Advice to implementors. Owing to the restrictions of the MPI_THREAD_SINGLE thread support level, implementators are discouraged from making this the default thread support level for Sessions. (*End of advice to implementors.*)

MPI_SESSION_FINALIZE(session)

IN	session	session to be finalized (handle)

C binding

```
int MPI_Session_finalize(MPI_Session *session)
```

Fortran 2008 binding

```
MPI_Session_finalize(session, ierror)
    TYPE(MPI_Session), INTENT(INOUT) :: session
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_SESSION_FINALIZE(SESSION, IERROR)
    INTEGER SESSION, IERROR
```

This routine cleans up all MPI state associated with the supplied session. Every instantiated Session must be finalized using MPI_SESSION_FINALIZE. The handle session is set to MPI_SESSION_NULL by the call.

Before an MPI process invokes MPI_SESSION_FINALIZE, the process must perform all MPI calls needed to complete its involvement in MPI communications with communicators, files, and windows associated with the supplied session: it must locally complete all MPI operations that it initiated and it must execute matching calls needed to complete MPI communications initiated by other processes. For example, if the process executed a non-blocking send, it must eventually call MPI_WAIT, MPI_TEST, MPI_REQUEST_FREE, or any derived function; if the process is the target of a send, then it must post the matching receive; if it is part of a group executing a collective operation, then it must have completed its participation in the operation.

The call to MPI_SESSION_FINALIZE does not free objects created by MPI calls; these objects are freed using MPI_XXX_FREE calls.

Advice to users. As an application can potentially initialize and finalize multiple sessions, users are advised to free MPI objects associated with a session prior to invocation of MPI_SESSION_FINALIZE to avoid leaking resources. The preferred way to free communicators associated with a session is by means of MPI_COMM_DISCONNECT. (*End of advice to users.*)

Advice to implementors. An MPI implementation should be able to implement MPI_SESSION_FINALIZE as a *local* function provided an application frees all MPI windows, closes all MPI files, and uses MPI_COMM_DISCONNECT to free all MPI communicators associated with a session prior to invoking MPI_SESSION_FINALIZE on the corresponding session handle. (*End of advice to implementors.*)

An MPI implementation may need to take steps as part of MPI_SESSION_FINALIZE to insure any pending communication on communicators that were associated with the

session has been completed prior to return from the function. Consequently, MPI_SESSION_FINALIZE is a *non-local* function if communicators associated with session were not freed using MPI_COMM_DISCONNECT prior to finalizing the session. In cases where the association of communicators to sessions is not uniform across MPI processes in an MPI job, it is possible for the sequence of calls to MPI_SESSION_FINALIZE in one process to match with different sequences of calls to MPI_SESSION_FINALIZE in other processes. The semantics of MPI_SESSION_FINALIZE is what would be obtained if the callers initiated a series of MPI_IALLTOALL calls over all communicators still associated with the session, followed by a call to MPI_WAITALL. The sequence of calls to MPI_SESSION_FINALIZE by each process must ensure this type of communication pattern would complete at each process to avoid deadlock. Note this requirement places restrictions on the way communicators, windows, and files can be associated with sessions in the event the application does not free MPI objects prior to invoking MPI_SESSION_FINALIZE.

The following pseudo-code snippets illustrate several correct and incorrect sequences of calls to MPI_SESSION_FINALIZE. To keep the examples compact, full function names are abbreviated as indicated in the following table.

MPI Function	abbreviation
MPI_SESSION_INIT	s_init
MPI_GROUP_FROM_SESSION_PSET	g_from_s
MPI_COMM_CREATE_FROM_GROUP	c_from_g
MPI_SESSION_FINALIZE	s_fin
MPI_COMM_DISCONNECT	c_dis

Table 11.1: List of abbreviated function names used in the examples below

Example 11.8 The following code is correct

```
Process 0                              Process 1
---------                              ---------
s_init(&s0a)                           s_init(&s1a)
g_from_s(s0a, "mpi://WORLD", &g0a)     g_from_s(s1a, "mpi://WORLD", &g1a)
c_from_g(g0a, "foobar", &cx)           c_from_g(g1a,"foobar", &cx)
                                       s_init(&s1b)
                                       g_from_s(s1b, "mpi://WORLD", &g1b)
c_from_g(g0a,"foobar2", &cy)           c_from_g(g1b,"foobar2", &cy)
      .                                      .
      .                                      .
      .                                      .
s_fin(&s0a)                            s_fin(&s1a)
                                       s_fin(&s1b)
```

In this example, communicators *cx* and *cy* are associated with session handle *s0a* in process 0 and with *s1a* and *s1b* in process 1. This code is correct because the sequence of two calls to MPI_SESSION_FINALIZE in process 1 matches with the single call to the function in process 0. The order of calls to MPI_SESSION_FINALIZE in process 1 could be reversed and the code would remain correct.

Example 11.9 The following code is incorrect

```
Process 0                              Process 1
---------                              ---------
s_init(&s0a)                           s_init(&s1a)
s_init(&s0b)                           s_init(&s1b)
g_from_s(s0a, "mpi://WORLD", &g0a)     g_from_s(s1a, "mpi://WORLD", &g1a)
g_from_s(s0b, "mpi://WORLD", &g0b)     g_from_s(s1b, "mpi://WORLD", &g1b)
c_from_g(g0a, "foobar", &cx)           c_from_g(g1a, "foobar", &cx)
c_from_g(g0a, "foobar2", &cy)          c_from_g(g1b, "foobar2", &cy)
c_from_g(g0b, "foobar3", &cz)          c_from_g(g1a, "foobar2", &cz)
      .                                      .
      .                                      .
      .                                      .
s_fin(&s0a)                            s_fin(&s1a)
s_fin(&s0b)                            s_fin(&s1b)
```

This example is incorrect because no sequence of calls to MPI_SESSION_FINALIZE can be made from either process that will not lead to deadlock owing to the way communicators were associated with sessions.

Example 11.10 The following code is correct.

```
Process 0                              Process 1
---------                              ---------
s_init(&s0a)                           s_init(&s1a)
s_init(&s0b)                           s_init(&s1b)
g_from_s(s0a, "mpi://WORLD", &g0a)     g_from_s(s1a, "mpi://WORLD", &g1a)
g_from_s(s0b, "mpi://WORLD", &g0b)     g_from_s(s1b, "mpi://WORLD", &g1b)
c_from_g(g0a, "foobar", &cx)           c_from_g(g1a, "foobar", &cx)
c_from_g(g0a, "foobar2", &cy)          c_from_g(g1b, "foobar2", &cy)
c_from_g(g0b, "foobar3", &cz)          c_from_g(g1a, "foobar2", &cz)
      .                                      .
      .                                      .
      .                                      .
c_dis(&cx)                             c_dis(&cx)
c_dis(&cy)                             c_dis(&cy)
c_dis(&cz)                             c_dis(&cz)
s_fin(&s0a)                            s_fin(&s1a)
s_fin(&s0b)                            s_fin(&s1b)
```

This example is correct because the application invokes MPI_COMM_DISCONNECT on the communicators associated with sessions before invoking MPI_SESSION_FINALIZE.

11.3.2 Processes Sets

Process sets are the mechanism for MPI applications to query the runtime. Process sets are identified by process set names. Process set names have a *Uniform Resource Identifier* (URI) format. Two process set names are mandated: "mpi://WORLD" and