are two of the few functions that can be called before MPI_INIT and after MPI_FINALIZE. [Valid (MPI_VERSION, MPI_SUBVERSION) pairs in this and previous versions of the MPI standard are (2,2), (2,1), (2,0), and (1,2).]

1.1.2 Environmental Inquiries

A set of attributes that describe the execution environment are attached to the communicator MPI_COMM_WORLD when MPI is initialized. The value of these attributes can be inquired by using the function MPI_COMM_GET_ATTR described in Chapter ??. It is erroneous to delete these attributes, free their keys, or change their values.

The list of predefined attribute keys include

MPI_TAG_UB Upper bound for tag value.

MPI_HOST Host process rank, if such exists, MPI_PROC_NULL, otherwise.

MPI_IO rank of a node that has regular I/O facilities (possibly myrank). Nodes in the same communicator may return different values for this parameter.

MPI_WTIME_IS_GLOBAL Boolean variable that indicates whether clocks are synchronized.

Vendors may add implementation specific parameters (such as node number, real memory size, virtual memory size, etc.)

These predefined attributes do not change value between MPI initialization (MPI_INIT and MPI completion (MPI_FINALIZE), and cannot be updated or deleted by users.

> *Advice to users.* Note that in the C binding, the value returned by these attributes is a *pointer* to an `int` containing the requested value. (*End of advice to users.*)

The required parameter values are discussed in more detail below:

Tag Values

Tag values range from 0 to the value returned for MPI_TAG_UB inclusive. These values are guaranteed to be unchanging during the execution of an MPI program. In addition, the tag upper bound value must be *at least* 32767. An MPI implementation is free to make the value of MPI_TAG_UB larger than this; for example, the value $2^{30} - 1$ is also a legal value for MPI_TAG_UB.

The attribute MPI_TAG_UB has the same value on all processes of MPI_COMM_WORLD.

Host Rank

The value returned for MPI_HOST gets the rank of the `HOST` process in the group associated with communicator MPI_COMM_WORLD, if there is such. MPI_PROC_NULL is returned if there is no host. MPI does not specify what it means for a process to be a `HOST`, nor does it requires that a `HOST` exists.

The attribute MPI_HOST has the same value on all processes of MPI_COMM_WORLD.

IO Rank

The value returned for MPI_IO is the rank of a processor that can provide language-standard I/O facilities. For Fortran, this means that all of the Fortran I/O operations are supported (e.g., `OPEN`, `REWIND`, `WRITE`). For C and C++, this means that all of the ISO C and C++, I/O operations are supported (e.g., `fopen`, `fprintf`, `lseek`).

If every process can provide language-standard I/O, then the value MPI_ANY_SOURCE will be returned. Otherwise, if the calling process can provide language-standard I/O, then its rank will be returned. Otherwise, if some process can provide language-standard I/O then the rank of one such process will be returned. The same value need not be returned by all processes. If no process can provide language-standard I/O, then the value MPI_PROC_NULL will be returned.

> *Advice to users.* Note that input is not collective, and this attribute does *not* indicate which process can or does provide input. (*End of advice to users.*)

Clock Synchronization

The value returned for MPI_WTIME_IS_GLOBAL is 1 if clocks at all processes in MPI_COMM_WORLD are synchronized, 0 otherwise. A collection of clocks is considered synchronized if explicit effort has been taken to synchronize them. The expectation is that the variation in time, as measured by calls to MPI_WTIME, will be less then one half the round-trip time for an MPI message of length zero. If time is measured at a process just before a send and at another process just after a matching receive, the second time should be always higher than the first one.

The attribute MPI_WTIME_IS_GLOBAL need not be present when the clocks are not synchronized (however, the attribute key MPI_WTIME_IS_GLOBAL is always valid). This attribute may be associated with communicators other then MPI_COMM_WORLD.

The attribute MPI_WTIME_IS_GLOBAL has the same value on all processes of MPI_COMM_WORLD.

MPI_GET_PROCESSOR_NAME(name, resultlen)

OUT	name	A unique specifier for the actual (as opposed to virtual) node [](string)
OUT	resultlen	Length (in printable characters) of the result returned in name [](integer)

ticket0-new. ticket0-new.

```
int MPI_Get_processor_name(char *name, int *resultlen)

MPI_GET_PROCESSOR_NAME( NAME, RESULTLEN, IERROR)
    CHARACTER*(*) NAME
    INTEGER RESULTLEN,IERROR
```

{void MPI::Get_processor_name(char* name, int& resultlen) *(binding deprecated, see Section* ??*)* }

This routine returns the name of the processor on which it was called at the moment of the call. The name is a character string for maximum flexibility. From this value it must be possible to identify a specific piece of hardware; possible values include "processor 9 in rack 4 of mpp.cs.org" and "231" (where 231 is the actual processor number in the running homogeneous system). The argument name must represent storage that is at least MPI_MAX_PROCESSOR_NAME characters long. MPI_GET_PROCESSOR_NAME may write up to this many characters into name.

The number of characters actually written is returned in the output argument, resultlen. In C, a null character is additionally stored at name[resultlen]. The resultlen cannot be larger ticket0-new. [then]than MPI_MAX_PROCESSOR_NAME-1. In Fortran, name is padded on the right with ticket0-new. blank characters. The resultlen cannot be larger [then]than MPI_MAX_PROCESSOR_NAME.

Rationale. This function allows MPI implementations that do process migration to return the current processor. Note that nothing in MPI *requires* or defines process migration; this definition of MPI_GET_PROCESSOR_NAME simply allows such an implementation. (*End of rationale.*)

Advice to users. The user must provide at least MPI_MAX_PROCESSOR_NAME space to write the processor name — processor names can be this long. The user should examine the output argument, resultlen, to determine the actual length of the name. (*End of advice to users.*)

The constant MPI_BSEND_OVERHEAD provides an upper bound on the fixed overhead per message buffered by a call to MPI_BSEND (see Section **??**).

1.2 Memory Allocation

In some systems, message-passing and remote-memory-access (RMA) operations run faster when accessing specially allocated memory (e.g., memory that is shared by the other processes in the communicating group on an SMP). MPI provides a mechanism for allocating and freeing such special memory. The use of such memory for message-passing or RMA is not mandatory, and this memory can be used without restrictions as any other dynamically allocated memory. However, implementations may restrict the use of the MPI_WIN_LOCK and MPI_WIN_UNLOCK functions to windows allocated in such memory (see Section **??**.)

MPI_ALLOC_MEM(size, info, baseptr)

IN	size	size of memory segment in bytes (non-negative integer)
IN	info	info argument (handle)
OUT	baseptr	pointer to beginning of memory segment allocated

```
int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr)

MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
    INTEGER INFO, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR

{void* MPI::Alloc_mem(MPI::Aint size, const MPI::Info& info)(binding
              deprecated, see Section ??) }
```

The info argument can be used to provide directives that control the desired location of the allocated memory. Such a directive does not affect the semantics of the call. Valid info values are implementation-dependent; a null directive value of info = MPI_INFO_NULL is always valid.

The function MPI_ALLOC_MEM may return an error code of class MPI_ERR_NO_MEM to indicate it failed because memory is exhausted.

MPI_FREE_MEM(base)

IN	base	initial address of memory segment allocated by MPI_ALLOC_MEM (choice)

```
int MPI_Free_mem(void *base)

MPI_FREE_MEM(BASE, IERROR)
    <type> BASE(*)
```

```
    INTEGER IERROR
```

{void MPI::Free_mem(void *base) *(binding deprecated, see Section ??)* }

The function MPI_FREE_MEM may return an error code of class MPI_ERR_BASE to indicate an invalid base argument.

> *Rationale.* The C and C++ bindings of MPI_ALLOC_MEM and MPI_FREE_MEM are similar to the bindings for the malloc and free C library calls: a call to MPI_Alloc_mem(..., &base) should be paired with a call to MPI_Free_mem(base) (one less level of indirection). Both arguments are declared to be of same type void* so as to facilitate type casting. The Fortran binding is consistent with the C and C++ bindings: the Fortran MPI_ALLOC_MEM call returns in baseptr the (integer valued) address of the allocated memory. The base argument of MPI_FREE_MEM is a choice argument, which passes (a reference to) the variable stored at that location. (*End of rationale.*)

> *Advice to implementors.* If MPI_ALLOC_MEM allocates special memory, then a design similar to the design of C malloc and free functions has to be used, in order to find out the size of a memory segment, when the segment is freed. If no special memory is used, MPI_ALLOC_MEM simply invokes malloc, and MPI_FREE_MEM invokes free.
>
> A call to MPI_ALLOC_MEM can be used in shared memory systems to allocate memory in a shared memory segment. (*End of advice to implementors.*)

Example 1.1

Example of use of MPI_ALLOC_MEM, in Fortran with pointer support. We assume 4-byte REALs, and assume that pointers are address-sized.

```
REAL A
POINTER (P, A(100,100))   ! no memory is allocated
CALL MPI_ALLOC_MEM(4*100*100, MPI_INFO_NULL, P, IERR)
! memory is allocated
...
A(3,5) = 2.71;
...
CALL MPI_FREE_MEM(A, IERR) ! memory is freed
```

Since standard Fortran does not support (C-like) pointers, this code is not Fortran 77 or Fortran 90 code. Some compilers (in particular, at the time of writing, g77 and Fortran compilers for Intel) do not support this code.

Example 1.2 Same example, in C

```
float  (* f)[100][100] ;
/* no memory is allocated */
MPI_Alloc_mem(sizeof(float)*100*100, MPI_INFO_NULL, &f);
/* memory allocated */
...
(*f)[5][3] = 2.71;
...
MPI_Free_mem(f);
```

1.3 Error Handling

An MPI implementation cannot or may choose not to handle some errors that occur during MPI calls. These can include errors that generate exceptions or traps, such as floating point errors or access violations. The set of errors that are handled by MPI is implementation-dependent. Each such error generates an **MPI exception**.

The above text takes precedence over any text on error handling within this document. Specifically, text that states that errors *will* be handled should be read as *may* be handled.

A user can associate error handlers to three types of objects: communicators, windows, and files. The specified error handling routine will be used for any MPI exception that occurs during a call to MPI for the respective object. MPI calls that are not related to any objects are considered to be attached to the communicator MPI_COMM_WORLD. The attachment of error handlers to objects is purely local: different processes may attach different error handlers to corresponding objects.

Several predefined error handlers are available in MPI:

MPI_ERRORS_ARE_FATAL The handler, when called, causes the program to abort on all executing processes. This has the same effect as if MPI_ABORT was called by the process that invoked the handler.

MPI_ERRORS_RETURN The handler has no effect other than returning the error code to the user.

Implementations may provide additional predefined error handlers and programmers can code their own error handlers.

The error handler MPI_ERRORS_ARE_FATAL is associated by default with MPI_COMM_WORLD after initialization. Thus, if the user chooses not to control error handling, every error that MPI handles is treated as fatal. Since (almost) all MPI calls return an error code, a user may choose to handle errors in its main code, by testing the return code of MPI calls and executing a suitable recovery code when the call was not successful. In this case, the error handler MPI_ERRORS_RETURN will be used. Usually it is more convenient and more efficient not to test for errors after each MPI call, and have such error handled by a non trivial MPI error handler.

After an error is detected, the state of MPI is undefined. That is, using a user-defined error handler, or MPI_ERRORS_RETURN, does *not* necessarily allow the user to continue to use MPI after an error is detected. The purpose of these error handlers is to allow a user to issue user-defined error messages and to take actions unrelated to MPI (such as flushing I/O buffers) before a program exits. An MPI implementation is free to allow MPI to continue after an error but is not required to do so.

Advice to implementors. A good quality implementation will, to the greatest possible extent, circumscribe the impact of an error, so that normal processing can continue after an error handler was invoked. The implementation documentation will provide information on the possible effect of each class of errors. (*End of advice to implementors.*)

An MPI error handler is an opaque object, which is accessed by a handle. MPI calls are provided to create new error handlers, to associate error handlers with objects, and to test which error handler is associated with an object. C and C++ have distinct typedefs for user defined error handling callback functions that accept communicator, file, and window arguments. In Fortran there are three user routines.

An error handler object is created by a call to MPI_XXX_CREATE_ERRHANDLER(function, errhandler), where XXX is, respectively, COMM, WIN, or FILE.

An error handler is attached to a communicator, window, or file by a call to MPI_XXX_SET_ERRHANDLER. The error handler must be either a predefined error handler, or an error handler that was created by a call to MPI_XXX_CREATE_ERRHANDLER, with matching XXX. The predefined error handlers MPI_ERRORS_RETURN and MPI_ERRORS_ARE_FATAL can be attached to communicators, windows, and files. In C++, the predefined error handler MPI::ERRORS_THROW_EXCEPTIONS can also be attached to communicators, windows, and files.

The error handler currently associated with a communicator, window, or file can be retrieved by a call to MPI_XXX_GET_ERRHANDLER.

The MPI function MPI_ERRHANDLER_FREE can be used to free an error handler that was created by a call to MPI_XXX_CREATE_ERRHANDLER.

MPI_{COMM,WIN,FILE}_GET_ERRHANDLER behave as if a new error handler object is created. That is, once the error handler is no longer needed, MPI_ERRHANDLER_FREE should be called with the error handler returned from MPI_ERRHANDLER_GET or MPI_{COMM,WIN,FILE}_GET_ERRHANDLER to mark the error handler for deallocation. This provides behavior similar to that of MPI_COMM_GROUP and MPI_GROUP_FREE.

Advice to implementors. High-quality implementation should raise an error when an error handler that was created by a call to MPI_XXX_CREATE_ERRHANDLER is attached to an object of the wrong type with a call to MPI_YYY_SET_ERRHANDLER. To do so, it is necessary to maintain, with each error handler, information on the typedef of the associated user function. (*End of advice to implementors.*)

The syntax for these calls is given below.

1.3.1 Error Handlers for Communicators

MPI_COMM_CREATE_ERRHANDLER(function, errhandler)

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

```
int MPI_Comm_create_errhandler(MPI_Comm_errhandler_function *function,
              MPI_Errhandler *errhandler)
```

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

{static MPI::Errhandler
    MPI::Comm::Create_errhandler(MPI::Comm::Errhandler_function*
    function) *(binding deprecated, see Section* **??***)* }

Creates an error handler that can be attached to communicators. This function is identical to MPI_ERRHANDLER_CREATE, whose use is deprecated.

The user routine should be, in C, a function of type MPI_Comm_errhandler_function, which is defined as

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

The first argument is the communicator in use. The second is the error code to be returned by the MPI routine that raised the error. If the routine would have returned MPI_ERR_IN_STATUS, it is the error code returned in the status for the request that caused the error handler to be invoked. The remaining arguments are "stdargs" arguments whose number and meaning is implementation-dependent. An implementation should clearly document these arguments. Addresses are used so that the handler may be written in Fortran. This typedef replaces MPI_Handler_function, whose use is deprecated.

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

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

In C++, the user routine should be of the form:

{typedef void MPI::Comm::Errhandler_function(MPI::Comm &, int *, ...);
    (binding deprecated, see Section **??***)*}

> *Rationale.* The variable argument list is provided because it provides an ISO-standard hook for providing additional information to the error handler; without this hook, ISO C prohibits additional arguments. (*End of rationale.*)

> *Advice to users.* A newly created communicator inherits the error handler that is associated with the "parent" communicator. In particular, the user can specify a "global" error handler for all communicators by associating this handler with the communicator MPI_COMM_WORLD immediately after initialization. (*End of advice to users.*)

MPI_COMM_SET_ERRHANDLER(comm, errhandler)

INOUT	comm	communicator (handle)
IN	errhandler	new error handler for communicator (handle)

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

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

{void MPI::Comm::Set_errhandler(const MPI::Errhandler& errhandler) *(binding deprecated, see Section ??)* }

Attaches a new error handler to a communicator. The error handler must be either a predefined error handler, or an error handler created by a call to MPI_COMM_CREATE_ERRHANDLER. This call is identical to MPI_ERRHANDLER_SET, whose use is deprecated.

MPI_COMM_GET_ERRHANDLER(comm, errhandler)

IN	comm	communicator (handle)
OUT	errhandler	error handler currently associated with communicator (handle)

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

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

{MPI::Errhandler MPI::Comm::Get_errhandler() const *(binding deprecated, see Section ??)* }

Retrieves the error handler currently associated with a communicator. This call is identical to MPI_ERRHANDLER_GET, whose use is deprecated.

Example: A library function may register at its entry point the current error handler for a communicator, set its own private error handler for this communicator, and restore before exiting the previous error handler.

1.3.2 Error Handlers for Windows

MPI_WIN_CREATE_ERRHANDLER(function, errhandler)

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

```
int MPI_Win_create_errhandler(MPI_Win_errhandler_function *function,
              MPI_Errhandler *errhandler)
```

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

{static MPI::Errhandler
MPI::Win::Create_errhandler(MPI::Win::Errhandler_function*
function) *(binding deprecated, see Section ??)* }

Creates an error handler that can be attached to a window object. The user routine should be, in C, a function of type MPI_Win_errhandler_function which is defined as

```
typedef void MPI_Win_errhandler_function(MPI_Win *, int *, ...);
```

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

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

```
SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)
    INTEGER WIN, ERROR_CODE
```

In C++, the user routine should be of the form:

{`typedef void MPI::Win::Errhandler_function(MPI::Win &, int *, ...);` *(binding deprecated, see Section* **??***)*}

MPI_WIN_SET_ERRHANDLER(win, errhandler)

INOUT	win	window (handle)
IN	errhandler	new error handler for window (handle)

```
int MPI_Win_set_errhandler(MPI_Win win, MPI_Errhandler errhandler)
```

```
MPI_WIN_SET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
    INTEGER WIN, ERRHANDLER, IERROR
```

{`void MPI::Win::Set_errhandler(const MPI::Errhandler& errhandler)` *(binding deprecated, see Section* **??***)* }

Attaches a new error handler to a window. The error handler must be either a predefined error handler, or an error handler created by a call to MPI_WIN_CREATE_ERRHANDLER.

MPI_WIN_GET_ERRHANDLER(win, errhandler)

IN	win	window (handle)
OUT	errhandler	error handler currently associated with window (handle)

```
int MPI_Win_get_errhandler(MPI_Win win, MPI_Errhandler *errhandler)
```

```
MPI_WIN_GET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
    INTEGER WIN, ERRHANDLER, IERROR
```

{`MPI::Errhandler MPI::Win::Get_errhandler() const` *(binding deprecated, see Section* **??***)* }

Retrieves the error handler currently associated with a window.

1.3.3 Error Handlers for Files

MPI_FILE_CREATE_ERRHANDLER(function, errhandler)

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

```
int MPI_File_create_errhandler(MPI_File_errhandler_function *function,
              MPI_Errhandler *errhandler)
```

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

{static MPI::Errhandler
MPI::File::Create_errhandler(MPI::File::Errhandler_function*
function) *(binding deprecated, see Section* ??*)* }

Creates an error handler that can be attached to a file object. The user routine should be, in C, a function of type MPI_File_errhandler_function, which is defined as

```
typedef void MPI_File_errhandler_function(MPI_File *, int *, ...);
```

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

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

```
SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE)
    INTEGER FILE, ERROR_CODE
```

In C++, the user routine should be of the form:

{typedef void MPI::File::Errhandler_function(MPI::File &, int *, ...);
(binding deprecated, see Section ??*)*}

MPI_FILE_SET_ERRHANDLER(file, errhandler)

INOUT	file	file (handle)
IN	errhandler	new error handler for file (handle)

```
int MPI_File_set_errhandler(MPI_File file, MPI_Errhandler errhandler)
```

```
MPI_FILE_SET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
    INTEGER FILE, ERRHANDLER, IERROR
```

{void MPI::File::Set_errhandler(const MPI::Errhandler& errhandler) *(binding deprecated, see Section* ??*)* }

Attaches a new error handler to a file. The error handler must be either a predefined error handler, or an error handler created by a call to MPI_FILE_CREATE_ERRHANDLER.

MPI_FILE_GET_ERRHANDLER(file, errhandler)

IN	file	file (handle)
OUT	errhandler	error handler currently associated with file (handle)

```
int MPI_File_get_errhandler(MPI_File file, MPI_Errhandler *errhandler)
```

```
MPI_FILE_GET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
    INTEGER FILE, ERRHANDLER, IERROR
```

{`MPI::Errhandler MPI::File::Get_errhandler() const`*(binding deprecated, see Section ??)* }

Retrieves the error handler currently associated with a file.

1.3.4 Freeing Errorhandlers and Retrieving Error Strings

MPI_ERRHANDLER_FREE(errhandler)

INOUT	errhandler	MPI error handler (handle)

```
int MPI_Errhandler_free(MPI_Errhandler *errhandler)
```

```
MPI_ERRHANDLER_FREE(ERRHANDLER, IERROR)
    INTEGER ERRHANDLER, IERROR
```

{`void MPI::Errhandler::Free()`*(binding deprecated, see Section ??)* }

Marks the error handler associated with errhandler for deallocation and sets errhandler to MPI_ERRHANDLER_NULL. The error handler will be deallocated after all the objects associated with it (communicator, window, or file) have been deallocated.

MPI_ERROR_STRING(errorcode, string, resultlen)

IN	errorcode	Error code returned by an MPI routine
OUT	string	Text that corresponds to the errorcode
OUT	resultlen	Length (in printable characters) of the result returned in string

```
int MPI_Error_string(int errorcode, char *string, int *resultlen)
```

```
MPI_ERROR_STRING(ERRORCODE, STRING, RESULTLEN, IERROR)
    INTEGER ERRORCODE, RESULTLEN, IERROR
    CHARACTER*(*) STRING
```

{`void MPI::Get_error_string(int errorcode, char* name, int& resultlen)`*(binding deprecated, see Section ??)* }

Returns the error string associated with an error code or class. The argument string must represent storage that is at least MPI_MAX_ERROR_STRING characters long.

The number of characters actually written is returned in the output argument, resultlen.

Rationale. The form of this function was chosen to make the Fortran and C bindings similar. A version that returns a pointer to a string has two difficulties. First, the return string must be statically allocated and different for each error message (allowing the pointers returned by successive calls to MPI_ERROR_STRING to point to the correct message). Second, in Fortran, a function declared as returning CHARACTER*(*) can not be referenced in, for example, a PRINT statement. (*End of rationale.*)

1.4 Error Codes and Classes

The error codes returned by MPI are left entirely to the implementation (with the exception of MPI_SUCCESS). This is done to allow an implementation to provide as much information as possible in the error code (for use with MPI_ERROR_STRING).

To make it possible for an application to interpret an error code, the routine MPI_ERROR_CLASS converts any error code into one of a small set of standard error codes, called *error classes*. Valid error classes are shown in Table 1.1 and Table 1.2.

The error classes are a subset of the error codes: an MPI function may return an error class number; and the function MPI_ERROR_STRING can be used to compute the error string associated with an error class. An MPI error class is a valid MPI error code. Specifically, the values defined for MPI error classes are valid MPI error codes.

The error codes satisfy,

$$0 = \text{MPI\_SUCCESS} < \text{MPI\_ERR\_...} \leq \text{MPI\_ERR\_LASTCODE}.$$

Rationale. The difference between MPI_ERR_UNKNOWN and MPI_ERR_OTHER is that MPI_ERROR_STRING can return useful information about MPI_ERR_OTHER.

Note that MPI_SUCCESS = 0 is necessary to be consistent with C practice; the separation of error classes and error codes allows us to define the error classes this way. Having a known LASTCODE is often a nice sanity check as well. (*End of rationale.*)

MPI_ERROR_CLASS(errorcode, errorclass)

IN	errorcode	Error code returned by an MPI routine
OUT	errorclass	Error class associated with errorcode

```
int MPI_Error_class(int errorcode, int *errorclass)
```

```
MPI_ERROR_CLASS(ERRORCODE, ERRORCLASS, IERROR)
    INTEGER ERRORCODE, ERRORCLASS, IERROR
```

{`int MPI::Get_error_class(int errorcode)` *(binding deprecated, see Section ??)* }

The function MPI_ERROR_CLASS maps each standard error code (error class) onto itself.

MPI_SUCCESS	No error
MPI_ERR_BUFFER	Invalid buffer pointer
MPI_ERR_COUNT	Invalid count argument
MPI_ERR_TYPE	Invalid datatype argument
MPI_ERR_TAG	Invalid tag argument
MPI_ERR_COMM	Invalid communicator
MPI_ERR_RANK	Invalid rank
MPI_ERR_REQUEST	Invalid request (handle)
MPI_ERR_ROOT	Invalid root
MPI_ERR_GROUP	Invalid group
MPI_ERR_OP	Invalid operation
MPI_ERR_TOPOLOGY	Invalid topology
MPI_ERR_DIMS	Invalid dimension argument
MPI_ERR_ARG	Invalid argument of some other kind
MPI_ERR_UNKNOWN	Unknown error
MPI_ERR_TRUNCATE	Message truncated on receive
MPI_ERR_OTHER	Known error not in this list
MPI_ERR_INTERN	Internal MPI (implementation) error
MPI_ERR_IN_STATUS	Error code is in status
MPI_ERR_PENDING	Pending request
MPI_ERR_KEYVAL	Invalid keyval has been passed
MPI_ERR_NO_MEM	MPI_ALLOC_MEM failed because memory is exhausted
MPI_ERR_BASE	Invalid base passed to MPI_FREE_MEM
MPI_ERR_INFO_KEY	Key longer than MPI_MAX_INFO_KEY
MPI_ERR_INFO_VALUE	Value longer than MPI_MAX_INFO_VAL
MPI_ERR_INFO_NOKEY	Invalid key passed to MPI_INFO_DELETE
MPI_ERR_SPAWN	Error in spawning processes
MPI_ERR_PORT	Invalid port name passed to MPI_COMM_CONNECT
MPI_ERR_SERVICE	Invalid service name passed to MPI_UNPUBLISH_NAME
MPI_ERR_NAME	Invalid service name passed to MPI_LOOKUP_NAME
MPI_ERR_WIN	Invalid win argument
MPI_ERR_SIZE	Invalid size argument
MPI_ERR_DISP	Invalid disp argument
MPI_ERR_INFO	Invalid info argument
MPI_ERR_LOCKTYPE	Invalid locktype argument
MPI_ERR_ASSERT	Invalid assert argument
MPI_ERR_RMA_CONFLICT	Conflicting accesses to window
MPI_ERR_RMA_SYNC	Wrong synchronization of RMA calls

Table 1.1: Error classes (Part 1)

MPI_ERR_FILE	Invalid file handle
MPI_ERR_NOT_SAME	Collective argument not identical on all processes, or collective routines called in a different order by different processes
MPI_ERR_AMODE	Error related to the amode passed to MPI_FILE_OPEN
MPI_ERR_UNSUPPORTED_DATAREP	Unsupported datarep passed to MPI_FILE_SET_VIEW
MPI_ERR_UNSUPPORTED_OPERATION	Unsupported operation, such as seeking on a file which supports sequential access only
MPI_ERR_NO_SUCH_FILE	File does not exist
MPI_ERR_FILE_EXISTS	File exists
MPI_ERR_BAD_FILE	Invalid file name (e.g., path name too long)
MPI_ERR_ACCESS	Permission denied
MPI_ERR_NO_SPACE	Not enough space
MPI_ERR_QUOTA	Quota exceeded
MPI_ERR_READ_ONLY	Read-only file or file system
MPI_ERR_FILE_IN_USE	File operation could not be completed, as the file is currently open by some process
MPI_ERR_DUP_DATAREP	Conversion functions could not be registered because a data representation identifier that was already defined was passed to MPI_REGISTER_DATAREP
MPI_ERR_CONVERSION	An error occurred in a user supplied data conversion function.
MPI_ERR_IO	Other I/O error
MPI_ERR_LASTCODE	Last error code

Table 1.2: Error classes (Part 2)

1.5 Error Classes, Error Codes, and Error Handlers

Users may want to write a layered library on top of an existing MPI implementation, and this library may have its own set of error codes and classes. An example of such a library is an I/O library based on MPI, see Chapter ?? on page ??. For this purpose, functions are needed to:

1. add a new error class to the ones an MPI implementation already knows.
2. associate error codes with this error class, so that MPI_ERROR_CLASS works.
3. associate strings with these error codes, so that MPI_ERROR_STRING works.
4. invoke the error handler associated with a communicator, window, or object.

Several functions are provided to do this. They are all local. No functions are provided to free error classes or codes: it is not expected that an application will generate them in significant numbers.

MPI_ADD_ERROR_CLASS(errorclass)

OUT	errorclass	value for the new error class (integer)

```
int MPI_Add_error_class(int *errorclass)
```

```
MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR)
    INTEGER ERRORCLASS, IERROR
```

{`int MPI::Add_error_class()` *(binding deprecated, see Section* **??***)* }

Creates a new error class and returns the value for it.

> *Rationale.* To avoid conflicts with existing error codes and classes, the value is set by the implementation and not by the user. (*End of rationale.*)

> *Advice to implementors.* A high-quality implementation will return the value for a new errorclass in the same deterministic way on all processes. (*End of advice to implementors.*)

> *Advice to users.* Since a call to MPI_ADD_ERROR_CLASS is local, the same errorclass may not be returned on all processes that make this call. Thus, it is not safe to assume that registering a new error on a set of processes at the same time will yield the same errorclass on all of the processes. However, if an implementation returns the new errorclass in a deterministic way, and they are always generated in the same order on the same set of processes (for example, all processes), then the value will be the same. However, even if a deterministic algorithm is used, the value can vary across processes. This can happen, for example, if different but overlapping groups of processes make a series of calls. As a result of these issues, getting the "same" error on multiple processes may not cause the same value of error code to be generated. (*End of advice to users.*)

The value of MPI_ERR_LASTCODE is a constant value and is not affected by new user-defined error codes and classes. Instead, a predefined attribute key MPI_LASTUSEDCODE is associated with MPI_COMM_WORLD. The attribute value corresponding to this key is the current maximum error class including the user-defined ones. This is a local value and may be different on different processes. The value returned by this key is always greater than or equal to MPI_ERR_LASTCODE.

> *Advice to users.* The value returned by the key MPI_LASTUSEDCODE will not change unless the user calls a function to explicitly add an error class/code. In a multi-threaded environment, the user must take extra care in assuming this value has not changed. Note that error codes and error classes are not necessarily dense. A user may not assume that each error class below MPI_LASTUSEDCODE is valid. (*End of advice to users.*)

MPI_ADD_ERROR_CODE(errorclass, errorcode)

IN	errorclass	error class (integer)
OUT	errorcode	new error code to associated with errorclass (integer)

```
int MPI_Add_error_code(int errorclass, int *errorcode)

MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR)
    INTEGER ERRORCLASS, ERRORCODE, IERROR
```

{int MPI::Add_error_code(int errorclass) *(binding deprecated, see Section ??)* }

Creates new error code associated with errorclass and returns its value in errorcode.

Rationale. To avoid conflicts with existing error codes and classes, the value of the new error code is set by the implementation and not by the user. (*End of rationale.*)

Advice to implementors. A high-quality implementation will return the value for a new errorcode in the same deterministic way on all processes. (*End of advice to implementors.*)

MPI_ADD_ERROR_STRING(errorcode, string)

IN	errorcode	error code or class (integer)
IN	string	text corresponding to errorcode (string)

```
int MPI_Add_error_string(int errorcode, char *string)

MPI_ADD_ERROR_STRING(ERRORCODE, STRING, IERROR)
    INTEGER ERRORCODE, IERROR
    CHARACTER*(*) STRING
```

{void MPI::Add_error_string(int errorcode, const char* string) *(binding deprecated, see Section ??)* }

Associates an error string with an error code or class. The string must be no more than MPI_MAX_ERROR_STRING characters long. The length of the string is as defined in the calling language. The length of the string does not include the null terminator in C or C++. Trailing blanks will be stripped in Fortran. Calling MPI_ADD_ERROR_STRING for an errorcode that already has a string will replace the old string with the new string. It is erroneous to call MPI_ADD_ERROR_STRING for an error code or class with a value $\leq$ MPI_ERR_LASTCODE.

If MPI_ERROR_STRING is called when no string has been set, it will return a empty string (all spaces in Fortran, "" in C and C++).

Section 1.3 on page 7 describes the methods for creating and associating error handlers with communicators, files, and windows.

MPI_COMM_CALL_ERRHANDLER (comm, errorcode)

IN	comm	communicator with error handler (handle)
IN	errorcode	error code (integer)

```
int MPI_Comm_call_errhandler(MPI_Comm comm, int errorcode)

MPI_COMM_CALL_ERRHANDLER(COMM, ERRORCODE, IERROR)
    INTEGER COMM, ERRORCODE, IERROR
```

{void MPI::Comm::Call_errhandler(int errorcode) const *(binding deprecated, see Section ??)* }

This function invokes the error handler assigned to the communicator with the error code supplied. This function returns MPI_SUCCESS in C and C++ and the same value in IERROR if the error handler was successfully called (assuming the process is not aborted and the error handler returns).

Advice to users. Users should note that the default error handler is MPI_ERRORS_ARE_FATAL. Thus, calling MPI_COMM_CALL_ERRHANDLER will abort the comm processes if the default error handler has not been changed for this communicator or on the parent before the communicator was created. (*End of advice to users.*)

MPI_WIN_CALL_ERRHANDLER (win, errorcode)

IN	win	window with error handler (handle)
IN	errorcode	error code (integer)

```
int MPI_Win_call_errhandler(MPI_Win win, int errorcode)

MPI_WIN_CALL_ERRHANDLER(WIN, ERRORCODE, IERROR)
    INTEGER WIN, ERRORCODE, IERROR
```

{void MPI::Win::Call_errhandler(int errorcode) const *(binding deprecated, see Section ??)* }

This function invokes the error handler assigned to the window with the error code supplied. This function returns MPI_SUCCESS in C and C++ and the same value in IERROR if the error handler was successfully called (assuming the process is not aborted and the error handler returns).

Advice to users. As with communicators, the default error handler for windows is MPI_ERRORS_ARE_FATAL. (*End of advice to users.*)

MPI_FILE_CALL_ERRHANDLER (fh, errorcode)

IN	fh	file with error handler (handle)
IN	errorcode	error code (integer)

```
int MPI_File_call_errhandler(MPI_File fh, int errorcode)

MPI_FILE_CALL_ERRHANDLER(FH, ERRORCODE, IERROR)
    INTEGER FH, ERRORCODE, IERROR
```

{`void MPI::File::Call_errhandler(int errorcode) const`*(binding deprecated, see Section ??)* }

This function invokes the error handler assigned to the file with the error code supplied. This function returns MPI_SUCCESS in C and C++ and the same value in IERROR if the error handler was successfully called (assuming the process is not aborted and the error handler returns).

> *Advice to users.* Unlike errors on communicators and windows, the default behavior for files is to have MPI_ERRORS_RETURN. (*End of advice to users.*)

> *Advice to users.* Users are warned that handlers should not be called recursively with MPI_COMM_CALL_ERRHANDLER, MPI_FILE_CALL_ERRHANDLER, or MPI_WIN_CALL_ERRHANDLER. Doing this can create a situation where an infinite recursion is created. This can occur if MPI_COMM_CALL_ERRHANDLER, MPI_FILE_CALL_ERRHANDLER, or MPI_WIN_CALL_ERRHANDLER is called inside an error handler.
>
> Error codes and classes are associated with a process. As a result, they may be used in any error handler. Error handlers should be prepared to deal with any error code they are given. Furthermore, it is good practice to only call an error handler with the appropriate error codes. For example, file errors would normally be sent to the file error handler. (*End of advice to users.*)

1.6 Timers and Synchronization

MPI defines a timer. A timer is specified even though it is not "message-passing," because timing parallel programs is important in "performance debugging" and because existing timers (both in POSIX 1003.1-1988 and 1003.4D 14.1 and in Fortran 90) are either inconvenient or do not provide adequate access to high-resolution timers. See also Section **??** on page **??**.

MPI_WTIME()

```
double MPI_Wtime(void)

DOUBLE PRECISION MPI_WTIME()
```

{`double MPI::Wtime()`*(binding deprecated, see Section ??)* }

MPI_WTIME returns a floating-point number of seconds, representing elapsed wall-clock time since some time in the past.

The "time in the past" is guaranteed not to change during the life of the process. The user is responsible for converting large numbers of seconds to other units if they are preferred.

This function is portable (it returns seconds, not "ticks"), it allows high-resolution, and carries no unnecessary baggage. One would use it like this:

```
{
   double starttime, endtime;
   starttime = MPI_Wtime();
    ....  stuff to be timed  ...
   endtime   = MPI_Wtime();
   printf("That took %f seconds\n",endtime-starttime);
}
```

The times returned are local to the node that called them. There is no requirement that different nodes return "the same time." (But see also the discussion of MPI_WTIME_IS_GLOBAL).

MPI_WTICK()

`double MPI_Wtick(void)`

`DOUBLE PRECISION MPI_WTICK()`

{`double MPI::Wtick()` *(binding deprecated, see Section* **??***)* }

MPI_WTICK returns the resolution of MPI_WTIME in seconds. That is, it returns, as a double precision value, the number of seconds between successive clock ticks. For example, if the clock is implemented by the hardware as a counter that is incremented every millisecond, the value returned by MPI_WTICK should be 10^{-3}.

1.7 Startup

One goal of MPI is to achieve *source code portability*. By this we mean that a program written using MPI and complying with the relevant language standards is portable as written, and must not require any source code changes when moved from one system to another. This explicitly does *not* say anything about how an MPI program is started or launched from the command line, nor what the user must do to set up the environment in which an MPI program will run. However, an implementation may require some setup to be performed before other MPI routines may be called. To provide for this, MPI includes an initialization routine MPI_INIT.

MPI_INIT()

`int MPI_Init(int *argc, char ***argv)`

```
MPI_INIT(IERROR)
    INTEGER IERROR
```

{`void MPI::Init(int& argc, char**& argv)` *(binding deprecated, see Section* **??***)* }

{`void MPI::Init()` *(binding deprecated, see Section* **??***)* }

All MPI programs must contain exactly one call to an MPI initialization routine: MPI_INIT or MPI_INIT_THREAD. Subsequent calls to any initialization routines are erroneous. The only MPI functions that may be invoked before the MPI initialization routines are called are MPI_GET_VERSION, []MPI_GET_LIBRARY_VERSION, MPI_INITIALIZED, and MPI_FINALIZED. The version for ISO C accepts the argc and argv that are provided by the arguments to main or NULL:

ticket204.

```
int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);

    /* parse arguments */
    /* main program    */

    MPI_Finalize();     /* see below */
}
```

The Fortran version takes only IERROR.

Conforming implementations of MPI are required to allow applications to pass NULL for both the argc and argv arguments of main in C and C++. In C++, there is an alternative binding for MPI::Init that does not have these arguments at all.

Rationale. In some applications, libraries may be making the call to MPI_Init, and may not have access to argc and argv from main. It is anticipated that applications requiring special information about the environment or information supplied by mpiexec can get that information from environment variables. (*End of rationale.*)

MPI_FINALIZE()

```
int MPI_Finalize(void)
```

```
MPI_FINALIZE(IERROR)
    INTEGER IERROR
```

{void MPI::Finalize() *(binding deprecated, see Section ??)* }

This routine cleans up all MPI state. Each process must call MPI_FINALIZE before it exits. Unless there has been a call to MPI_ABORT, each process must ensure that all pending nonblocking communications are (locally) complete before calling MPI_FINALIZE. Further, at the instant at which the last process calls MPI_FINALIZE, all pending sends must be matched by a receive, and all pending receives must be matched by a send.

For example, the following program is correct:

```
        Process 0                Process 1
        ---------                ---------
        MPI_Init();              MPI_Init();
        MPI_Send(dest=1);        MPI_Recv(src=0);
        MPI_Finalize();          MPI_Finalize();
```

Without the matching receive, the program is erroneous:

```
Process 0               Process 1
-----------             -----------
MPI_Init();             MPI_Init();
MPI_Send (dest=1);
MPI_Finalize();         MPI_Finalize();
```

A successful return from a blocking communication operation or from MPI_WAIT or MPI_TEST tells the user that the buffer can be reused and means that the communication is completed by the user, but does not guarantee that the local process has no more work to do. A successful return from MPI_REQUEST_FREE with a request handle generated by an MPI_ISEND nullifies the handle but provides no assurance of operation completion. The MPI_ISEND is complete only when it is known by some means that a matching receive has completed. MPI_FINALIZE guarantees that all local actions required by communications the user has completed will, in fact, occur before it returns.

MPI_FINALIZE guarantees nothing about pending communications that have not been completed (completion is assured only by MPI_WAIT, MPI_TEST, or MPI_REQUEST_FREE combined with some other verification of completion).

Example 1.3 This program is correct:

```
rank 0                          rank 1
=====================================================
...                             ...
MPI_Isend();                    MPI_Recv();
MPI_Request_free();             MPI_Barrier();
MPI_Barrier();                  MPI_Finalize();
MPI_Finalize();                 exit();
exit();
```

Example 1.4 This program is erroneous and its behavior is undefined:

```
rank 0                          rank 1
=====================================================
...                             ...
MPI_Isend();                    MPI_Recv();
MPI_Request_free();             MPI_Finalize();
MPI_Finalize();                 exit();
exit();
```

If no MPI_BUFFER_DETACH occurs between an MPI_BSEND (or other buffered send) and MPI_FINALIZE, the MPI_FINALIZE implicitly supplies the MPI_BUFFER_DETACH.

Example 1.5 This program is correct, and after the MPI_Finalize, it is as if the buffer had been detached.

```
rank 0                          rank 1
=====================================================
...                             ...
buffer = malloc(1000000);       MPI_Recv();
MPI_Buffer_attach();            MPI_Finalize();
MPI_Bsend();                    exit();
MPI_Finalize();
free(buffer);
exit();
```

Example 1.6 In this example, MPI_Iprobe() must return a FALSE flag. MPI_Test_cancelled() must return a TRUE flag, independent of the relative order of execution of MPI_Cancel() in process 0 and MPI_Finalize() in process 1.

The MPI_Iprobe() call is there to make sure the implementation knows that the "tag1" message exists at the destination, without being able to claim that the user knows about it.

```
rank 0                          rank 1
========================================================
MPI_Init();                     MPI_Init();
MPI_Isend(tag1);
MPI_Barrier();                  MPI_Barrier();
                                MPI_Iprobe(tag2);
MPI_Barrier();                  MPI_Barrier();
                                MPI_Finalize();
                                exit();
MPI_Cancel();
MPI_Wait();
MPI_Test_cancelled();
MPI_Finalize();
exit();
```

Advice to implementors. An implementation may need to delay the return from MPI_FINALIZE until all potential future message cancellations have been processed. One possible solution is to place a barrier inside MPI_FINALIZE (*End of advice to implementors.*)

ticket204. Once MPI_FINALIZE returns, no MPI routine (not even MPI_INIT) may be called, except for MPI_GET_VERSION, MPI_GET_LIBRARY_VERSION, MPI_INITIALIZED, and MPI_FINALIZED. Each process must complete any pending communication it initiated before it calls MPI_FINALIZE. If the call returns, each process may continue local computations, or exit, without participating in further MPI communication with other processes. MPI_FINALIZE is collective over all connected processes. If no processes were spawned, accepted or connected then this means over MPI_COMM_WORLD; otherwise it is collective over the union of all processes that have been and continue to be connected, as explained in Section ?? on page ??.

Advice to implementors. Even though a process has completed all the communication it initiated, such communication may not yet be completed from the viewpoint of the underlying MPI system. E.g., a blocking send may have completed, even though the data is still buffered at the sender. The MPI implementation must ensure that a process has completed any involvement in MPI communication before MPI_FINALIZE returns. Thus, if a process exits after the call to MPI_FINALIZE, this will not cause an ongoing communication to fail. (*End of advice to implementors.*)

Although it is not required that all processes return from MPI_FINALIZE, it is required that at least process 0 in MPI_COMM_WORLD return, so that users can know that the MPI portion of the computation is over. In addition, in a POSIX environment, they may desire to supply an exit code for each process that returns from MPI_FINALIZE.

Example 1.7 The following illustrates the use of requiring that at least one process return and that it be known that process 0 is one of the processes that return. One wants code like the following to work no matter how many processes return.

```
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
...
MPI_Finalize();
if (myrank == 0) {
    resultfile = fopen("outfile","w");
    dump_results(resultfile);
    fclose(resultfile);
}
exit(0);
```

MPI_INITIALIZED(flag)

OUT	flag	Flag is true if MPI_INIT has been called and false otherwise.

```
int MPI_Initialized(int *flag)
```

```
MPI_INITIALIZED(FLAG, IERROR)
    LOGICAL FLAG
    INTEGER IERROR
```

{`bool MPI::Is_initialized()` *(binding deprecated, see Section* **??***)* }

This routine may be used to determine whether MPI_INIT has been called. MPI_INITIALIZED returns true if the calling process has called MPI_INIT. Whether MPI_FINALIZE has been called does not affect the behavior of MPI_INITIALIZED. It is one of the few routines that may be called before MPI_INIT is called.

MPI_ABORT(comm, errorcode)

IN	comm	communicator of tasks to abort
IN	errorcode	error code to return to invoking environment

```
int MPI_Abort(MPI_Comm comm, int errorcode)
```

```
MPI_ABORT(COMM, ERRORCODE, IERROR)
    INTEGER COMM, ERRORCODE, IERROR
```

{`void MPI::Comm::Abort(int errorcode)` *(binding deprecated, see Section* **??***)* }

This routine makes a "best attempt" to abort all tasks in the group of comm. This function does not require that the invoking environment take any action with the error code. However, a Unix or POSIX environment should handle this as a `return errorcode` from the main program.

It may not be possible for an MPI implementation to abort only the processes represented by comm if this is a subset of the processes. In this case, the MPI implementation should attempt to abort all the connected processes but should not abort any unconnected processes. If no processes were spawned, accepted or connected then this has the effect of aborting all the processes associated with MPI_COMM_WORLD.

Rationale. The communicator argument is provided to allow for future extensions of MPI to environments with, for example, dynamic process management. In particular, it allows but does not require an MPI implementation to abort a subset of MPI_COMM_WORLD. (*End of rationale.*)

Advice to users. Whether the errorcode is returned from the executable or from the MPI process startup mechanism (e.g., `mpiexec`), is an aspect of quality of the MPI library but not mandatory. (*End of advice to users.*)

Advice to implementors. Where possible, a high-quality implementation will try to return the errorcode from the MPI process startup mechanism (e.g. `mpiexec` or singleton init). (*End of advice to implementors.*)

1.7.1 Allowing User Functions at Process Termination

There are times in which it would be convenient to have actions happen when an MPI process finishes. For example, a routine may do initializations that are useful until the MPI job (or that part of the job that being terminated in the case of dynamically created processes) is finished. This can be accomplished in MPI by attaching an attribute to MPI_COMM_SELF with a callback function. When MPI_FINALIZE is called, it will first execute the equivalent of an MPI_COMM_FREE on MPI_COMM_SELF. This will cause the delete callback function to be executed on all keys associated with MPI_COMM_SELF, in the reverse order that they were set on MPI_COMM_SELF. If no key has been attached to MPI_COMM_SELF, then no callback is invoked. The "freeing" of MPI_COMM_SELF occurs before any other parts of MPI are affected. Thus, for example, calling MPI_FINALIZED will return false in any of these callback functions. Once done with MPI_COMM_SELF, the order and rest of the actions taken by MPI_FINALIZE is not specified.

Advice to implementors. Since attributes can be added from any supported language, the MPI implementation needs to remember the creating language so the correct callback is made. Implementations that use the attribute delete callback on MPI_COMM_SELF internally should register their internal callbacks before returning from MPI_INIT / MPI_INIT_THREAD, so that libraries or applications will not have portions of the MPI implementation shut down before the application-level callbacks are made. (*End of advice to implementors.*)

1.7.2 Determining Whether MPI Has Finished

One of the goals of MPI was to allow for layered libraries. In order for a library to do this cleanly, it needs to know if MPI is active. In MPI the function MPI_INITIALIZED was provided to tell if MPI had been initialized. The problem arises in knowing if MPI has been finalized. Once MPI has been finalized it is no longer active and cannot be restarted. A library needs to be able to determine this to act accordingly. To achieve this the following function is needed:

MPI_FINALIZED(flag)

OUT	flag	true if MPI was finalized (logical)

```
int MPI_Finalized(int *flag)
```

```
MPI_FINALIZED(FLAG, IERROR)
    LOGICAL FLAG
    INTEGER IERROR
```

{`bool MPI::Is_finalized()` *(binding deprecated, see Section* **??***)* }

This routine returns true if MPI_FINALIZE has completed. It is legal to call MPI_FINALIZED before MPI_INIT and after MPI_FINALIZE.

Advice to users. MPI is "active" and it is thus safe to call MPI functions if MPI_INIT *has* completed and MPI_FINALIZE *has not* completed. If a library has no other way of knowing whether MPI is active or not, then it can use MPI_INITIALIZED and MPI_FINALIZED to determine this. For example, MPI is "active" in callback functions that are invoked during MPI_FINALIZE. (*End of advice to users.*)

1.8 Portable MPI Process Startup

A number of implementations of MPI provide a startup command for MPI programs that is of the form

```
mpirun <mpirun arguments> <program> <program arguments>
```

Separating the command to start the program from the program itself provides flexibility, particularly for network and heterogeneous implementations. For example, the startup script need not run on one of the machines that will be executing the MPI program itself.

Having a standard startup mechanism also extends the portability of MPI programs one step further, to the command lines and scripts that manage them. For example, a validation suite script that runs hundreds of programs can be a portable script if it is written using such a standard starup mechanism. In order that the "standard" command not be confused with existing practice, which is not standard and not portable among implementations, instead of `mpirun` MPI specifies `mpiexec`.

While a standardized startup mechanism improves the usability of MPI, the range of environments is so diverse (e.g., there may not even be a command line interface) that MPI cannot mandate such a mechanism. Instead, MPI specifies an `mpiexec` startup command and recommends but does not require it, as advice to implementors. However, if an implementation does provide a command called `mpiexec`, it must be of the form described below.

It is suggested that

```
mpiexec -n <numprocs> <program>
```

be at least one way to start `<program>` with an initial MPI_COMM_WORLD whose group contains `<numprocs>` processes. Other arguments to `mpiexec` may be implementation-dependent.

Advice to implementors. Implementors, if they do provide a special startup command for MPI programs, are advised to give it the following form. The syntax is chosen in order that `mpiexec` be able to be viewed as a command-line version of MPI_COMM_SPAWN (See Section ??).

Analogous to MPI_COMM_SPAWN, we have

```
mpiexec -n    <maxprocs>
       -soft  <        >
       -host  <        >
       -arch  <        >
       -wdir  <        >
       -path  <        >
       -file  <        >
        ...
       <command line>
```

for the case where a single command line for the application program and its arguments will suffice. See Section ?? for the meanings of these arguments. For the case corresponding to MPI_COMM_SPAWN_MULTIPLE there are two possible formats:

Form A:

```
mpiexec { <above arguments> } : { ... } : { ... } : ... : { ... }
```

As with MPI_COMM_SPAWN, all the arguments are optional. (Even the `-n x` argument is optional; the default is implementation dependent. It might be `1`, it might be taken from an environment variable, or it might be specified at compile time.) The names and meanings of the arguments are taken from the keys in the `info` argument to MPI_COMM_SPAWN. There may be other, implementation-dependent arguments as well.

Note that Form A, though convenient to type, prevents colons from being program arguments. Therefore an alternate, file-based form is allowed:

Form B:

```
mpiexec -configfile <filename>
```

where the lines of <`filename`> are of the form separated by the colons in Form A. Lines beginning with '#' are comments, and lines may be continued by terminating the partial line with '\'.

Example 1.8 Start 16 instances of `myprog` on the current or default machine:

```
mpiexec -n 16 myprog
```

Example 1.9 Start 10 processes on the machine called `ferrari`:

```
mpiexec -n 10 -host ferrari myprog
```

Example 1.10 Start three copies of the same program with different command-line arguments:

```
mpiexec myprog infile1 : myprog infile2 : myprog infile3
```

Example 1.11 Start the `ocean` program on five Suns and the `atmos` program on 10 RS/6000's:

```
mpiexec -n 5 -arch sun ocean : -n 10 -arch rs6000 atmos
```

It is assumed that the implementation in this case has a method for choosing hosts of the appropriate type. Their ranks are in the order specified.

Example 1.12 Start the `ocean` program on five Suns and the `atmos` program on 10 RS/6000's (Form B):

```
mpiexec -configfile myfile
```

where `myfile` contains

```
-n 5  -arch sun    ocean
-n 10 -arch rs6000 atmos
```

(*End of advice to implementors.*)

Chapter 2

Language Bindings Summary

In this section we summarize the specific bindings for C, Fortran, and C++. First we present the constants, type definitions, info values and keys. Then we present the routine prototypes separately for each binding. Listings are alphabetical within chapter.

2.1 Defined Values and Handles

2.1.1 Defined Constants

The C and Fortran name is listed in the left column and the C++ name is listed in the middle or right column. Constants with the type `const int` may also be implemented as literal integer constants substituted by the preprocessor.

Return Codes

C type: `const int` (or unnamed `enum`) Fortran type: `INTEGER`	C++ type: `const int` (or unnamed `enum`)
MPI_SUCCESS	MPI::SUCCESS
MPI_ERR_BUFFER	MPI::ERR_BUFFER
MPI_ERR_COUNT	MPI::ERR_COUNT
MPI_ERR_TYPE	MPI::ERR_TYPE
MPI_ERR_TAG	MPI::ERR_TAG
MPI_ERR_COMM	MPI::ERR_COMM
MPI_ERR_RANK	MPI::ERR_RANK
MPI_ERR_REQUEST	MPI::ERR_REQUEST
MPI_ERR_ROOT	MPI::ERR_ROOT
MPI_ERR_GROUP	MPI::ERR_GROUP
MPI_ERR_OP	MPI::ERR_OP
MPI_ERR_TOPOLOGY	MPI::ERR_TOPOLOGY
MPI_ERR_DIMS	MPI::ERR_DIMS
MPI_ERR_ARG	MPI::ERR_ARG
MPI_ERR_UNKNOWN	MPI::ERR_UNKNOWN
MPI_ERR_TRUNCATE	MPI::ERR_TRUNCATE
MPI_ERR_OTHER	MPI::ERR_OTHER
MPI_ERR_INTERN	MPI::ERR_INTERN
MPI_ERR_PENDING	MPI::ERR_PENDING

(Continued on next page)

Return Codes (continued)

MPI_ERR_IN_STATUS	MPI::ERR_IN_STATUS
MPI_ERR_ACCESS	MPI::ERR_ACCESS
MPI_ERR_AMODE	MPI::ERR_AMODE
MPI_ERR_ASSERT	MPI::ERR_ASSERT
MPI_ERR_BAD_FILE	MPI::ERR_BAD_FILE
MPI_ERR_BASE	MPI::ERR_BASE
MPI_ERR_CONVERSION	MPI::ERR_CONVERSION
MPI_ERR_DISP	MPI::ERR_DISP
MPI_ERR_DUP_DATAREP	MPI::ERR_DUP_DATAREP
MPI_ERR_FILE_EXISTS	MPI::ERR_FILE_EXISTS
MPI_ERR_FILE_IN_USE	MPI::ERR_FILE_IN_USE
MPI_ERR_FILE	MPI::ERR_FILE
MPI_ERR_INFO_KEY	MPI::ERR_INFO_VALUE
MPI_ERR_INFO_NOKEY	MPI::ERR_INFO_NOKEY
MPI_ERR_INFO_VALUE	MPI::ERR_INFO_KEY
MPI_ERR_INFO	MPI::ERR_INFO
MPI_ERR_IO	MPI::ERR_IO
MPI_ERR_KEYVAL	MPI::ERR_KEYVAL
MPI_ERR_LOCKTYPE	MPI::ERR_LOCKTYPE
MPI_ERR_NAME	MPI::ERR_NAME
MPI_ERR_NO_MEM	MPI::ERR_NO_MEM
MPI_ERR_NOT_SAME	MPI::ERR_NOT_SAME
MPI_ERR_NO_SPACE	MPI::ERR_NO_SPACE
MPI_ERR_NO_SUCH_FILE	MPI::ERR_NO_SUCH_FILE
MPI_ERR_PORT	MPI::ERR_PORT
MPI_ERR_QUOTA	MPI::ERR_QUOTA
MPI_ERR_READ_ONLY	MPI::ERR_READ_ONLY
MPI_ERR_RMA_CONFLICT	MPI::ERR_RMA_CONFLICT
MPI_ERR_RMA_SYNC	MPI::ERR_RMA_SYNC
MPI_ERR_SERVICE	MPI::ERR_SERVICE
MPI_ERR_SIZE	MPI::ERR_SIZE
MPI_ERR_SPAWN	MPI::ERR_SPAWN
MPI_ERR_UNSUPPORTED_DATAREP	MPI::ERR_UNSUPPORTED_DATAREP
MPI_ERR_UNSUPPORTED_OPERATION	MPI::ERR_UNSUPPORTED_OPERATION
MPI_ERR_WIN	MPI::ERR_WIN
MPI_ERR_LASTCODE	MPI::ERR_LASTCODE

Buffer Address Constants

C type: `void * const` Fortran type: (predefined memory location)	C++ type: `void * const`
MPI_BOTTOM	MPI::BOTTOM
MPI_IN_PLACE	MPI::IN_PLACE

Assorted Constants

C type: const int (or unnamed enum) Fortran type: INTEGER	C++ type: const int (or unnamed enum)
MPI_PROC_NULL	MPI::PROC_NULL
MPI_ANY_SOURCE	MPI::ANY_SOURCE
MPI_ANY_TAG	MPI::ANY_TAG
MPI_UNDEFINED	MPI::UNDEFINED
MPI_BSEND_OVERHEAD	MPI::BSEND_OVERHEAD
MPI_KEYVAL_INVALID	MPI::KEYVAL_INVALID
MPI_LOCK_EXCLUSIVE	MPI::LOCK_EXCLUSIVE
MPI_LOCK_SHARED	MPI::LOCK_SHARED
MPI_ROOT	MPI::ROOT

Status size and reserved index values (Fortran only)

Fortran type: INTEGER	
MPI_STATUS_SIZE	Not defined for C++
MPI_SOURCE	Not defined for C++
MPI_TAG	Not defined for C++
MPI_ERROR	Not defined for C++

Variable Address Size (Fortran only)

Fortran type: INTEGER	
MPI_ADDRESS_KIND	Not defined for C++
MPI_INTEGER_KIND	Not defined for C++
MPI_OFFSET_KIND	Not defined for C++

Error-handling specifiers

C type: MPI_Errhandler Fortran type: INTEGER	C++ type: MPI::Errhandler
MPI_ERRORS_ARE_FATAL	MPI::ERRORS_ARE_FATAL
MPI_ERRORS_RETURN	MPI::ERRORS_RETURN
	MPI::ERRORS_THROW_EXCEPTIONS

Maximum Sizes for Strings

C type: const int (or unnamed enum) Fortran type: INTEGER	C++ type: const int (or unnamed enum)
MPI_MAX_PROCESSOR_NAME	MPI::MAX_PROCESSOR_NAME
[ticket204.][] MPI_MAX_LIBRARY_VERSION_STRING	
MPI_MAX_ERROR_STRING	MPI::MAX_ERROR_STRING
MPI_MAX_DATAREP_STRING	MPI::MAX_DATAREP_STRING
MPI_MAX_INFO_KEY	MPI::MAX_INFO_KEY
MPI_MAX_INFO_VAL	MPI::MAX_INFO_VAL
MPI_MAX_OBJECT_NAME	MPI::MAX_OBJECT_NAME
MPI_MAX_PORT_NAME	MPI::MAX_PORT_NAME

Named Predefined Datatypes		**C/C++ types**
C type: `MPI_Datatype` Fortran type: `INTEGER`	C++ type: `MPI::Datatype`	
MPI_CHAR	MPI::CHAR	`char` (treated as printable character)
MPI_SHORT	MPI::SHORT	`signed short int`
MPI_INT	MPI::INT	`signed int`
MPI_LONG	MPI::LONG	`signed long`
MPI_LONG_LONG_INT	MPI::LONG_LONG_INT	`signed long long`
MPI_LONG_LONG	MPI::LONG_LONG	`long long` (synonym)
MPI_SIGNED_CHAR	MPI::SIGNED_CHAR	`signed char` (treated as integral value)
MPI_UNSIGNED_CHAR	MPI::UNSIGNED_CHAR	`unsigned char` (treated as integral value)
MPI_UNSIGNED_SHORT	MPI::UNSIGNED_SHORT	`unsigned short`
MPI_UNSIGNED	MPI::UNSIGNED	`unsigned int`
MPI_UNSIGNED_LONG	MPI::UNSIGNED_LONG	`unsigned long`
MPI_UNSIGNED_LONG_LONG	MPI::UNSIGNED_LONG_LONG	`unsigned long long`
MPI_FLOAT	MPI::FLOAT	`float`
MPI_DOUBLE	MPI::DOUBLE	`double`
MPI_LONG_DOUBLE	MPI::LONG_DOUBLE	`long double`
MPI_WCHAR	MPI::WCHAR	`wchar_t` (defined in `<stddef.h>`) (treated as printable character)
MPI_C_BOOL	(use C datatype handle)	`_Bool`
MPI_INT8_T	(use C datatype handle)	`int8_t`
MPI_INT16_T	(use C datatype handle)	`int16_t`
MPI_INT32_T	(use C datatype handle)	`int32_t`
MPI_INT64_T	(use C datatype handle)	`int64_t`
MPI_UINT8_T	(use C datatype handle)	`uint8_t`
MPI_UINT16_T	(use C datatype handle)	`uint16_t`
MPI_UINT32_T	(use C datatype handle)	`uint32_t`
MPI_UINT64_T	(use C datatype handle)	`uint64_t`
MPI_AINT	(use C datatype handle)	`MPI_Aint`
MPI_OFFSET	(use C datatype handle)	`MPI_Offset`
MPI_C_COMPLEX	(use C datatype handle)	`float _Complex`
MPI_C_FLOAT_COMPLEX	(use C datatype handle)	`float _Complex`
MPI_C_DOUBLE_COMPLEX	(use C datatype handle)	`double _Complex`
MPI_C_LONG_DOUBLE_COMPLEX	(use C datatype handle)	`long double _Complex`
MPI_BYTE	MPI::BYTE	(any C/C++ type)
MPI_PACKED	MPI::PACKED	(any C/C++ type)

Named Predefined Datatypes		**Fortran types**
C type: `MPI_Datatype` Fortran type: `INTEGER`	C++ type: `MPI::Datatype`	
MPI_INTEGER	MPI::INTEGER	`INTEGER`
MPI_REAL	MPI::REAL	`REAL`
MPI_DOUBLE_PRECISION	MPI::DOUBLE_PRECISION	`DOUBLE PRECISION`
MPI_COMPLEX	MPI::F_COMPLEX	`COMPLEX`
MPI_LOGICAL	MPI::LOGICAL	`LOGICAL`
MPI_CHARACTER	MPI::CHARACTER	`CHARACTER(1)`
MPI_AINT	(use C datatype handle)	`INTEGER (KIND=MPI_ADDRESS_KIND)`
MPI_OFFSET	(use C datatype handle)	`INTEGER (KIND=MPI_OFFSET_KIND)`
MPI_BYTE	MPI::BYTE	(any Fortran type)
MPI_PACKED	MPI::PACKED	(any Fortran type)

C++-Only Named Predefined Datatypes	**C++ types**
C++ type: `MPI::Datatype`	
MPI::BOOL	`bool`
MPI::COMPLEX	`Complex<float>`
MPI::DOUBLE_COMPLEX	`Complex<double>`
MPI::LONG_DOUBLE_COMPLEX	`Complex<long double>`

Optional datatypes (Fortran)		**Fortran types**
C type: `MPI_Datatype` Fortran type: `INTEGER`	C++ type: `MPI::Datatype`	
MPI_DOUBLE_COMPLEX	MPI::F_DOUBLE_COMPLEX	`DOUBLE COMPLEX`
MPI_INTEGER1	MPI::INTEGER1	`INTEGER*1`
MPI_INTEGER2	MPI::INTEGER2	`INTEGER*8`
MPI_INTEGER4	MPI::INTEGER4	`INTEGER*4`
MPI_INTEGER8	MPI::INTEGER8	`INTEGER*8`
MPI_INTEGER16		`INTEGER*16`
MPI_REAL2	MPI::REAL2	`REAL*2`
MPI_REAL4	MPI::REAL4	`REAL*4`
MPI_REAL8	MPI::REAL8	`REAL*8`
MPI_REAL16		`REAL*16`
MPI_COMPLEX4		`COMPLEX*4`
MPI_COMPLEX8		`COMPLEX*8`
MPI_COMPLEX16		`COMPLEX*16`
MPI_COMPLEX32		`COMPLEX*32`

Datatypes for reduction functions (C and C++)

C type: `MPI_Datatype` Fortran type: `INTEGER`	C++ type: `MPI::Datatype`
MPI_FLOAT_INT	MPI::FLOAT_INT
MPI_DOUBLE_INT	MPI::DOUBLE_INT
MPI_LONG_INT	MPI::LONG_INT
MPI_2INT	MPI::TWOINT
MPI_SHORT_INT	MPI::SHORT_INT
MPI_LONG_DOUBLE_INT	MPI::LONG_DOUBLE_INT

Datatypes for reduction functions (Fortran)

C type: `MPI_Datatype` Fortran type: `INTEGER`	C++ type: `MPI::Datatype`
MPI_2REAL	MPI::TWOREAL
MPI_2DOUBLE_PRECISION	MPI::TWODOUBLE_PRECISION
MPI_2INTEGER	MPI::TWOINTEGER

Special datatypes for constructing derived datatypes

C type: `MPI_Datatype` Fortran type: `INTEGER`	C++ type: `MPI::Datatype`
MPI_UB	MPI::UB
MPI_LB	MPI::LB

Reserved communicators

C type: `MPI_Comm` Fortran type: `INTEGER`	C++ type: `MPI::Intracomm`
MPI_COMM_WORLD	MPI::COMM_WORLD
MPI_COMM_SELF	MPI::COMM_SELF

Results of communicator and group comparisons

C type: `const int` (or unnamed `enum`) Fortran type: `INTEGER`	C++ type: `const int` (or unnamed `enum`)
MPI_IDENT	MPI::IDENT
MPI_CONGRUENT	MPI::CONGRUENT
MPI_SIMILAR	MPI::SIMILAR
MPI_UNEQUAL	MPI::UNEQUAL

Environmental inquiry keys

C type: `const int` (or unnamed `enum`) Fortran type: `INTEGER`	C++ type: `const int` (or unnamed `enum`)
MPI_TAG_UB	MPI::TAG_UB
MPI_IO	MPI::IO
MPI_HOST	MPI::HOST
MPI_WTIME_IS_GLOBAL	MPI::WTIME_IS_GLOBAL

Collective Operations

C type: `MPI_Op`	C++ type: `const MPI::Op`
Fortran type: `INTEGER`	
MPI_MAX	MPI::MAX
MPI_MIN	MPI::MIN
MPI_SUM	MPI::SUM
MPI_PROD	MPI::PROD
MPI_MAXLOC	MPI::MAXLOC
MPI_MINLOC	MPI::MINLOC
MPI_BAND	MPI::BAND
MPI_BOR	MPI::BOR
MPI_BXOR	MPI::BXOR
MPI_LAND	MPI::LAND
MPI_LOR	MPI::LOR
MPI_LXOR	MPI::LXOR
MPI_REPLACE	MPI::REPLACE

Null Handles

C/Fortran name C type / Fortran type	C++ name C++ type
MPI_GROUP_NULL `MPI_Group` / `INTEGER`	MPI::GROUP_NULL `const MPI::Group`
MPI_COMM_NULL `MPI_Comm` / `INTEGER`	MPI::COMM_NULL [1])
MPI_DATATYPE_NULL `MPI_Datatype` / `INTEGER`	MPI::DATATYPE_NULL `const MPI::Datatype`
MPI_REQUEST_NULL `MPI_Request` / `INTEGER`	MPI::REQUEST_NULL `const MPI::Request`
MPI_OP_NULL `MPI_Op` / `INTEGER`	MPI::OP_NULL `const MPI::Op`
MPI_ERRHANDLER_NULL `MPI_Errhandler` / `INTEGER`	MPI::ERRHANDLER_NULL `const MPI::Errhandler`
MPI_FILE_NULL `MPI_File` / `INTEGER`	MPI::FILE_NULL
MPI_INFO_NULL `MPI_Info` / `INTEGER`	MPI::INFO_NULL `const MPI::Info`
MPI_WIN_NULL `MPI_Win` / `INTEGER`	MPI::WIN_NULL

[1]) C++ type: See Section **??** on page **??** regarding class hierarchy and the specific type of MPI::COMM_NULL

Empty group

C type: `MPI_Group`	C++ type: `const MPI::Group`
Fortran type: `INTEGER`	
MPI_GROUP_EMPTY	MPI::GROUP_EMPTY

Topologies

C type: const int (or unnamed enum) Fortran type: INTEGER	C++ type: const int (or unnamed enum)
MPI_GRAPH	MPI::GRAPH
MPI_CART	MPI::CART
MPI_DIST_GRAPH	MPI::DIST_GRAPH

Predefined functions

C/Fortran name C type / Fortran type	C++ name C++ type
MPI_COMM_NULL_COPY_FN MPI_Comm_copy_attr_function / COMM_COPY_ATTR_FN	MPI_COMM_NULL_COPY_FN same as in C [1])
MPI_COMM_DUP_FN MPI_Comm_copy_attr_function / COMM_COPY_ATTR_FN	MPI_COMM_DUP_FN same as in C [1])
MPI_COMM_NULL_DELETE_FN MPI_Comm_delete_attr_function / COMM_DELETE_ATTR_FN	MPI_COMM_NULL_DELETE_FN same as in C [1])
MPI_WIN_NULL_COPY_FN MPI_Win_copy_attr_function / WIN_COPY_ATTR_FN	MPI_WIN_NULL_COPY_FN same as in C [1])
MPI_WIN_DUP_FN MPI_Win_copy_attr_function / WIN_COPY_ATTR_FN	MPI_WIN_DUP_FN same as in C [1])
MPI_WIN_NULL_DELETE_FN MPI_Win_delete_attr_function / WIN_DELETE_ATTR_FN	MPI_WIN_NULL_DELETE_FN same as in C [1])
MPI_TYPE_NULL_COPY_FN MPI_Type_copy_attr_function / TYPE_COPY_ATTR_FN	MPI_TYPE_NULL_COPY_FN same as in C [1])
MPI_TYPE_DUP_FN MPI_Type_copy_attr_function / TYPE_COPY_ATTR_FN	MPI_TYPE_DUP_FN same as in C [1])
MPI_TYPE_NULL_DELETE_FN MPI_Type_delete_attr_function / TYPE_DELETE_ATTR_FN	MPI_TYPE_NULL_DELETE_FN same as in C [1])

[1] See the advice to implementors on MPI_COMM_NULL_COPY_FN, ... in Section ?? on page ??

Deprecated predefined functions

C/Fortran name C type / Fortran type	C++ name C++ type
MPI_NULL_COPY_FN MPI_Copy_function / COPY_FUNCTION	MPI::NULL_COPY_FN MPI::Copy_function
MPI_DUP_FN MPI_Copy_function / COPY_FUNCTION	MPI::DUP_FN MPI::Copy_function
MPI_NULL_DELETE_FN MPI_Delete_function / DELETE_FUNCTION	MPI::NULL_DELETE_FN MPI::Delete_function

Predefined Attribute Keys

C type: const int (or unnamed enum) Fortran type: INTEGER	C++ type: const int (or unnamed enum)
MPI_APPNUM	MPI::APPNUM
MPI_LASTUSEDCODE	MPI::LASTUSEDCODE
MPI_UNIVERSE_SIZE	MPI::UNIVERSE_SIZE
MPI_WIN_BASE	MPI::WIN_BASE
MPI_WIN_DISP_UNIT	MPI::WIN_DISP_UNIT
MPI_WIN_SIZE	MPI::WIN_SIZE

Mode Constants

C type: const int (or unnamed enum) Fortran type: INTEGER	C++ type: const int (or unnamed enum)
MPI_MODE_APPEND	MPI::MODE_APPEND
MPI_MODE_CREATE	MPI::MODE_CREATE
MPI_MODE_DELETE_ON_CLOSE	MPI::MODE_DELETE_ON_CLOSE
MPI_MODE_EXCL	MPI::MODE_EXCL
MPI_MODE_NOCHECK	MPI::MODE_NOCHECK
MPI_MODE_NOPRECEDE	MPI::MODE_NOPRECEDE
MPI_MODE_NOPUT	MPI::MODE_NOPUT
MPI_MODE_NOSTORE	MPI::MODE_NOSTORE
MPI_MODE_NOSUCCEED	MPI::MODE_NOSUCCEED
MPI_MODE_RDONLY	MPI::MODE_RDONLY
MPI_MODE_RDWR	MPI::MODE_RDWR
MPI_MODE_SEQUENTIAL	MPI::MODE_SEQUENTIAL
MPI_MODE_UNIQUE_OPEN	MPI::MODE_UNIQUE_OPEN
MPI_MODE_WRONLY	MPI::MODE_WRONLY

Datatype Decoding Constants

C type: `const int` (or unnamed `enum`) Fortran type: `INTEGER`	C++ type: `const int` (or unnamed `enum`)
MPI_COMBINER_CONTIGUOUS	MPI::COMBINER_CONTIGUOUS
MPI_COMBINER_DARRAY	MPI::COMBINER_DARRAY
MPI_COMBINER_DUP	MPI::COMBINER_DUP
MPI_COMBINER_F90_COMPLEX	MPI::COMBINER_F90_COMPLEX
MPI_COMBINER_F90_INTEGER	MPI::COMBINER_F90_INTEGER
MPI_COMBINER_F90_REAL	MPI::COMBINER_F90_REAL
MPI_COMBINER_HINDEXED_INTEGER	MPI::COMBINER_HINDEXED_INTEGER
MPI_COMBINER_HINDEXED	MPI::COMBINER_HINDEXED
MPI_COMBINER_HVECTOR_INTEGER	MPI::COMBINER_HVECTOR_INTEGER
MPI_COMBINER_HVECTOR	MPI::COMBINER_HVECTOR
MPI_COMBINER_INDEXED_BLOCK	MPI::COMBINER_INDEXED_BLOCK
MPI_COMBINER_INDEXED	MPI::COMBINER_INDEXED
MPI_COMBINER_NAMED	MPI::COMBINER_NAMED
MPI_COMBINER_RESIZED	MPI::COMBINER_RESIZED
MPI_COMBINER_STRUCT_INTEGER	MPI::COMBINER_STRUCT_INTEGER
MPI_COMBINER_STRUCT	MPI::COMBINER_STRUCT
MPI_COMBINER_SUBARRAY	MPI::COMBINER_SUBARRAY
MPI_COMBINER_VECTOR	MPI::COMBINER_VECTOR

Threads Constants

C type: `const int` (or unnamed `enum`) Fortran type: `INTEGER`	C++ type: `const int` (or unnamed `enum`)
MPI_THREAD_FUNNELED	MPI::THREAD_FUNNELED
MPI_THREAD_MULTIPLE	MPI::THREAD_MULTIPLE
MPI_THREAD_SERIALIZED	MPI::THREAD_SERIALIZED
MPI_THREAD_SINGLE	MPI::THREAD_SINGLE

File Operation Constants, Part 1

C type: `const MPI_Offset` (or unnamed `enum`) Fortran type: `INTEGER (KIND=MPI_OFFSET_KIND)`	C++ type: `const MPI::Offset` (or unnamed `enum`)
MPI_DISPLACEMENT_CURRENT	MPI::DISPLACEMENT_CURRENT

File Operation Constants, Part 2

C type: `const int` (or unnamed `enum`) Fortran type: `INTEGER`	C++ type: `const int` (or unnamed `enum`)
MPI_DISTRIBUTE_BLOCK	MPI::DISTRIBUTE_BLOCK
MPI_DISTRIBUTE_CYCLIC	MPI::DISTRIBUTE_CYCLIC
MPI_DISTRIBUTE_DFLT_DARG	MPI::DISTRIBUTE_DFLT_DARG
MPI_DISTRIBUTE_NONE	MPI::DISTRIBUTE_NONE
MPI_ORDER_C	MPI::ORDER_C
MPI_ORDER_FORTRAN	MPI::ORDER_FORTRAN
MPI_SEEK_CUR	MPI::SEEK_CUR
MPI_SEEK_END	MPI::SEEK_END
MPI_SEEK_SET	MPI::SEEK_SET

F90 Datatype Matching Constants

C type: `const int` (or unnamed `enum`) Fortran type: `INTEGER`	C++ type: `const int` (or unnamed `enum`)
MPI_TYPECLASS_COMPLEX	MPI::TYPECLASS_COMPLEX
MPI_TYPECLASS_INTEGER	MPI::TYPECLASS_INTEGER
MPI_TYPECLASS_REAL	MPI::TYPECLASS_REAL

Constants Specifying Empty or Ignored Input

C/Fortran name C type / Fortran type	C++ name C++ type
MPI_ARGVS_NULL `char***` / 2-dim. array of `CHARACTER*(*)`	MPI::ARGVS_NULL `const char ***`
MPI_ARGV_NULL `char**` / array of `CHARACTER*(*)`	MPI::ARGV_NULL `const char **`
MPI_ERRCODES_IGNORE `int*` / `INTEGER` array	Not defined for C++
MPI_STATUSES_IGNORE `MPI_Status*` / `INTEGER, DIMENSION(MPI_STATUS_SIZE,*)`	Not defined for C++
MPI_STATUS_IGNORE `MPI_Status*` / `INTEGER, DIMENSION(MPI_STATUS_SIZE)`	Not defined for C++
MPI_UNWEIGHTED	Not defined for C++

C Constants Specifying Ignored Input (no C++ or Fortran)

C type: `MPI_Fint*`
MPI_F_STATUSES_IGNORE
MPI_F_STATUS_IGNORE

C and C++ preprocessor Constants and Fortran Parameters

C/C++ type: `const int` (or unnamed `enum`) Fortran type: `INTEGER`
MPI_SUBVERSION
MPI_VERSION

2.1.2 Types

The following are defined C type definitions, included in the file `mpi.h`.

```
/* C opaque types */
MPI_Aint
MPI_Fint
MPI_Offset
MPI_Status

/* C handles to assorted structures */
MPI_Comm
MPI_Datatype
MPI_Errhandler
MPI_File
MPI_Group
MPI_Info
MPI_Op
MPI_Request
MPI_Win

// C++ opaque types (all within the MPI namespace)
MPI::Aint
MPI::Offset
MPI::Status

// C++ handles to assorted structures (classes,
// all within the MPI namespace)
MPI::Comm
MPI::Intracomm
MPI::Graphcomm
MPI::Distgraphcomm
MPI::Cartcomm
MPI::Intercomm
MPI::Datatype
MPI::Errhandler
MPI::Exception
MPI::File
MPI::Group
MPI::Info
MPI::Op
MPI::Request
MPI::Prequest
MPI::Grequest
MPI::Win
```

2.1.3 Prototype [d]Definitions

ticket0.

The following are defined C typedefs for user-defined functions, also included in the file `mpi.h`.

```
/* prototypes for user-defined functions */
typedef void MPI_User_function(void *invec, void *inoutvec, int *len,
              MPI_Datatype *datatype);

typedef int MPI_Comm_copy_attr_function(MPI_Comm oldcomm,
              int comm_keyval, void *extra_state, void *attribute_val_in,
              void *attribute_val_out, int*flag);
typedef int MPI_Comm_delete_attr_function(MPI_Comm comm,
              int comm_keyval, void *attribute_val, void *extra_state);

typedef int MPI_Win_copy_attr_function(MPI_Win oldwin, int win_keyval,
              void *extra_state, void *attribute_val_in,
              void *attribute_val_out, int *flag);
typedef int MPI_Win_delete_attr_function(MPI_Win win, int win_keyval,
              void *attribute_val, void *extra_state);

typedef int MPI_Type_copy_attr_function(MPI_Datatype oldtype,
              int type_keyval, void *extra_state,
              void *attribute_val_in, void *attribute_val_out, int *flag);
typedef int MPI_Type_delete_attr_function(MPI_Datatype type,
              int type_keyval, void *attribute_val, void *extra_state);

typedef void MPI_Comm_errhandler_function(MPI_Comm *, int *, ...);
typedef void MPI_Win_errhandler_function(MPI_Win *, int *, ...);
typedef void MPI_File_errhandler_function(MPI_File *, int *, ...);

typedef int MPI_Grequest_query_function(void *extra_state,
            MPI_Status *status);
typedef int MPI_Grequest_free_function(void *extra_state);
typedef int MPI_Grequest_cancel_function(void *extra_state, int complete);

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

For Fortran, here are examples of how each of the user-defined subroutines should be declared.

The user-function argument to MPI_OP_CREATE should be declared like this:

```
SUBROUTINE USER_FUNCTION(INVEC, INOUTVEC, LEN, TYPE)
   <type> INVEC(LEN), INOUTVEC(LEN)
   INTEGER LEN, TYPE
```

The copy and delete function arguments to MPI_COMM_CREATE_KEYVAL should be declared like these:

```
SUBROUTINE COMM_COPY_ATTR_FN(OLDCOMM, COMM_KEYVAL, EXTRA_STATE,
             ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
   INTEGER OLDCOMM, COMM_KEYVAL, IERROR
   INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
             ATTRIBUTE_VAL_OUT
   LOGICAL FLAG

SUBROUTINE COMM_DELETE_ATTR_FN(COMM, COMM_KEYVAL, ATTRIBUTE_VAL,
             EXTRA_STATE, IERROR)
   INTEGER COMM, COMM_KEYVAL, IERROR
   INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
```

The copy and delete function arguments to MPI_WIN_CREATE_KEYVAL should be declared like these:

```
SUBROUTINE WIN_COPY_ATTR_FN(OLDWIN, WIN_KEYVAL, EXTRA_STATE,
             ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
   INTEGER OLDWIN, WIN_KEYVAL, IERROR
   INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
             ATTRIBUTE_VAL_OUT
   LOGICAL FLAG

SUBROUTINE WIN_DELETE_ATTR_FN(WIN, WIN_KEYVAL, ATTRIBUTE_VAL,
             EXTRA_STATE, IERROR)
   INTEGER WIN, WIN_KEYVAL, IERROR
   INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
```

The copy and delete function arguments to MPI_TYPE_CREATE_KEYVAL should be declared like these:

```
SUBROUTINE TYPE_COPY_ATTR_FN(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE,
              ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
   INTEGER OLDTYPE, TYPE_KEYVAL, IERROR
   INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE,
              ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT
   LOGICAL FLAG

SUBROUTINE TYPE_DELETE_ATTR_FN(TYPE, TYPE_KEYVAL, ATTRIBUTE_VAL,
              EXTRA_STATE, IERROR)
   INTEGER TYPE, TYPE_KEYVAL, IERROR
   INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
```

The handler-function argument to MPI_COMM_CREATE_ERRHANDLER should be declared like this:

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

The handler-function argument to MPI_WIN_CREATE_ERRHANDLER should be declared like this:

```
SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)
   INTEGER WIN, ERROR_CODE
```

The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be declared like this:

```
SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE)
   INTEGER FILE, ERROR_CODE
```

The query, free, and cancel function arguments to MPI_GREQUEST_START should be declared like these:

```
SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)
   INTEGER STATUS(MPI_STATUS_SIZE), IERROR
   INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE

SUBROUTINE GREQUEST_FREE_FUNCTION(EXTRA_STATE, IERROR)
   INTEGER IERROR
   INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE

SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR)
   INTEGER IERROR
   INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
   LOGICAL COMPLETE
```

The extend and conversion function arguments to MPI_REGISTER_DATAREP should be declared like these:

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

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

The following are defined C++ typedefs, also included in the file `mpi.h`.

```
namespace MPI {
  typedef void User_function(const void* invec, void *inoutvec,
               int len, const Datatype& datatype);

  typedef int Comm::Copy_attr_function(const Comm& oldcomm,
```

```
            int comm_keyval, void* extra_state, void* attribute_val_in,
            void* attribute_val_out, bool& flag);
  typedef int Comm::Delete_attr_function(Comm& comm, int
            comm_keyval, void* attribute_val, void* extra_state);

  typedef int Win::Copy_attr_function(const Win& oldwin,
            int win_keyval, void* extra_state, void* attribute_val_in,
            void* attribute_val_out, bool& flag);
  typedef int Win::Delete_attr_function(Win& win, int
            win_keyval, void* attribute_val, void* extra_state);

  typedef int Datatype::Copy_attr_function(const Datatype& oldtype,
            int type_keyval, void* extra_state,
            const void* attribute_val_in, void* attribute_val_out,
            bool& flag);
  typedef int Datatype::Delete_attr_function(Datatype& type,
            int type_keyval, void* attribute_val, void* extra_state);

  typedef void Comm::Errhandler_function(Comm &, int *, ...);
  typedef void Win::Errhandler_function(Win &, int *, ...);
  typedef void File::Errhandler_function(File &, int *, ...);

  typedef int Grequest::Query_function(void* extra_state, Status& status);
  typedef int Grequest::Free_function(void* extra_state);
  typedef int Grequest::Cancel_function(void* extra_state, bool complete);

  typedef void Datarep_extent_function(const Datatype& datatype,
             Aint& file_extent, void* extra_state);
  typedef void Datarep_conversion_function(void* userbuf,
             Datatype& datatype, int count, void* filebuf,
             Offset position, void* extra_state);
}
```

2.1.4 Deprecated [p]Prototype [d]Definitions

ticket0.
ticket0.

The following are defined C typedefs for deprecated user-defined functions, also included in the file mpi.h.

```
/* prototypes for user-defined functions */
typedef int MPI_Copy_function(MPI_Comm oldcomm, int keyval,
              void *extra_state, void *attribute_val_in,
              void *attribute_val_out, int *flag);
typedef int MPI_Delete_function(MPI_Comm comm, int keyval,
              void *attribute_val, void *extra_state);
typedef void MPI_Handler_function(MPI_Comm *, int *, ...);
```

The following are deprecated Fortran user-defined callback subroutine prototypes. The deprecated copy and delete function arguments to MPI_KEYVAL_CREATE should be declared like these:

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

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

The deprecated handler-function for error handlers should be declared like this:

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

2.1.5 Info Keys

access_style
appnum
arch
cb_block_size
cb_buffer_size
cb_nodes
chunked_item
chunked_size
chunked
collective_buffering
file_perm
filename
file
host
io_node_list
ip_address
ip_port
nb_proc
no_locks
num_io_nodes
path
soft
striping_factor
striping_unit
wdir

2.1.6 Info Values

false
random
read_mostly
read_once

reverse_sequential
sequential
true
write_mostly
write_once

Bibliography

[1] Martin Schulz and Bronis R. de Supinski. P^{N}MPI Tools: A Whole Lot Greater Than the Sum of Their Parts. In *ACM/IEEE Supercomputing Conference (SC)*, pages 1–10. ACM, 2007.

Examples Index

This index lists code examples throughout the text. Some examples are referred to by content; others are listed by the major MPI function that they are demonstrating. MPI functions listed in all capital letter are Fortran examples; MPI functions listed in mixed case are C/C++ examples.

MPI Constant and Predefined Handle Index

This index lists predefined MPI constants and handles.

MPI Declarations Index

This index refers to declarations needed in C/C++, such as address kind integers, handles, etc. The underlined page numbers is the “main” reference (sometimes there are more than one when key concepts are discussed in multiple areas).

MPI Callback Function Prototype Index

This index lists the C typedef names for callback routines, such as those used with attribute caching or user-defined reduction operations. C++ names for these typedefs and Fortran example prototypes are given near the text of the C name.

MPI Function Index

The underlined page numbers refer to the function definitions.