MPI_WTIME()

```
double MPI_Wtime(void)
```

```
DOUBLE PRECISION MPI_WTIME()
```

{`double MPI::Wtime()` *(binding deprecated, see Section 15.2)* }

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 15.2)* }

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}.

ticket277.

Timer requests allow applications to control how much time they spend in MPI_WAITANY and related functions. Timer requests complete after a specified time interval.

MPI_TIMER_CREATE(due_time, request)

IN	due_time	number of seconds until timer expires (double)
OUT	request	timer request (handle)

```
int MPI_Timer_create(double due_time, MPI_Request *request)
```

```
MPI_TIMER_CREATE(DUE_TIME, REQUEST, IERROR)
    DOUBLE PRECISION DUE_TIME
    INTEGER REQUEST, IERROR
```

Creates a timer request that will complete after the specified due time. The due_time argument is relative to MPI_WTIME. For example, a call to MPI_TIMER_CREATE at time A = MPI_WTIME will result in the timer request completing when MPI_WTIME returns A + 5 or greater.

Similar to other MPI requests, a timer request can be cancelled by MPI_CANCEL and may be freed by MPI_REQUEST_FREE.

Specifying a due_time less than or equal to zero causes the timer to be created in the completed state, such that a call to MPI_WAIT returns immediately. A positive due_time causes the timer request to be active such that it must be completed via the usual request completion functions, such as MPI_TEST or MPI_WAIT.

Advice to users. MPI only guarantees that timer requests will not complete before the due time arrives. MPI does not guarantee that timer requests will complete exactly when the timer's due time arrives. (*End of advice to users.*)

MPI_TIMER_RESET(due_time, request)

IN	due_time	number of seconds until timer expires (double)
INOUT	request	timer request (handle)

```
int MPI_Timer_reset(double due_time, MPI_Request *request)
```

```
MPI_TIMER_RESET(DUE_TIME, REQUEST, IERROR)
    DOUBLE PRECISION DUE_TIME
    INTEGER REQUEST, IERROR
```

Resets the due time of an active timer request.

Advice to users. This function is provided as a convenience for cases where applications need to reset a timer that has not yet completed (regardless of whether its due_time has arrived or not). It saves the step of canceling an active timer and creating a new one. (*End of advice to users.*)

Advice to implementors. Similar to MPI_START, the request argument is INOUT to allow implementations to return a new request if necessary. (*End of advice to implementors.*)

8.7 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