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 15.2)* }

{`void MPI::Init()` *(binding deprecated, see Section 15.2)* }

All MPI processes 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_INITIALIZED, and MPI_FINALIZED.

The version for ISO C accepts the argc and argv that are provided by the arguments to main or NULL:

```
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 e 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. ticket313. ticket313.

> *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.*)

] ticket313.

After MPI is initialized, the application can access information about the execution environment by querying the predefined info object MPI_INFO_ENV. The following keys are predefined for this object, corresponding to the arguments of MPI_COMM_SPAWN or of mpiexec:

MPI_ENV_N Total number of MPI processes.

MPI_ENV_THREAD_LEVEL Level of thread support.

MPI_ENV_HOST Hostname.

MPI_ENV_ARCH Architecture name.

MPI_ENV_WDIR Working directory of the MPI process

MPI_ENV_FILE Value is the name of a file in which additional information is specified.

Implementations may provide additional, implementation specific, keys.

MPI_FINALIZE()

```
int MPI_Finalize(void)
```

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

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

ticket313. 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, before each process exits 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

] If an MPI program terminates normally (i.e., not due to a call to MPI_ABORT or an unrecovered error) then MPI must be finalized at each MPI process. MPI is finalized at a process by a call to MPI_FINALIZE on this process.

Before MPI is finalized at an MPI process, the process must locally complete all MPI calls and free all objects created by MPI calls on that process.

When the last process calls MPI_FINALIZE, all non-local MPI calls at each process must be matched by MPI calls at the other processes that are needed to complete the relevant operation: For each send, the matching receive has occurred, each collective operation has been invoked at all involved processes, etc.

The following examples illustrates these rules

Example 8.3 The following code is correct

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

Example 8.4 Without a 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). ticket313.

[

Example 8.5 This program is correct HEADER SKIP ENDHEADER

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

Example 8.6 This program is erroneous and its behavior is undefined: HEADER SKIP ENDHEADER

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

] ticket313.

Example 8.7 This program is erroneous: The MPI_Isend call is not guaranteed to be complete even if the send request was freed and the matching receive completed; the call MPI_Finalize cannot be used to complete the call.

```
Process 0                    Process 1
---------                    ---------
MPI_Isend();                 MPI_Recv();
MPI_Request_free();          MPI_Barrier();
MPI_Barrier();               MPI_Finalize();
MPI_Finalize();
```

ticket313.

[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 8.8 This program is correct, and after the MPI_Finalize, it is as if the buffer had been detached. HEADER SKIP ENDHEADER

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

ticket313.]

Example 8.9 This program is erroneous; the program must call MPI_Buffer_detach before the call to MPI_FINALIZE.

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

ticket313. [

Example 8.10 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.

HEADER SKIP ENDHEADER

```
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();
```

] ticket313.

Example 8.11 This program is correct. The cancel operation must succeed, since the send cannot complete normally.

```
Process 0                Process 1
---------                ---------
MPI_Isend();             MPI_Finalize();
MPI_Cancel();
MPI_Wait();
MPI_Finalize();
```

ticket313.

[

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.*)

] ticket313.

Advice to implementors. An implementation may need to delay the return from MPI_FINALIZE on a process even if all communications related to MPI calls by that process have completed; the process may still receive cancel requests for messages it has completed receiving. One possible solution is to place a barrier inside MPI_FINALIZE.

(*End of advice to implementors.*)

Once MPI_FINALIZE returns, no MPI routine (not even MPI_INIT) may be called, except for MPI_GET_VERSION, MPI_INITIALIZED, and MPI_FINALIZED. ticket313.

[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 10.5.4 on page 358. ticket310.

Advice to implementors. An implementation should check whether MPI is already finalized, before executing the finalization code.

High-quality MPI implementations will free MPI resources not freed by the user before the finalize call, when MPI_FINALIZE executes. (*End of advice to implementors.*)

The level(s) of thread support that can be provided by MPI_INIT_THREAD will depend on the implementation, and may depend on information provided by the user before the program started to execute (e.g., with arguments to `mpiexec`). If possible, the call will return provided = required. Failing this, the call will return the least supported level such that provided > required (thus providing a stronger level of support than required by the user). Finally, if the user requirement cannot be satisfied, then the call will return in provided the highest supported level.

A **thread compliant** MPI implementation will be able to return provided = MPI_THREAD_MULTIPLE. Such an implementation may always return provided = MPI_THREAD_MULTIPLE, irrespective of the value of required. [At the other extreme, an MPI library that is not thread compliant may always return provided = MPI_THREAD_SINGLE, irrespective of the value of required.]

ticket313.

An MPI library that is not thread compliant must always return provided=MPI_THREAD_SINGLE, even if MPI_INIT_THREAD is called on a multithreaded process.

A call to MPI_INIT has the same effect as a call to MPI_INIT_THREAD with a required = MPI_THREAD_SINGLE.

Vendors may provide (implementation dependent) means to specify the level(s) of thread support available when the MPI program is started, e.g., with arguments to `mpiexec`. This will affect the outcome of calls to MPI_INIT and MPI_INIT_THREAD. Suppose, for example, that an MPI program has been started so that only MPI_THREAD_MULTIPLE is available. Then MPI_INIT_THREAD will return provided = MPI_THREAD_MULTIPLE, irrespective of the value of required; a call to MPI_INIT will also initialize the MPI thread support level to MPI_THREAD_MULTIPLE. Suppose, on the other hand, that an MPI program has been started so that all four levels of thread support are available. Then, a call to MPI_INIT_THREAD will return provided = required; on the other hand, a call to MPI_INIT will initialize the MPI thread support level to MPI_THREAD_SINGLE.

Rationale. Various optimizations are possible when MPI code is executed single-threaded, or is executed on multiple threads, but not concurrently: mutual exclusion code may be omitted. Furthermore, if only one thread executes, then the MPI library can use library functions that are not thread safe, without risking conflicts with user threads. Also, the model of one communication thread, multiple computation threads fits many applications well, e.g., if the process code is a sequential Fortran/C/C++ program with MPI calls that has been parallelized by a compiler for execution on an SMP node, in a cluster of SMPs, then the process computation is multi-threaded, but MPI calls will likely execute on a single thread.

The design accommodates a static specification of the thread support level, for environments that require static binding of libraries, and for compatibility for current multi-threaded MPI codes. (*End of rationale.*)

Advice to implementors. If provided is not MPI_THREAD_SINGLE then the MPI library should not invoke C/ C++/Fortran library calls that are not thread safe, e.g., in an environment where `malloc` is not thread safe, then `malloc` should not be used by the MPI library.

Some implementors may want to use different MPI libraries for different levels of thread support. They can do so using dynamic linking and selecting which library will be linked when MPI_INIT_THREAD is invoked. If this is not possible, then optimizations for lower levels of thread support will occur only when the level of thread support required is specified at link time. (*End of advice to implementors.*)

The following function can be used to query the current level of thread support.

MPI_QUERY_THREAD(provided)

OUT	provided	provided level of thread support (integer)

```
int MPI_Query_thread(int *provided)
```

```
MPI_QUERY_THREAD(PROVIDED, IERROR)
    INTEGER PROVIDED, IERROR
```

{`int MPI::Query_thread()` *(binding deprecated, see Section 15.2)* }

The call returns in provided the current level of thread [support. This]support, which will be the value returned in provided by MPI_INIT_THREAD, if MPI was initialized by a call to MPI_INIT_THREAD(). ticket0.

MPI_IS_THREAD_MAIN(flag)

OUT	flag	true if calling thread is main thread, false otherwise (logical)

```
int MPI_Is_thread_main(int *flag)
```

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

{`bool MPI::Is_thread_main()` *(binding deprecated, see Section 15.2)* }

This function can be called by a thread to [find out whether]determine if it is the main thread (the thread that called MPI_INIT or MPI_INIT_THREAD). ticket0.

All routines listed in this section must be supported by all MPI implementations.

Rationale. MPI libraries are required to provide these calls even if they do not support threads, so that portable code that contains invocations to these functions [be able to]can link correctly. MPI_INIT continues to be supported so as to provide compatibility with current MPI codes. (*End of rationale.*) ticket0.

Advice to users. It is possible to spawn threads before MPI is initialized, but no MPI call other than [MPI_INITIALIZED] MPI_GET_VERSION, MPI_INITIALIZED, or MPI_FINALIZED should be executed by these threads, until MPI_INIT_THREAD is invoked by one thread (which, thereby, becomes the main thread). In particular, it is possible to enter the MPI execution with a multi-threaded process. ticket313.

The level of thread support provided is a global property of the MPI process that can be specified only once, when MPI is initialized on that process (or before). Portable third party libraries have to be written so as to accommodate any provided level of thread support. Otherwise, their usage will be restricted to specific level(s) of thread support. If such a library can run only with specific level(s) of thread support, e.g., only with MPI_THREAD_MULTIPLE, then MPI_QUERY_THREAD can be used to check whether the user initialized MPI to the correct level of thread support and, if not, raise an exception. (*End of advice to users.*)

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

ticket313.

MPI_ENV_N
MPI_ENV_THREAD_:LEVEL
MPI_ENV_HOST
MPI_ENV_ARCH
MPI_ENV_WDIR
MPI_ENV_FILE

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