D R A F T

Document for a Standard Message-Passing Interface

Message Passing Interface Forum

October 7, 2021

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 8

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.

8.1 Implementation Information

8.1.1 Version Inquiries

In order to cope with changes to the MPI Standard, there are both compile-time and runtime 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,

```
#define MPI_VERSION    4
#define MPI_SUBVERSION 0
```

in Fortran,

```
INTEGER :: MPI_VERSION, MPI_SUBVERSION
PARAMETER (MPI_VERSION    = 4)
PARAMETER (MPI_SUBVERSION = 0)
```

For runtime determination,

MPI_GET_VERSION(version, subversion)

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

C binding

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

Fortran 2008 binding

```
MPI_Get_version(version, subversion, ierror)
```

```
    INTEGER, INTENT(OUT) :: version, subversion
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

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

MPI_GET_VERSION can be called at any time in an MPI program. This function must always be thread-safe, as defined in Section 11.6. Valid (MPI_VERSION, MPI_SUBVERSION) pairs in this and previous versions of the MPI standard are (4,0), (3,1), (3,0), (2,2), (2,1), (2,0), and (1,2).

MPI_GET_LIBRARY_VERSION(version, resultlen)

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

C binding

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

Fortran 2008 binding

```
MPI_Get_library_version(version, resultlen, ierror)
    CHARACTER(LEN=MPI_MAX_LIBRARY_VERSION_STRING), INTENT(OUT) :: version
    INTEGER, INTENT(OUT) :: resultlen
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_GET_LIBRARY_VERSION(VERSION, RESULTLEN, 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 version[resultlen]. The value of resultlen cannot be larger than MPI_MAX_LIBRARY_VERSION_STRING - 1. In Fortran, version is padded on the right with blank characters. The value of resultlen cannot be larger than MPI_MAX_LIBRARY_VERSION_STRING.

MPI_GET_LIBRARY_VERSION can be called at any time in an MPI program. This function must always be thread-safe, as defined in Section 11.6.

8.1.2 Environmental Inquiries

When using the World Model (Section 11.2), a set of attributes that describe the execution environment is attached to the communicator MPI_COMM_WORLD when MPI is initialized. The values of these attributes can be inquired by using the function MPI_COMM_GET_ATTR described in Section 7.7 and in Section 19.3.7. 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.

When using the Sessions Model (Section 11.3), only the MPI_TAG_UB attribute is available.

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 valid value for MPI_TAG_UB.

In the Sessions Model, the attribute MPI_TAG_UB is attached to all communicators created by MPI_COMM_CREATE_FROM_GROUP and MPI_INTERCOMM_CREATE_FROM_GROUPS, with the same value on all MPI processes in the communicator. In the World Model, 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, this means that all of the ISO 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.

Inquire Processor Name

MPI_GET_PROCESSOR_NAME(name, resultlen)

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

C binding

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

Fortran 2008 binding

```
MPI_Get_processor_name(name, resultlen, ierror)
    CHARACTER(LEN=MPI_MAX_PROCESSOR_NAME), INTENT(OUT) :: name
    INTEGER, INTENT(OUT) :: resultlen
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

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

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 value of resultlen cannot be larger than MPI_MAX_PROCESSOR_NAME-1. In Fortran, name is padded on the right with blank characters. The value of resultlen cannot be larger 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.*)

Inquire Hardware Resource Types and Status

MPI_GET_HW_RESOURCE_TYPES(hw_info)

OUT	hw_info	info object created (handle)

C binding

```
int MPI_Get_hw_resource_types(MPI_Info *hw_info)
```

Fortran 2008 binding

```
MPI_Get_hw_resource_types(hw_info, ierror)
    TYPE(MPI_Info), INTENT(OUT) :: hw_info
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_GET_HW_RESOURCE_TYPES(HW_INFO, IERROR)
    INTEGER HW_INFO, IERROR
```

MPI_GET_HW_RESOURCE_TYPES returns an info object containing information pertaining to the hardware platform on which the calling MPI process is executing at the moment of the call. The information is stored in the following info keys:

- "mpi_hw_res_nresources" is an integer that represents the number of all hardware resource types recognized by the MPI implementation and to which the calling MPI process can be restricted.
- "mpi_hw_res_i_type" is the type of the i-th hardware resource to which the calling MPI process can be restricted (with $i \in \{0, \ldots,$ "mpi_hw_res_nresources" $- 1\}$).
- "mpi_hw_res_i_naliases" is an integer that represents the number of hardware resource types that are aliases to "mpi_hw_res_i_type" (with $i \in \{0, \ldots,$ "mpi_hw_res_nresources"$-1\}$).
- "mpi_hw_res_i_alias_k" with $k \in \{0, \ldots,$ "mpi_hw_res_i_naliases"$-1\}$ is an integer j (with $j \in \{0, \ldots,$ "mpi_hw_res_nresources" $- 1\}$) such that "mpi_hw_res_j_type" is an alias to "mpi_hw_res_i_type".
- "mpi_hw_res_i_occupied", where $i \in \{0, \ldots,$ "mpi_hw_res_nresources" $- 1\}$, is "true" if the calling MPI process is restricted to an instance of a hardware resource of type "mpi_hw_res_i_type" at the moment of the call. Otherwise, it is "false".

This local routine can return different information if the MPI processes are executing on different hardware resources. The user is responsible for freeing hw_info via MPI_INFO_FREE.

Advice to users. The types returned by this routine can be used in MPI_COMM_SPLIT_TYPE with the split_type value MPI_COMM_TYPE_HW_GUIDED as key values for the info key "mpi_hw_resource_type". However, the information returned in hw_info may not be constant throughout the execution of the program because an MPI process can relocate (e.g., migrate or change its hardware restrictions). (*End of advice to users.*)

The following routine can be used to detect whether an MPI process is restricted to a hardware resource which type is returned by MPI_GET_HW_RESOURCE_TYPES.

MPI_GET_HW_RESOURCE_STATUS(name, status)

IN	name	name of hardware resource type (string)
OUT	status	status of the hardware resource type queried upon (integer)

C binding

```
int MPI_Get_hw_resource_status(char *name, int *status)
```

Fortran 2008 binding

```
MPI_Get_hw_resource_status(name, status, ierror)
    CHARACTER(LEN=MPI_MAX_INFO_KEY), INTENT(IN) :: name
    INTEGER, INTENT(OUT) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_GET_HW_RESOURCE_STATUS(NAME, STATUS, IERROR)
```

```
    CHARACTER*(*) NAME
    INTEGER STATUS, IERROR
```

MPI_HW_PRESENT results if a hardware resource of type name is recognized but the calling MPI process cannot be restricted to an instance of a resource of this type (e.g., a network interface card). MPI_HW_USABLE results if the calling MPI process is able to be restricted to an instance of a hardware resource of type name at the moment of the call. MPI_HW_OCCUPIED results if the calling MPI process is actually restricted to an instance of a hardware resource of type name at the moment of the call. MPI_HW_UNKNOWN results otherwise.

If the status of a hardware resource is MPI_HW_OCCUPIED, its type can then be used as the value for the info key "mpi_hw_resource_type" in the function MPI_COMM_SPLIT_TYPE with the split_type value MPI_COMM_TYPE_HW_GUIDED. In that case, the splitting operation is guaranteed to produce a valid communicator for that MPI process.

8.2 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 some RMA functionality as defined in Section 12.5.3.

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

C binding

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

Fortran 2008 binding

```
MPI_Alloc_mem(size, info, baseptr, ierror)
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(C_PTR), INTENT(OUT) :: baseptr
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

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

If the Fortran compiler provides TYPE(C_PTR), then the following generic interface must be provided in the mpi module and should be provided in mpif.h through overloading, i.e., with the same routine name as the routine with INTEGER(KIND=MPI_ADDRESS_KIND) BASEPTR, but with a different specific procedure name:

```
INTERFACE MPI_ALLOC_MEM
    SUBROUTINE MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
        IMPORT :: MPI_ADDRESS_KIND
        INTEGER :: INFO, IERROR
        INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE, BASEPTR
    END SUBROUTINE
    SUBROUTINE MPI_ALLOC_MEM_CPTR(SIZE, INFO, BASEPTR, IERROR)
        USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
        IMPORT :: MPI_ADDRESS_KIND
        INTEGER :: INFO, IERROR
        INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
        TYPE(C_PTR) :: BASEPTR
    END SUBROUTINE
END INTERFACE
```

The base procedure name of this overloaded function is MPI_ALLOC_MEM_CPTR. The implied specific procedure names are described in Section 19.1.5.

By default, the allocated memory shall be aligned to at least the alignment required for load/store accesses of any datatype corresponding to a predefined MPI datatype. The info argument may be used to specify a desired alternative minimum alignment in bytes for the allocated memory by setting the value of the key "mpi_minimum_memory_alignment" to an integral number equal to a power of two. An implementation may ignore values smaller than the default required alignment. The info argument can also 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. The corresponding 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)

C binding

```
int MPI_Free_mem(void *base)
```

Fortran 2008 binding

```
MPI_Free_mem(base, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: base
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_FREE_MEM(BASE, IERROR)
```

```
<type> BASE(*)
INTEGER IERROR
```

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

Rationale. The 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 bindings: the Fortran MPI_ALLOC_MEM call returns in baseptr the `TYPE(C_PTR)` pointer or 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 8.1 Example of use of MPI_ALLOC_MEM, in Fortran with `TYPE(C_PTR)` pointers. We assume 4-byte `REAL`s.

```
USE mpi_f08   !  or  USE mpi      (not guaranteed with INCLUDE 'mpif.h')
USE, INTRINSIC :: ISO_C_BINDING
TYPE(C_PTR) :: p
REAL, DIMENSION(:,:), POINTER :: a           ! no memory is allocated
INTEGER, DIMENSION(2) :: shape
INTEGER(KIND=MPI_ADDRESS_KIND) :: size
shape = (/100,100/)
size = 4 * shape(1) * shape(2)               ! assuming 4 bytes per REAL
CALL MPI_Alloc_mem(size,MPI_INFO_NULL,p,ierr) ! memory is allocated and
CALL C_F_POINTER(p, a, shape) ! intrinsic     ! now accessible via a(i,j)
...                           ! in ISO_C_BINDING
a(3,5) = 2.71
...
CALL MPI_Free_mem(a, ierr)                   ! memory is freed
```

Example 8.2 Example of use of MPI_ALLOC_MEM, in Fortran with nonstandard *Cray-pointers*. We assume 4-byte `REAL`s, and assume that these pointers are address-sized.

```
REAL A
POINTER (P, A(100,100))   ! no memory is allocated
INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
```

```
SIZE = 4*100*100
CALL MPI_ALLOC_MEM(SIZE, MPI_INFO_NULL, P, IERR)
! memory is allocated
...
A(3,5) = 2.71
...
CALL MPI_FREE_MEM(A, IERR) ! memory is freed
```

This code is not Fortran 77 or Fortran 90 code. Some compilers may not support this code or need a special option, e.g., the GNU gFortran compiler needs `-fcray-pointer`.

Advice to implementors. Some compilers map Cray-pointers to address-sized integers, some to `TYPE(C_PTR)` pointers (e.g., Cray Fortran, version 7.3.3). From the user's viewpoint, this mapping is irrelevant because Examples 8.2 should work correctly with an MPI-3.0 (or later) library if Cray-pointers are available. (*End of advice to implementors.*)

Example 8.3 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);
```

8.3 Error Handling

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

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. More background information about how MPI treats errors can be found in Section 2.8.

A user can associate error handlers to four types of objects: communicators, windows, files, and sessions. The specified error handling routine will be used for any error that occurs during a call to MPI for the respective object. MPI calls that are not related to any MPI objects are considered to be attached to the communicator MPI_COMM_SELF when using the World Model (see Section 11.2). When MPI_COMM_SELF is not initialized (i.e., before MPI_INIT / MPI_INIT_THREAD, after MPI_FINALIZE, or when using the Sessions Model exclusively) the error raises the initial error handler (set during the launch operation, see 11.8.4). 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 all connected MPI processes. This is similar to calling MPI_ABORT using a communicator containing all connected processes with an implementation-specific value as the errorcode argument.

MPI_ERRORS_ABORT The handler, when called, is invoked on a communicator in a manner similar to calling MPI_ABORT on that communicator. If the error handler is invoked on an window or file, it is similar to calling MPI_ABORT using a communicator containing the group of MPI processes associated with the window or file, respectively. If the error handler is invoked on a session, the operation aborts only the local MPI process. In all cases, the value that would be provided as the errorcode argument to MPI_ABORT is implementation-specific.

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

Advice to implementors. The implementation-specific error information resulting from MPI_ERRORS_ARE_FATAL and MPI_ERRORS_ABORT provided to the invoking environment should be meaningful to the end-user, for example a predefined error class. (*End of advice to implementors.*)

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

Unless otherwise requested, the error handler MPI_ERRORS_ARE_FATAL is set as the default initial error handler and associated with predefined communicators. 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 nontrivial MPI error handler. Note that unlike predefined communicators, windows and files do not inherit from the initial error handler, as defined in Sections 12.6 and 14.7 respectively.

When an error is raised, MPI will provide the user information about that error using an error code. Some errors might prevent MPI from completing further API calls successfully and those functions will continue to report errors until the cause of the error is corrected or the user terminates the application. The user can make the determination of whether or not to attempt to continue when handling such an error.

Advice to users. For example, users may be unable to correct errors corresponding to some error classes, such as MPI_ERR_INTERN. Such errors may cause subsequent MPI calls to complete in error. (*End of advice to users.*)

Advice to implementors. A high-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 and available recovery actions. (*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 has distinct typedefs for user defined error handling callback functions that accept communicator, file, window, and session arguments. In Fortran there are four user routines.

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

An error handler is attached to a communicator, window, file, or session 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. An error handler can also be attached to a session using the errorhandler argument to MPI_SESSION_INIT. The predefined error handlers MPI_ERRORS_RETURN and MPI_ERRORS_ARE_FATAL can be attached to communicators, windows, files, or sessions.

The error handler currently associated with a communicator, window, file, or session 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_XXX_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_XXX_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 implementations 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.

8.3.1 Error Handlers for Communicators

MPI_COMM_CREATE_ERRHANDLER(comm_errhandler_fn, errhandler)

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

C binding

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

Fortran 2008 binding

```
MPI_Comm_create_errhandler(comm_errhandler_fn, errhandler, ierror)
    PROCEDURE(MPI_Comm_errhandler_function) :: comm_errhandler_fn
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
```

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

Fortran binding

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

Creates an error handler that can be attached to communicators.

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 *comm, int *error_code,
             ...);
```

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 "`varargs`" 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. With the Fortran `mpi_f08` module, the user routine comm_errhandler_fn should be of the form:

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Comm_errhandler_function(comm, error_code)
    TYPE(MPI_Comm) :: comm
    INTEGER :: error_code
```

With the Fortran `mpi` module and `mpif.h`, the user routine COMM_ERRHANDLER_FN should be of the form:

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

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)

C binding

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

Fortran 2008 binding

```
MPI_Comm_set_errhandler(comm, errhandler, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

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

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.

MPI_COMM_GET_ERRHANDLER(comm, errhandler)

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

C binding

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

Fortran 2008 binding

```
MPI_Comm_get_errhandler(comm, errhandler, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

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

Retrieves the error handler currently associated with a communicator.

For 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.

8.3.2 Error Handlers for Windows

MPI_WIN_CREATE_ERRHANDLER(win_errhandler_fn, errhandler)

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

C binding

```
int MPI_Win_create_errhandler(
            MPI_Win_errhandler_function *win_errhandler_fn,
```

```
            MPI_Errhandler *errhandler)
```

Fortran 2008 binding

```
MPI_Win_create_errhandler(win_errhandler_fn, errhandler, ierror)
    PROCEDURE(MPI_Win_errhandler_function) :: win_errhandler_fn
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

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

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 *win, int *error_code,
             ...);
```

The first argument is the window in use, the second is the error code to be returned. The remaining arguments are "varargs" arguments whose number and meaning is implementation-dependent. An implementation should clearly document these arguments. With the Fortran mpi_f08 module, the user routine win_errhandler_fn should be of the form:

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Win_errhandler_function(win, error_code)
    TYPE(MPI_Win) :: win
    INTEGER :: error_code
```

With the Fortran mpi module and mpif.h, the user routine WIN_ERRHANDLER_FN should be of the form:

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

MPI_WIN_SET_ERRHANDLER(win, errhandler)

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

C binding

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

Fortran 2008 binding

```
MPI_Win_set_errhandler(win, errhandler, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

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

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 object (handle)
OUT	errhandler	error handler currently associated with window (handle)

C binding

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

Fortran 2008 binding

```
MPI_Win_get_errhandler(win, errhandler, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

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

Retrieves the error handler currently associated with a window.

8.3.3 Error Handlers for Files

MPI_FILE_CREATE_ERRHANDLER(file_errhandler_fn, errhandler)

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

C binding

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

Fortran 2008 binding

```
MPI_File_create_errhandler(file_errhandler_fn, errhandler, ierror)
    PROCEDURE(MPI_File_errhandler_function) :: file_errhandler_fn
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

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

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 *file, int *error_code,
             ...);
```

The first argument is the file in use, the second is the error code to be returned. The remaining arguments are "varargs" arguments whose number and meaning is implementation-dependent. An implementation should clearly document these arguments.
With the Fortran mpi_f08 module, the user routine file_errhandler_fn should be of the form:

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_File_errhandler_function(file, error_code)
    TYPE(MPI_File) :: file
    INTEGER :: error_code
```

With the Fortran mpi module and mpif.h, the user routine FILE_ERRHANDLER_FN should be of the form:

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

MPI_FILE_SET_ERRHANDLER(file, errhandler)

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

C binding

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

Fortran 2008 binding

```
MPI_File_set_errhandler(file, errhandler, ierror)
    TYPE(MPI_File), INTENT(IN) :: file
    TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

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

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)

C binding

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

Fortran 2008 binding

```
MPI_File_get_errhandler(file, errhandler, ierror)
    TYPE(MPI_File), INTENT(IN) :: file
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

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

Retrieves the error handler currently associated with a file.

8.3.4 Error Handlers for Sessions

MPI_SESSION_CREATE_ERRHANDLER(session_errhandler_fn, errhandler)

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

C binding

```
int MPI_Session_create_errhandler(
             MPI_Session_errhandler_function *session_errhandler_fn,
             MPI_Errhandler *errhandler)
```

Fortran 2008 binding

```
MPI_Session_create_errhandler(session_errhandler_fn, errhandler, ierror)
    PROCEDURE(MPI_Session_errhandler_function) :: session_errhandler_fn
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_SESSION_CREATE_ERRHANDLER(SESSION_ERRHANDLER_FN, ERRHANDLER, IERROR)
    EXTERNAL SESSION_ERRHANDLER_FN
    INTEGER ERRHANDLER, IERROR
```

Creates an error handler that can be attached to a session object. In C, the session_errhandler_fn argument should be a function of type MPI_Session_errhandler_function, which is defined as

```
typedef void MPI_Session_errhandler_function(MPI_Session *session,
             int *error_code, ...);
```

The first argument is the session in use, the second is the error code to be returned. The remaining arguments are "varargs" arguments whose number and meaning is implementation-dependent. An implementation should clearly document these arguments. With the Fortran mpi_f08 module, the session_errhandler_fn argument should be of the form:

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Session_errhandler_function(session, error_code)
    TYPE(MPI_Session) :: session
```

```
    INTEGER :: error_code
```

With the Fortran mpi module and mpif.h, the SESSION_ERRHANDLER_FN argument should be of the form:

```
SUBROUTINE SESSION_ERRHANDLER_FUNCTION(SESSION, ERROR_CODE)
    INTEGER SESSION, ERROR_CODE
```

MPI_SESSION_SET_ERRHANDLER(session, errhandler)

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

C binding

```
int MPI_Session_set_errhandler(MPI_Session session,
             MPI_Errhandler errhandler)
```

Fortran 2008 binding

```
MPI_Session_set_errhandler(session, errhandler, ierror)
    TYPE(MPI_Session), INTENT(IN) :: session
    TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_SESSION_SET_ERRHANDLER(SESSION, ERRHANDLER, IERROR)
    INTEGER SESSION, ERRHANDLER, IERROR
```

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

MPI_SESSION_GET_ERRHANDLER(session, errhandler)

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

C binding

```
int MPI_Session_get_errhandler(MPI_Session session,
             MPI_Errhandler *errhandler)
```

Fortran 2008 binding

```
MPI_Session_get_errhandler(session, errhandler, ierror)
    TYPE(MPI_Session), INTENT(IN) :: session
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_SESSION_GET_ERRHANDLER(SESSION, ERRHANDLER, IERROR)
    INTEGER SESSION, ERRHANDLER, IERROR
```

Retrieves the error handler currently associated with a session.

8.3.5 Freeing Errorhandlers and Retrieving Error Strings

MPI_ERRHANDLER_FREE(errhandler)

INOUT	errhandler	MPI error handler (handle)

C binding

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

Fortran 2008 binding

```
MPI_Errhandler_free(errhandler, ierror)
    TYPE(MPI_Errhandler), INTENT(INOUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

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

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

C binding

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

Fortran 2008 binding

```
MPI_Error_string(errorcode, string, resultlen, ierror)
    INTEGER, INTENT(IN) :: errorcode
    CHARACTER(LEN=MPI_MAX_ERROR_STRING), INTENT(OUT) :: string
    INTEGER, INTENT(OUT) :: resultlen
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

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

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.

This function must always be thread-safe, as defined in Section 11.6. It is one of the few routines that may be called before MPI is initialized or after MPI is finalized.

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.*)

8.4 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).

All MPI function calls shall return MPI_SUCCESS if and only if the specification of that function has been fulfilled at the point of return. For multiple completion functions, if the function returns MPI_ERR_IN_STATUS, the error code in each status object shall be set to MPI_SUCCESS if and only if the specification of the operation represented by the corresponding MPI_Request has been fulfilled at the point of return.

When an operation raises an error, it may not satisfy its specification (for example, a synchronizing operation may not have synchronized) and the content of the output buffers, targeted memory, or output parameters is undefined. However, a valid error code shall always be set when an operation raises an error, whether in the return value, error field in the status object, or element in an array of error codes.

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 8.1 and Table 8.2.

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. The values defined for MPI error classes are valid MPI error codes.

The error codes satisfy,

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

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_SUCCESS	No error
MPI_ERR_ACCESS	Permission denied
MPI_ERR_AMODE	Error related to the amode passed to MPI_FILE_OPEN
MPI_ERR_ARG	Invalid argument of some other kind
MPI_ERR_ASSERT	Invalid assertion argument
MPI_ERR_BAD_FILE	Invalid file name (e.g., path name too long)
MPI_ERR_BASE	Invalid base passed to MPI_FREE_MEM
MPI_ERR_BUFFER	Invalid buffer pointer argument
MPI_ERR_COMM	Invalid communicator argument
MPI_ERR_CONVERSION	An error occurred in a user supplied data conversion function
MPI_ERR_COUNT	Invalid count argument
MPI_ERR_DIMS	Invalid dimension argument
MPI_ERR_DISP	Invalid displacement argument
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_FILE	Invalid file handle argument
MPI_ERR_FILE_EXISTS	File exists
MPI_ERR_FILE_IN_USE	File operation could not be completed, as the file is currently open by some process
MPI_ERR_GROUP	Invalid group argument
MPI_ERR_INFO	Invalid info argument
MPI_ERR_INFO_KEY	Key longer than MPI_MAX_INFO_KEY
MPI_ERR_INFO_NOKEY	Invalid key passed to MPI_INFO_DELETE
MPI_ERR_INFO_VALUE	Value longer than MPI_MAX_INFO_VAL
MPI_ERR_IN_STATUS	Error code is in status
MPI_ERR_INTERN	Internal MPI (implementation) error
MPI_ERR_IO	Other I/O error
MPI_ERR_KEYVAL	Invalid keyval argument
MPI_ERR_LOCKTYPE	Invalid locktype argument
MPI_ERR_NAME	Invalid service name passed to MPI_LOOKUP_NAME
MPI_ERR_NO_MEM	MPI_ALLOC_MEM failed because memory is exhausted
MPI_ERR_NO_SPACE	Not enough space
MPI_ERR_NO_SUCH_FILE	File does not exist
MPI_ERR_NOT_SAME	Collective argument not identical on all processes, or collective routines called in a different order by different processes

Table 8.1: Error classes (Part 1)

MPI_ERR_OP	Invalid operation argument
MPI_ERR_OTHER	Known error not in this list
MPI_ERR_PENDING	Pending request
MPI_ERR_PORT	Invalid port name passed to MPI_COMM_CONNECT
MPI_ERR_PROC_ABORTED	Operation failed because a peer process has aborted
MPI_ERR_QUOTA	Quota exceeded
MPI_ERR_RANK	Invalid rank argument
MPI_ERR_READ_ONLY	Read-only file or file system
MPI_ERR_REQUEST	Invalid request argument
MPI_ERR_RMA_ATTACH	Memory cannot be attached (e.g., because of resource exhaustion)
MPI_ERR_RMA_CONFLICT	Conflicting accesses to window
MPI_ERR_RMA_FLAVOR	Passed window has the wrong flavor for the called function
MPI_ERR_RMA_RANGE	Target memory is not part of the window (in the case of a window created with MPI_WIN_CREATE_DYNAMIC, target memory is not attached)
MPI_ERR_RMA_SHARED	Memory cannot be shared (e.g., some process in the group of the specified communicator cannot expose shared memory)
MPI_ERR_RMA_SYNC	Wrong synchronization of RMA calls
MPI_ERR_ROOT	Invalid root argument
MPI_ERR_SERVICE	Invalid service name passed to MPI_UNPUBLISH_NAME
MPI_ERR_SESSION	Invalid session argument
MPI_ERR_SIZE	Invalid size argument
MPI_ERR_SPAWN	Error in spawning processes
MPI_ERR_TAG	Invalid tag argument
MPI_ERR_TOPOLOGY	Invalid topology argument
MPI_ERR_TRUNCATE	Message truncated on receive
MPI_ERR_TYPE	Invalid datatype argument
MPI_ERR_UNKNOWN	Unknown error
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_VALUE_TOO_LARGE	Value is too large to store
MPI_ERR_WIN	Invalid window argument
MPI_ERR_LASTCODE	Last error code

Table 8.2: Error classes (Part 2)

MPI_ERROR_CLASS(errorcode, errorclass)

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

C binding

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

Fortran 2008 binding

```
MPI_Error_class(errorcode, errorclass, ierror)
    INTEGER, INTENT(IN) :: errorcode
    INTEGER, INTENT(OUT) :: errorclass
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

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

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

This function must always be thread-safe, as defined in Section 11.6. It is one of the few routines that may be called before MPI is initialized or after MPI is finalized.

8.5 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 14. 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)

C binding

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

Fortran 2008 binding

```
MPI_Add_error_class(errorclass, ierror)
    INTEGER, INTENT(OUT) :: errorclass
```

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

Fortran binding

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

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 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. 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 multithreaded 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 be associated with errorclass (integer)

C binding

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

Fortran 2008 binding

```
MPI_Add_error_code(errorclass, errorcode, ierror)
    INTEGER, INTENT(IN) :: errorclass
    INTEGER, INTENT(OUT) :: errorcode
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

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

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.*)

MPI_ADD_ERROR_STRING(errorcode, string)

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

C binding

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

Fortran 2008 binding

```
MPI_Add_error_string(errorcode, string, ierror)
    INTEGER, INTENT(IN) :: errorcode
    CHARACTER(LEN=*), INTENT(IN) :: string
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

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

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. 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).

Section 8.3 describes the methods for creating and associating error handlers with communicators, files, windows, and sessions.

MPI_COMM_CALL_ERRHANDLER(comm, errorcode)

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

C binding

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

Fortran 2008 binding

```
MPI_Comm_call_errhandler(comm, errorcode, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: errorcode
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

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

This function invokes the error handler assigned to the communicator with the error code supplied. This function returns MPI_SUCCESS in 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).

MPI_WIN_CALL_ERRHANDLER(win, errorcode)

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

C binding

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

Fortran 2008 binding

```
MPI_Win_call_errhandler(win, errorcode, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, INTENT(IN) :: errorcode
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

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

This function invokes the error handler assigned to the window with the error code supplied. This function returns MPI_SUCCESS in 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.* In contrast to communicators, the error handler MPI_ERRORS_ARE_FATAL is associated with a window when it is created. (*End of advice to users.*)

MPI_FILE_CALL_ERRHANDLER(fh, errorcode)

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

C binding

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

Fortran 2008 binding

```
MPI_File_call_errhandler(fh, errorcode, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER, INTENT(IN) :: errorcode
```

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

Fortran binding

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

This function invokes the error handler assigned to the file with the error code supplied. This function returns MPI_SUCCESS in 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.* The default error handler for files is MPI_ERRORS_RETURN. (*End of advice to users.*)

MPI_SESSION_CALL_ERRHANDLER(session, errorcode)

IN	session	session with error handler (handle)
IN	errorcode	error code (integer)

C binding

```
int MPI_Session_call_errhandler(MPI_Session session, int errorcode)
```

Fortran 2008 binding

```
MPI_Session_call_errhandler(session, errorcode, ierror)
    TYPE(MPI_Session), INTENT(IN) :: session
    INTEGER, INTENT(IN) :: errorcode
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_SESSION_CALL_ERRHANDLER(SESSION, ERRORCODE, IERROR)
    INTEGER SESSION, ERRORCODE, IERROR
```

This function invokes the error handler assigned to the session with the error code supplied. This function returns MPI_SUCCESS in 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 are warned that handlers should not be called recursively with MPI_COMM_CALL_ERRHANDLER, MPI_FILE_CALL_ERRHANDLER, MPI_WIN_CALL_ERRHANDLER, or MPI_SESSION_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, MPI_WIN_CALL_ERRHANDLER, or MPI_SESSION_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.*)

8.6 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 2.6.4.

MPI_WTIME()

C binding
`double MPI_Wtime(void)`

Fortran 2008 binding
`DOUBLE PRECISION MPI_Wtime()`

Fortran binding
`DOUBLE PRECISION MPI_WTIME()`

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"), and it allows high-resolution. 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 in Section 8.1.2).

MPI_WTICK()

C binding
`double MPI_Wtick(void)`

Fortran 2008 binding
`DOUBLE PRECISION MPI_Wtick()`

Fortran binding
`DOUBLE PRECISION MPI_WTICK()`

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}).

Index