IBM Parallel Environment for AIX
MPL Programming and Subroutine Reference
Version 2, Release 1

Document Number GC23-3893-00

Note

Before using this information and the product is supports, be sure to read the general information under “Notices” on page iv

First Edition (August 1995)

This edition applies to the IBM Parallel Environment for AIX program product, Version 2 Release 1 and to all subsequent releases and modifications until otherwise indicated in new editions or Technical Newsletters.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

IBM welcomes your comments. A form for your comments appears at the back of this publication. If the form has been removed, address your comments to:

IBM Corporation, Department 55JA, Mail Station P384
522 South Road
Poughkeepsie, NY
12601-5400
United States of America

FAX: (International Access Code)+1+914+432-9405

IBMLink (United States customers only): KGNRCF at KGNVMC
IBM Mail Exchange: USIB27BQ at IBMMAIL
Internet: kgnrcf@vnet.ibm.com

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Copyright International Business Machines Corporation 1995. All rights reserved. Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corporation.

Contents

Notices iv
Trademarks iv

About This Book v
Who Should Use This Book v
How This Book Is Organized v
Overview of Contents v
Document Conventions vi
How FORTRAN And C Are Documented vi
Related Publications vii
Parallel Environment for AIX Publications vii
Ordering Additional Copies Of This Book vii
Accessing Online Help Information vii

Chapter 1. Subroutines Listed Alphabetically 1

Appendix A. Subroutine Argument Quick Reference 109

Appendix B. Subroutine Argument Definitions 111

Appendix C. MPL Size Limits 113

Appendix D. Reduction Operations 115

Appendix E. Parallel Utility Functions 117

Appendix F. Tracing Routines 131

Appendix G. Programming Considerations for User Applications in POE 149

Appendix H. Using Signals and Parallel Environment Programs 159

© Copyright IBM Corp. 1995

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY 10594
USA

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P131
522 South Road
Poughkeepsie, NY 12601-5400
USA
Attention: Information Request

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

AIX
IBM
RISC System/6000
Scalable POWERparallel Systems

X-Windows System is a trademark of the Massachusetts Insitute of Technology

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

PostScript is a trademark of Adobe Systems, Incorporated

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

About This Book

This book lists the subroutines a programmer can use when writing parallel applications along with the associated parameters, syntax, and examples. In addition, this book provides brief introductory information regarding parallel programming tasks.

Who Should Use This Book

IMPORTANT NOTE

If you are a new user and do not have any Message Passing Library(MPL) routines, use the *IBM Parallel Environment for AIX: MPI Programming and Subroutine Reference* to write Message-Passing Interface(MPI) routines.

If you have existing MPL routines, use this manual.

This book is intended for experienced programmers who want to write parallel applications using either the C or FORTRAN programming languages. Readers of this book should know C and FORTRAN and should be familiar with AIX and UNIX commands, file formats, and special files. In addition, readers should be familiar with distributed-memory machines.

How This Book Is Organized

Overview of Contents

This book is divided into the following sections:

- Chapter 1, "Subroutines Listed Alphabetically" on page 1 lists the subroutines alphabetically along with their descriptions, type and page number.
- Appendix A, "Subroutine Argument Quick Reference" on page 109 lists the subroutines and their arguments. Use it as a quick reference. For detailed information on the subroutines refer to Chapter 1, "Subroutines Listed Alphabetically" on page 1.
- Appendix B, "Subroutine Argument Definitions" on page 111 briefly lists the arguments and their definitions. For detailed definitions of the argument definitions, refer to Chapter 1, "Subroutines Listed Alphabetically" on page 1.
- Appendix C, "MPL Size Limits" on page 113 gives information about the MPL size limits.
- Appendix D, "Reduction Operations" on page 115 gives additional information about reduction functions.
- Appendix E, "Parallel Utility Functions" on page 117 contains the syntax man pages of the user-callable functions that take advantage of the Parallel Operating Environment(POE).
- Appendix F, "Tracing Routines" on page 131 contains the syntax man pages for modifying trace generation for the visualization tool.

© Copyright IBM Corp. 1995

- Appendix G, “Programming Considerations for User Applications in POE” on page 149 contains various information for user applications written to run under the Parallel Environment.
- Appendix H, “Using Signals and Parallel Environment Programs” on page 159 explains timer signals and includes sample programs.

Document Conventions

This manual uses visual cues to help you locate and identify information quickly and easily. The typographic conventions used in this book are outlined in the following table.

Typographic	Usage
Bold	• **Bold** words or characters represent system elements that you must use literally, such as commands, flags, and path names. • **Bold** words also indicate the first use of a term included in the glossary.
Italic	• *Italic* words or characters represent variable values that you must supply. • *Italics* are also used for book titles and for general emphasis in text.
`Constant width`	Examples and information that the system displays appear in `constant width` typeface.
[ ]	Brackets enclose optional items in format and syntax descriptions.
{ }	Braces enclose a list from which you must choose an item in format and syntax descriptions.
|	A vertical bar separates items in a list of choices. (In other words, it means “or.”)
< >	Angle brackets (less-than and greater-than) enclose the name of a key on the keyboard. For example, <**Enter**> refers to the key on your terminal or workstation that is labeled with the word Enter.
...	An ellipsis indicates that you can repeat the preceding item one or more times.
<**Ctrl-***x*>	The notation <**Ctrl-***x*> indicates a control character sequence. For example, <**Ctrl-c**> means that you hold down the control key while pressing <**c**>.

How FORTRAN And C Are Documented

FORTRAN is not case sensitive while the reverse is true for C. This means that you can enter the FORTRAN subroutines in either upper or lower case or in mixed case. However, if you use the "MIXED" FORTRAN compiler option, subroutines must be entered in lower case. The C subroutines must be entered in lower case only.

For the purpose of distinguishing between the FORTRAN and C syntax in this document, FORTRAN subroutines are documented in all upper case while the C subroutines appear in all lower case.

Related Publications

Parallel Environment for AIX Publications

- *IBM Parallel Environment for AIX: General Information*, GC23-3906
- *IBM Parallel Environment for AIX: Hitchhiker's Guide*, GC23-3895-00
- *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*, GC23-3892-00
- *IBM Parallel Environment for AIX: Operation and Use*, GC23-3891-00
- *IBM Parallel Environment for AIX: MPL Programming and Subroutine Reference*, GC23-3893-00
- *IBM Parallel Environment for AIX: MPI Programming and Subroutine Reference*, GC23-3894-00
- *IBM Parallel Environment for AIX: Licensed Program Specification*, GC23-3896-00

As an alternative to ordering the individual books, you can use GBOF-8588 to order the entire Parallel Environment for AIX library.

Ordering Additional Copies Of This Book

To order additional copies of this book, use order number GC23-3893.

To order the full set of Parallel Environment for AIX books, use order number GBOF-8588.

Accessing Online Help Information

The PE program product provides manual pages as well as online help information in InfoExplorer format. InfoExplorer is the AIX online information facility. You can use InfoExplorer's extensive search capabilities to find help on any part of the PE product. When you press **Help** on one of the product's X-Window's applications, the appropriate book in the Parallel Environment library will appear on the screen in InfoExplorer format. In particular, the X-Windows applications display error messages in their own X-Windows. Pressing **Help** in one of these windows invokes InfoExplorer and opens the appropriate book to the appropriate spot for error resolution.

You can also access the books in InfoExplorer format without using the graphical user interface. To access this book or any other book in this library in InfoExplorer format:

ENTER **info -l (lowercase L) pe**

To access a list of manual pages for IBM Parallel Environment for AIX:

ENTER **man pe**

Note: In order to use the PE manual pages or InfoExplorer, the *ppe.pedocs* file set must first be installed. See *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis* for more information.

Chapter 1. Subroutines Listed Alphabetically

The following lists the subroutines alphabetically. Refer to the appropriate page number for information related to subroutine purpose, syntax, and so on.

FORTRAN/C Subroutine	Page Number	Type	Description
MP_BCAST mpc_bcast	6	Collective Communication	Distributes a message from one task to all the tasks in the same group.
MP_BRECV mpc_brecv	9	Message Passing	Receives a message and waits until the requested data arrives.
MP_BSEND mpc_bsend	12	Message Passing	Sends a message and waits until the output buffer can be reused.
MP_BSENDRECV mpc_bsendrecv	15	Message Passing	Sends a message and waits until a message of the same type has been received.
MP_BVRECV mpc_bvrecv	19	Message Passing	Receives a non-contiguous message and waits until the message is available.
MP_BVSEND mpc_bvsend	22	Message Passing	Send a non-contiguous message and waits until the message buffer can be reused.
MP_COMBINE mpc_combine	25	Collective Communication	Performs a reduction operation on data distributed across a group and distributes the result in all tasks in the group.
MP_CONCAT mpc_concat	28	Collective Communication	Concatenates a distinct message from each task to create one message.
MP_ENVIRON mpc_environ	31	Task Management	Returns the number of tasks assigned to the partition and the caller's task id.
MP_GATHER mpc_gather	33	Collective Communication	Concatenates a distinct message from each task to create a single message on task **dest**.
MP_GETLABEL mpc_getlabel	36	Task Group	Returns the label assigned to a group.
MP_GETMEMBERS mpc_getmembers	38	Task Group	Gets the membership array of a specified task group.
MP_GETRANK mpc_getrank	40	Task Group	Checks membership and gets the rank of a given task within a task group.
MP_GETSIZE mpc_getsize	43	Task Group	Gets the size of a specified task group.
MP_GETTASKID mpc_gettaskid	45	Task Group	Returns the task id for a **rank** within a group.
MP_GROUP mpc_group	48	Task Group	Allows you to define a task group by explicitly specifying the tasks that will form the group.
MP_INDEX mpc_index	51	Collective Communication	Each task sends a distinct message to all the tasks in the same group.

© Copyright IBM Corp. 1995

FORTRAN/C Subroutine	Page Number	Type	Description
MP_LOCKRNC mpc_lockrnc	54	Task Management	Asserts or frees a lock which can block MP_RCVNCALL handlers and returns the old lock value.
MP_PACK mpc_pack	56	Utilities	Packs data from a non-contiguous buffer into a contiguous buffer.
MP_PARTITION mpc_partition	59	Task Group	Creates a task group by partitioning a previously defined task group.
MP_PREFIX mpc_prefix	61	Collective Communication	Applies a parallel prefix operation with respect to a reduction operation on data distributed across a group and places the result in each task in the group.
MP_PROBE mpc_probe	64	Message Passing	Checks for any messages which may have arrived without matching a pending receive.
MP_RCVNCALL mpc_rcvncall	67	Message Passing	Posts a receive buffer and a handler function for a message. Returns without waiting for the message arrival.
MP_RECV mpc_recv	74	Message Passing	Posts a receive buffer for a message and returns without waiting for the message arrival.
MP_REDUCE mpc_reduce	77	Collective Communication	Performs a reduction operation on data distributed across a group.
MP_SCATTER mpc_scatter	80	Collective Communication	Distributes distinct messages to each task in the group.
MP_SEND mpc_send	83	Message Passing	Identifies a message to be sent and returns without waiting for the send to complete.
MP_SHIFT mpc_shift	86	Collective Communication	Performs linear or circular data shifts across a group.
MP_STATUS mpc_status	89	Message Passing	Returns the status of a non-blocking send or receive.
MP_STOPALL mpc_stopall	92	Task Management	Forces an abnormal termination.
MP_SYNC mpc_sync	94	Collective Communication	Synchronizes tasks at a barrier.
MP_TASK_QUERY mpc_task_query	96	Task Management	Returns information about system variables and constants.
MP_TASK_SET mpc_task_set	99	Task Management	Sets user-definable parameters for the task.
MP_UNPACK mpc_unpack	103	Utilities	Unpacks data from a contiguous buffer into a non-contiguous buffer.
MP_WAIT mpc_wait	106	Message Passing	Waits until a non-blocking send or receive has completed.

The following pages list the subroutines available for parallel programming. For each subroutine, a purpose, C and FORTRAN synopsis, description, notes, return values, error conditions, and examples are provided. Review the following sample subroutine before proceeding to better understand how the subroutines are structured.

NAME

MP_SAMPLE, mpc_sample – The text presented in this section briefly describes the use of the subroutine.

LIBRARY

libmpi.a

The mpcc, mpCC and mpxlf scripts automatically include the libraries identified in this manual.

C SYNOPSIS

- Header file mpproto.h supplies both ANSI-C and C++ prototypes for every function described in the message passing subroutine section of this manual.

```
#include <mpproto.h>
int mpc_sample (parameter(s));
```

In the C prototype:

- A declaration of void * indicates that a pointer to any data type is allowable.
- If ANSI-C type checking is not going to be enabled, then any variable declared as size_t can be declared as int.

FORTRAN SYNOPSIS

```
MP_SAMPLE (parameter(s))
```

In the FORTRAN routines, formal parameters are described using a subroutine prototype format, even though FORTRAN does not support prototyping.

PARAMETERS

Variable or parameter definitions appear below:

parameter1 parameter description.

parameter2 parameter description.

DESCRIPTION

This section contains a more detailed description of the subroutine or function.

NOTES

If applicable, this section contains notes about standards, among other things.

RETURN VALUES

Return values appear here. In almost every routine, the C version is invoked as a function returning integer. In the FORTRAN version, the routine is CALLed as a subroutine; that is, it has no return value.

ERRORS

Error conditions appear here.

EXAMPLES

C Example

A C example appears here.

FORTRAN Example

A FORTRAN example appears here.

RELATED INFORMATION

This section contains a list of related functions or routines in this book among other things.

NAME

MP_BCAST, mpc_bcast – Distributes a message throughout a group.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mpproto.h>
int mpc_bcast(void *msg, size_t msglen, int source, int gid);
```

FORTRAN SYNOPSIS

```
MP_BCAST(INTEGER MSG(*), INTEGER MSGLEN, INTEGER SOURCE, INTEGER GID)
```

PARAMETERS

msg	for task id=**source**, **msg** is the application buffer that contains the message to send. For all other tasks in the same group, **msg** is the application buffer into which the incoming message is received.
msglen	is a non-negative integer that specifies the size, in bytes, of the message.
source	is an integer that specifies the task that originates the send.
gid	is a group identifier for the group of participating tasks.

DESCRIPTION

This routine sends a message from the task **source** to all other tasks in the group which is uniquely identified by **gid**. The message is sent from **msg** in the task with **task id**=**source** to **msg** in all other tasks, and is of length **msglen** bytes. All tasks which are members of **gid** must call **MP_BCAST** and specify consistent parameters.

RETURN VALUES

In C, return code = 0 means the operation was successful. Return code = -1 means that an error occurred and the global variable **mperrno** contains the error number. This number, “nnn,” can be used to find a description of the error condition by looking for **0032-nnn** in *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

ERRORS

msg	is not a valid buffer
msglen	is less than zero
source	is not a valid task id or is not a member of **gid**
gid	is undefined in the calling task

Develop mode error if:

- all tasks in **gid** do not use identical values for **source** and **msglen**

EXAMPLES

A message is sent from the task with the highest task id in the partition to all other tasks using MP_BCAST.

C Example

```
   #include <mpproto.h>

int bcast_example()
{
   int numtask,taskid,source,msg,msglen,rc;

   mpc_environ(&numtask,&taskid);

   /* Let the "last" task be the source (broadcaster).  */
   source = numtask - 1;
   msglen = sizeof(int);
   if(taskid == source)  msg = 5;      /* initialize data to broadcast  */

   rc = mpc_bcast(&msg,msglen,source,ALLGRP);
   if(rc)  return(-1);  /* error */

   /* All tasks now have "msg" set to "5"  */

   return(0);
}
```

FORTRAN Example

```
      SUBROUTINE BCAST_EXAMPLE

      INTEGER NUMTASK,TASKID,SOURCE,MSG,MSGLEN,NBUF(4),ALLGRP

      CALL MP_ENVIRON(NUMTASK,TASKID)

      CALL MP_TASK_QUERY(NBUF,4,3)       !  get "ALLGRP" value
      ALLGRP = NBUF(4)

c     Let the "last" task be the source (broadcaster).
      SOURCE = NUMTASK - 1
      MSGLEN = 4
      IF(TASKID .EQ. SOURCE)  MSG = 5    ! initialize data to broadcast

      CALL MP_BCAST(MSG,MSGLEN,SOURCE,ALLGRP)

c     All tasks now have "MSG" set to "5"

      END
```

NAME

MP_BRECV, mpc_brecv – Receives a message and waits until the requested data arrives.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mpproto.h>
int mpc_brecv(void *inmsg, size_t msglen, int *source, int *type,
              size_t *nbytes);
```

FORTRAN SYNOPSIS

```
MP_BRECV(INTEGER INMSG(*), INTEGER MSGLEN, INTEGER SOURCE,
         INTEGER TYPE, INTEGER NBYTES)
```

PARAMETERS

inmsg is the application buffer into which the message is to be received.

msglen is an integer that specifies the size, in bytes, of the buffer **inmsg**.

source is an integer that specifies the task from which the message originated. If **source**=**dontcare**, the next message satisfying the **type** parameter is read, and the sending task is returned in **source**.

type is an integer that specifies the type of the message being received. If **type**=**dontcare**, the next message satisfying the **source** parameter is read, and the message type is returned in **type**. If both **source**=**dontcare** and **typedontcare**, the next message is read, and the source task and type of that incoming message are returned in **source** and **type**, respectively.

nbytes is an integer which returns the number of bytes received. If the message truncation mode is enabled and the length of the message is greater than **msglen**, **nbytes** is returned as equal to the **msglen** parameter.

DESCRIPTION

This routine receives a message into the application buffer and waits until the requested data is available in the application buffer in the receiving task.

RETURN VALUES

In C, return code = 0 means the operation was successful. Return code = -1 means that an error occurred and the global variable **mperrno** contains the error number. This number, “nnn,” can be used to find a description of the error condition by looking for **0032-nnn** in *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

ERRORS

inmsg is not a valid buffer

msglen is less than zero

source is not a valid task id or wildcard value

type is not a valid message type or wildcard value

It is an error if the message truncation mode is not enabled and the length of the message to be received is greater than **msglen**. For information on enabling the message truncation mode, see MP_TASK_SET.

EXAMPLES

A message from task 0 is received by task 1 using MP_BRECV. Immediately after the call to MP_BRECV completes, the received data is available in **inmsg**.

C Example

```
   #include <mpproto.h>

int brecv_example()
{
   int source,dest,type,msglen,nbytes,rc;
   int numtask,taskid,inmsg[2],outmsg[2];

   mpc_environ(&numtask,&taskid);

   type = 1;
   source = 0;
   dest = 1;
   msglen = sizeof(outmsg);

   if(taskid == source)  {
      outmsg[0] = 150;
      outmsg[1] = 250;

      rc = mpc_bsend(outmsg,msglen,dest,type);
      if(rc)  return(-1);  /* error */
   } else {
      if(taskid == dest) {
         rc = mpc_brecv(inmsg,msglen,&source,&type,&nbytes);
         if(rc)  return(-1);  /* error */

         /* Task 1 now has "inmsg" = {150,250}  */
      }
   }
   return(0);
}
```

FORTRAN Example

```
      SUBROUTINE BRECV_EXAMPLE

      INTEGER SOURCE,DEST,TYPE,NUMTASK,TASKID,INMSG(2),OUTMSG(2)

      CALL MP_ENVIRON(NUMTASK,TASKID)

      TYPE = 1
      SOURCE = 0
      DEST = 1
      MSGLEN = 8

      IF(TASKID .EQ. SOURCE)  THEN
         OUTMSG(1) = 150
         OUTMSG(2) = 250
         CALL MP_BSEND(OUTMSG,MSGLEN,DEST,TYPE)
      ELSE
         IF(TASKID .EQ. DEST)  THEN

            CALL MP_BRECV(INMSG,MSGLEN,SOURCE,TYPE,NBYTES)

c           Task 1 now has "INMSG" = {150,250}
c           "NBYTES" returns the actual number of bytes sent.

         ENDIF
      ENDIF

      END
```

RELATED INFORMATION

Subroutines: MP_BSEND(3), mpc_bsend(3), MP_RECV(3), mpc_recv(3)

NAME

MP_BSEND, mpc_bsend – Sends a message and waits until the output buffer can be reused.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mpproto.h>
int mpc_bsend(const void *outmsg, size_t msglen, int dest, int type);
```

FORTRAN SYNOPSIS

```
MP_BSEND(INTEGER OUTMSG(*), INTEGER MSGLEN, INTEGER DEST, INTEGER TYPE)
```

PARAMETERS

outmsg	is the application buffer that contains the message to be sent.
msglen	is a non-negative integer that specifies the size, in bytes, of the message to be sent.
dest	is an integer that specifies the task to which the message is to be sent.
type	is an integer that specifies the type of the message being sent.

DESCRIPTION

This subroutine sends a message and waits until the application buffer **outmsg** is available for reuse. Completion does not imply that the message has been received by the destination task.

RETURN VALUES

In C, return code = 0 means the operation was successful. Return code = -1 means that an error occurred and the global variable **mperrno** contains the error number. This number, “nnn,” can be used to find a description of the error condition by looking for **0032-nnn** in *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

ERRORS

outmsg	is not a valid buffer
msglen	is less than zero
dest	is not a valid task id
type	is not a valid message type

EXAMPLES

A message is sent from task 0 to task 1 using MP_BSEND. Immediately after the call to MP_BSEND completes, the send buffer **outmsg** is ready to be reused.

C Example

```
   #include <mpproto.h>

int bsend_example()
{
   int source,dest,type,msglen,nbytes,rc;
   int numtask,taskid,inmsg[2],outmsg[2];

   mpc_environ(&numtask,&taskid);

   type = 1;
   source = 0;
   dest = 1;
   msglen = sizeof(outmsg);

   if(taskid == source)  {
      outmsg[0] = 150;
      outmsg[1] = 250;

      rc = mpc_bsend(outmsg,msglen,dest,type);
      if(rc)  return(-1);  /* error */

      /* "outmsg" can now be reused  */
   } else {
      if(taskid == dest) {
         rc = mpc_brecv(inmsg,msglen,&source,&type,&nbytes);
         if(rc)  return(-1);  /* error */

         /* Task 1 now has "inmsg" = {150,250}  */
      }
   }
   return(0);
}
```

FORTRAN Example

```
      SUBROUTINE BSEND_EXAMPLE

      INTEGER SOURCE,DEST,TYPE,NUMTASK,TASKID,INMSG(2),OUTMSG(2)

      CALL MP_ENVIRON(NUMTASK,TASKID)

      TYPE = 1
      SOURCE = 0
      DEST = 1
      MSGLEN = 8

      IF(TASKID .EQ. SOURCE)  THEN
         OUTMSG(1) = 150
         OUTMSG(2) = 250

         CALL MP_BSEND(OUTMSG,MSGLEN,DEST,TYPE)

c        "OUTMSG" can now be reused
      ELSE
         IF(TASKID .EQ. DEST)  THEN
            CALL MP_BRECV(INMSG,MSGLEN,SOURCE,TYPE,NBYTES)

c           Task 1 now has "INMSG" = {150,250}
c           "NBYTES" returns the actual number of bytes sent.

         ENDIF
      ENDIF

      END
```

RELATED INFORMATION

Subroutines: MP_BRECV(3), mpc_brecv(3), MP_SEND(3), mpc_send(3)

NAME

MP_BSENDRECV, mpc_bsendrecv – Sends a message and waits until a message of the same type is received.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mpproto.h>
int mpc_bsendrecv(const void *outmsg, size_t outmsglen,int dest,
                  int type,void *inmsg, size_t inmsglen,int *source,
                  size_t *nbytes);
```

FORTRAN SYNOPSIS

```
MP_BSENDRECV(INTEGER OUTMSG(*), INTEGER OUTMSGLEN, INTEGER DEST,
             INTEGER TYPE, INTEGER INMSG(*), INTEGER INMSGLEN,
             INTEGER SOURCE, INTEGER NBYTES)
```

PARAMETERS

outmsg is the application buffer that contains the message to be sent.

outmsglen is a non-negative integer that specifies the size, in bytes, of the message to be sent.

dest is an integer that specifies the task to which the message is to be sent. If **dest**=**nulltask**, no message is sent.

type is an integer that specifies the type of the message being sent and the type of the message to be received. **type** cannot be set equal to **dontcare**.

inmsg is the application buffer into which the message is received. The receiving application buffer, **inmsg**, and the sending application buffer, **outmsg**, cannot overlap in memory.

inmsglen is a non-negative integer that specifies the size, in bytes, of the buffer **inmsg**.

source is an integer that specifies the task from which the message originated. If **source**=**dontcare**, the next message satisfying the **type** parameter is read, and the sending task is returned in **source**. If **source**=**nulltask**, no message is received.

nbytes is an integer which returns the number of bytes received. If the message truncation mode is enabled and the length of the message is greater than **inmsglen**, **nbytes** is returned as equal to the **inmsglen** parameter. If **source**=**nulltask**, **nbytes** returns 0.

DESCRIPTION

This routine sends a message to **dest** and waits until a message of the same type is received from **source**. If two tasks each name the other as both **source** and **dest**, then there is an exchange of messages of the same type between the two tasks. However, by making **source** and **dest** distinct, you can use MP_BSENDRECV or mpc_bsendrecv to create different topologies such as a ring structure. In this case, each task receives a message from **source** and sends a message to **dest**. MP_BSENDRECV waits until the **outmsg** buffer is available for reuse and the **inmsg** buffer contains the completed message.

Programs must use MP_BSENDRECV to pass messages in conjunction only with other MP_BSENDRECVs. This means that you cannot mix MP_BSENDRECV on one task with ordinary sends and receives on other tasks. Such programs may work on some implementations but they will be implementation-dependent and cannot be guaranteed to be portable or upward-compatible.

Either **source** or **dest** can take **nulltask**, as an argument. Using **nulltask** means that the corresponding part of the send/receive operation will be inoperative. This is useful in ring operations where in shifting data “up,” for example, one may not wish task 0 to receive a message from the last task.

RETURN VALUES

In C, return code = 0 means the operation was successful. Return code = -1 means that an error occurred and the global variable **mperrno** contains the error number. This number, “nnn,” can be used to find a description of the error condition by looking for **0032-nnn** in *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

ERRORS

outmsg *and* **inmsg** overlap in memory

outmsg *or* **inmsg** is not a valid buffer

outmsglen *or* **inmsglen** is less than zero

dest *or* **source** is not a valid task id or wildcard value

type is not a valid message type

To use MP_BSENDRECV in conjunction with the other message passing commands such as MP_SEND, MP_BSEND, MP_RECV, and MP_BRECV is an error in semantics. Programming this way may not be portable between implementations.

Undetectable and implementation-dependent error to send a message to another task that has **source**=**nulltask** or to attempt to receive a message from a task that has **dest**=**nulltask**.

EXAMPLES

In a partition with an even number of tasks, pairs of tasks exchange messages with a single call using MP_BSENDRECV.

C Example

```
   #include <mpproto.h>

int bsendrecv_example()
{
   int source,dest,type,msglen,nbytes,rc;
   int numtask,taskid,inmsg,outmsg;

   mpc_environ(&numtask,&taskid);
   if(numtask % 2)  return(-1);
   /* "numtask" must be even in this example  */

   if(taskid % 2)   source = taskid - 1;         /* partner to "left" */
   else             source = taskid + 1;         /* partner to "right" */

   dest   = source;
   outmsg = taskid;
   msglen = sizeof(int);
   type = 1;

   rc = mpc_bsendrecv(&outmsg,msglen,dest,type,&inmsg,msglen,
                      &source, &nbytes);
   if(rc)  return(-1);  /* error */

   /*  Each task now has "inmsg" set to the task id of its partner  */

   return(0);
}
```

FORTRAN Example

```
      SUBROUTINE BSENDRECV_EXAMPLE

      INTEGER SOURCE,DEST,TYPE,MSGLEN,NBYTES
      INTEGER NUMTASK,TASKID,INMSG,OUTMSG

      CALL MP_ENVIRON(NUMTASK,TASKID)
c     "NUMTASK" must be even in this example
      IF(MOD(NUMTASK,2) .EQ. 1)  STOP 'NUMTASK MUST BE EVEN'

      IF(MOD(TASKID,2) .EQ. 1)  THEN
         SOURCE = TASKID - 1
      ELSE
         SOURCE = TASKID + 1
      ENDIF

      DEST   = SOURCE
      OUTMSG = TASKID
      MSGLEN = 4
      TYPE = 1

      CALL MP_BSENDRECV(OUTMSG,MSGLEN,DEST,TYPE,INMSG,MSGLEN,
     +                  SOURCE,NBYTES)

c     Each task now has "INMSG" set to the task id of its partner
c     "NBYTES" returns the actual number of bytes sent.

      END
```

RELATED INFORMATION

Subroutines: MP_SEND(3), mpc_send(3), MP_RECV(3), mpc_recv(3)

NAME

MP_BVRECV, mpc_bvrecv – Receives a non-contiguous message and waits until message is available

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mpproto.h>
int mpc_bvrecv(void *inmsg, size_t blklen, size_t offset,
               int blknum, int *source, int *type, size_t *nbytes);
```

FORTRAN SYNOPSIS

```
MP_BVRECV(INTEGER INMSG(*), INTEGER BLKLEN, INTEGER OFFSET, INTEGER BLKNUM,
         INTEGER SOURCE, INTEGER TYPE, INTEGER NBYTES)
```

PARAMETERS

inmsg is the starting location of the buffer into which the message is to be received. **inmsg** consists of **blknum** blocks, with each block consisting of **blklen** contiguous bytes. The starting addresses of consecutive blocks are separated by **offset** bytes.

blklen is a non-negative integer that specifies the size, in bytes, of each block contained in buffer **inmsg**.

offset is an integer that specifies the separation, in bytes, between the starting addresses of consecutive blocks. **offset** must be greater than or equal to **blklen**.

blknum is an integer that specifies the number of blocks contained in the buffer **inmsg**.

source is an integer that specifies the task from which the message has originated. If **source**=**dontcare**, the next message satisfying the **type** parameter will be read, and the sending task will be returned in source.

type is an integer that specifies the type of the message being received. If **type**=**dontcare**, the next message satisfying the **source** parameter will be read, and the message type will be returned in **type**. If both **source**=**dontcare** and **type**=**dontcare**, the next message will be read, and the source task and type returned in **source** and **type**, respectively.

nbytes is an integer that specifies the number of bytes received.

DESCRIPTION

This subroutine receives a message into non-contiguous memory and waits until the requested data is available in the application buffer in the receiving task.

RETURN VALUES

In C, return code = 0 means the operation was successful. Return code = -1 means that an error occurred and the global variable **mperrno** contains the error number. This number, “nnn,” can be used to find a description of the error condition by looking for **0032-nnn** in *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

ERRORS

inmsg is not a valid buffer

blklen is less than zero

offset is less than **blklen**

blknum is less than zero

source is not a valid task id or wildcard value

type is not a valid message type or wildcard value

EXAMPLES

A message from task 0 is received by task 1, and the data values are placed at equal intervals in **inmsg** using MP_BRECV. In the comments, data values listed as "x" are not sent, received, or modified.

C Example

```
   #include <mpproto.h>

int bvrecv_example()
{
   int source,dest,type,blklen,offset,blknum,nbytes,rc;
   int numtask,taskid,inmsg[7],outmsg[7];

   mpc_environ(&numtask,&taskid);

   type = 1;
   blklen = sizeof(int);
   offset = 3 * sizeof(int);
   blknum = 3;
   source = 0;
   dest = 1;

   if(taskid == source)  {
      outmsg[0] = 1;
      outmsg[3] = 2;
      outmsg[6] = 3;
      /* "outmsg" is now {1,x,x,2,x,x,3}  */

      rc = mpc_bvsend(outmsg,blklen,offset,blknum,dest,type);
      if(rc)  return(-1);  /* error */
   } else {
      if(taskid == dest) {
```

```
         rc = mpc_bvrecv(inmsg,blklen,offset,blknum,&source,
                         &type,&nbytes);
         if(rc)  return(-1);  /* error */

         /* "inmsg" is now = {1,x,x,2,x,x,3}  */
      }
   }
   return(0);
}
```

FORTRAN Example

```
      SUBROUTINE BVRECV_EXAMPLE

      INTEGER SOURCE,DEST,TYPE,BLKLEN,OFFSET,BLKNUM,NBYTES
      INTEGER NUMTASK,TASKID,INMSG(7),OUTMSG(7)

      CALL MP_ENVIRON(NUMTASK,TASKID)

      TYPE = 1
      BLKLEN = 4
      OFFSET = 3 * 4
      BLKNUM = 3
      SOURCE = 0
      DEST = 1

      IF(TASKID .EQ. SOURCE)  THEN
         OUTMSG(1) = 1
         OUTMSG(4) = 2
         OUTMSG(7) = 3
c        "OUTMSG" is now = {1,x,x,2,x,x,3}

         CALL MP_BVSEND(OUTMSG,BLKLEN,OFFSET,BLKNUM,DEST,TYPE)
      ELSE
         IF(TASKID .EQ. DEST)  THEN

            CALL MP_BVRECV(INMSG,BLKLEN,OFFSET,BLKNUM,SOURCE,
     +                     TYPE,NBYTES)

c           "INMSG" is now = {1,x,x,2,x,x,3}

         ENDIF
      ENDIF

      END
```

RELATED INFORMATION

Subroutines: MP_BVSEND(3), mpc_bvsend(3), MP_UNPACK(3), mpc_unpack(3)

NAME

MP_BVSEND, mpc_bvsend – Sends a non-contiguous message and waits until the message buffer can be reused.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mpproto.h>
int mpc_bvsend(const void *outmsg, size_t blklen, size_t offset, int blknum,
               int dest, int type);
```

FORTRAN SYNOPSIS

```
MP_BVSEND(INTEGER OUTMSG(*), INTEGER BLKLEN, INTEGER OFFSET, INTEGER BLKNUM,
          INTEGER DEST, INTEGER TYPE)
```

PARAMETERS

outmsg is the starting location of the buffer that contains the message to be sent. **outmsg** consists of **blknum** blocks, with each block consisting of **blklen** contiguous bytes. The starting addresses of consecutive blocks are separated by **offset** bytes.

blklen is a non-negative integer that specifies the size, in bytes, of each block contained in buffer **outmsg**.

offset is an integer that specifies the separation, in bytes, between the starting addresses of consecutive blocks. **offset** must be greater than or equal to **blklen**.

blknum is an integer that specifies the number of blocks to be sent.

dest is an integer that specifies the task to which the message is to be sent.

type is an integer that specifies the type of the message to be sent.

DESCRIPTION

Takes data out of non-contiguous memory and sends it as a message. Then, the task waits until the application buffer in the sending task is free for reuse.

Completion does not imply that the message has been received by the destination task.

RETURN VALUES

In C, return code = 0 means the operation was successful. Return code = -1 means that an error occurred and the global variable **mperrno** contains the error number. This number, “nnn,” can be used to find a description of the error condition by looking for **0032-nnn** in *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

ERRORS

outmsg is not a valid buffer

blklen is less than zero

offset is less than **blklen**

blknum is less than zero

dest is not a valid task id

type is not a valid message type

EXAMPLES

Selected values from **outmsg** on task 0 are sent to task 1 using MP_BVSEND. In the comments, data values listed as "x" are not sent, received, or modified.

C Example

```
   #include <mpproto.h>

int bvsend_example()
{
   int source,dest,type,blklen,offset,blknum,nbytes,rc;
   int numtask,taskid,inmsg[7],outmsg[7];

   mpc_environ(&numtask,&taskid);

   type = 1;
   blklen = sizeof(int);
   offset = 3 * sizeof(int);
   blknum = 3;
   source = 0;
   dest = 1;

   if(taskid == source)  {
      outmsg[0] = 1;
      outmsg[3] = 2;
      outmsg[6] = 3;
      /* "outmsg" is now {1,x,x,2,x,x,3}  */

      rc = mpc_bvsend(outmsg,blklen,offset,blknum,dest,type);
      if(rc)  return(-1);  /* error */
   } else {
      if(taskid == dest) {
         rc = mpc_bvrecv(inmsg,blklen,offset,blknum,&source,
                         &type,&nbytes);
         if (rc)  return(-1);  /* error */

         /* "inmsg" is now {1,x,x,2,x,x,3}  */
      }
```

```
   }
   return(0);
}
```

FORTRAN Example

```
      SUBROUTINE BVSEND_EXAMPLE

      INTEGER SOURCE,DEST,TYPE,BLKLEN,OFFSET,BLKNUM,NBYTES
      INTEGER NUMTASK,TASKID,INMSG(7),OUTMSG(7)

      CALL MP_ENVIRON(NUMTASK,TASKID)

      TYPE = 1
      BLKLEN = 4
      OFFSET = 3 * 4
      BLKNUM = 3
      SOURCE = 0
      DEST = 1

      IF(TASKID .EQ. SOURCE)  THEN
         OUTMSG(1) = 1
         OUTMSG(4) = 2
         OUTMSG(7) = 3
c        "OUTMSG" is now = {1,x,x,2,x,x,3}

         CALL MP_BVSEND(OUTMSG,BLKLEN,OFFSET,BLKNUM,DEST,TYPE)

      ELSE
         IF(TASKID .EQ. DEST)  THEN
            CALL MP_BVRECV(INMSG,BLKLEN,OFFSET,BLKNUM,SOURCE,
     +                     TYPE,NBYTES)

c           "INMSG" is now = {1,x,x,2,x,x,3}

         ENDIF
      ENDIF

      END
```

RELATED INFORMATION

Subroutines: MP_BVRECV(3), mpc_bvrecv(3), MP_PACK(3), mpc_pack(3)

NAME

MP_COMBINE, mpc_combine – Performs a reduction operation on data distributed across a group and distributes the result to all tasks in the group.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mpproto.h>
int mpc_combine(const void *outmsg, void *inmsg, size_t msglen,
                void (*func)(), int gid);
```

FORTRAN SYNOPSIS

```
MP_COMBINE(INTEGER OUTMSG(*), INTEGER INMSG(*), INTEGER MSGLEN,
           EXTERN FUNC, INTEGER GID)
```

PARAMETERS

outmsg is the application buffer that contains the local share of the distributed data.

inmsg is the application buffer in which the result of a reduction operation is stored.

msglen is a non-negative integer that specifies, in bytes, the size of **outmsg** and **inmsg**.

func is an external function which implements a reduction operation.

gid is a group identifier for the group of participating tasks.

DESCRIPTION

This routine performs a reduction operation on data distributed across a group. The result of the operation is stored in the application buffer **inmsg** in all tasks in the group.

The reduction is generally applied to one vector per task and the result is also a vector of the same length. **outmsg** and **inmsg** are of the same size.

You can define the reduction function or it can be one of the predefined functions listed in Appendix D, “Reduction Operations” on page 115.

outmsg and **inmsg** cannot overlap in memory.

RETURN VALUES

In C, return code = 0 means the operation was successful. Return code = -1 means that an error occurred and the global variable **mperrno** contains the error number. This number, "nnn," can be used to find a description of the error condition by looking for **0032-nnn** in *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

ERRORS

outmsg *or* **inmsg**	is not a valid buffer
msglen	is less than zero
gid	is undefined in the calling task

Develop mode error if

- all tasks in **gid** do not use identical values for **msglen**

EXAMPLES

In a partition of 4 tasks, MP_COMBINE is used to find the sum of the **outmsg** values from all of the tasks, and place it in **inmsg** on each task.

C Example

```
   #include <mpproto.h>
   /* extern void i_vadd() is declared by mpproto.h */
   /* The constant ALLGRP is defined by mpproto.h */
int combine_example()
{
   int numtask,taskid,inmsg,outmsg,msglen,rc;

   mpc_environ(&numtask,&taskid);

   msglen = sizeof(int);
   outmsg = 1;

   rc = mpc_combine(&outmsg,&inmsg,msglen,i_vadd,ALLGRP);
   if(rc)  return(-1);  /* error */

   /* "inmsg" is now equal to the number of tasks in the partition  */

   return(0);
}
```

FORTRAN Example

```
      SUBROUTINE COMBINE_EXAMPLE

      INTEGER NUMTASK,TASKID,INMSG,OUTMSG,MSGLEN,ALLGRP,NBUF(4)
      EXTERNAL I_VADD

      CALL MP_ENVIRON(NUMTASK,TASKID)
      CALL MP_TASK_QUERY(NBUF,4,3)     ! get "ALLGRP" value
      ALLGRP = NBUF(4)

      MSGLEN = 4
      OUTMSG = 1

      CALL MP_COMBINE(OUTMSG,INMSG,MSGLEN,I_VADD,ALLGRP)

c     "INMSG" is now equal to the number of tasks in the partition

      END
```

RELATED INFORMATION

Subroutines: MP_REDUCE(3), mpc_reduce(3), MP_PREFIX(3), mpc_reduce(3), i_vadd(3), s_vadd(3), d_vadd(3), i_vmul(3), s_vmul(3), d_vmul(3), i_vmax(3), s_vmax(3), d_vmax(3), i_vmin(3), s_vmin(3), d_vmin(3), b_vand(3), b_vor(3), b_vxor(3), l_vand(3), l_vor(3)

See the user-defined reduction functions listed in Appendix D, “Reduction Operations” on page 115.

NAME

MP_CONCAT, mpc_concat – Concatenates a distinct message from each task to create one message.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mpproto.h>
int mpc_concat(const void *outmsg, void *inmsg, size_t blklen, int gid);
```

FORTRAN SYNOPSIS

```
MP_CONCAT(INTEGER OUTMSG(*), INTEGER INMSG(*), INTEGER BLKLEN, INTEGER GID)
```

PARAMETERS

outmsg	is the application buffer that contains a distinct outgoing message for each task in the calling group.
inmsg	is the application buffer that stores the final concatenation result.
blklen	is a non-negative integer that specifies the size, in bytes, of a distinct message from each task in the calling group.
gid	is a group identifier for the group of participating tasks.

DESCRIPTION

This routine takes a message that has the size **blklen** from the buffer **outmsg** in each task in the group **gid**. Then, it concatenates them into a message having the size **blklen** times the number of tasks in the calling group, and places the resulting message in the local buffer **inmsg** in each task. Logically, this is the same as MP_GATHER followed by MP_BCAST.

A distinct outgoing message is sent from each task. The incoming identical message for each task is a concatenation according to the order of the group.

outmsg and **inmsg** cannot overlap in memory.

RETURN VALUES

In C, return code = 0 means the operation was successful. Return code = -1 means that an error occurred and the global variable **mperrno** contains the error number. This number, “nnn,” can be used to find a description of the error condition by looking for **0032-nnn** in *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

ERRORS

outmsg *or* **inmsg**	is not a valid buffer
blklen	is less than zero
gid	is undefined in the calling task

Develop mode error if:

- all tasks in **gid** do not use identical values for **blklen**

EXAMPLES

An ordered list of task ids is placed in **inmsg** on each task by MP_CONCAT. A partition of 4 tasks is assumed.

C Example

```
   #include <mpproto.h>

int concat_example()
{
   int numtask,taskid,blklen,rc;
   int inmsg[4],outmsg;

   mpc_environ(&numtask,&taskid);

   blklen = sizeof(int);
   outmsg = taskid;

   rc = mpc_concat(&outmsg,&inmsg[0],blklen,ALLGRP);
   if(rc)  return(-1);  /* error */

   /* "inmsg" now contains a list of task ids in ascending order,
      i.e. {0,1,2,3} for a partition of 4 tasks  */

   return(0);
}
```

FORTRAN Example

```
      SUBROUTINE CONCAT_EXAMPLE

      INTEGER NUMTASK,TASKID,INMSG(4),OUTMSG,BLKLEN,ALLGRP,NBUF(4)

      CALL MP_ENVIRON(NUMTASK,TASKID)
      CALL MP_TASK_QUERY(NBUF,4,3)     !  get "ALLGRP" value
      ALLGRP = NBUF(4)

      BLKLEN = 4
      OUTMSG = TASKID

      CALL MP_CONCAT(OUTMSG,INMSG,BLKLEN,ALLGRP)

c     "INMSG" now contains a list of task ids in ascending order,
c     i.e. {0,1,2,3} for a partition of 4 tasks
      END
```

RELATED INFORMATION

Subroutines: MP_INDEX(3), mpc_index(3), MP_GATHER(3), mpc_gather(3)

NAME

MP_ENVIRON, mpc_environ – Returns the number of tasks in the partition and the caller's task id.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mpproto.h>
int mpc_environ(int *numtask, int *taskid);
```

FORTRAN SYNOPSIS

```
MP_ENVIRON(INTEGER NUMTASK, INTEGER TASKID)
```

PARAMETERS

numtask is an integer which returns the number of tasks in the parallel partition.

taskid is an integer which returns the task id. Each task is assigned a unique integer between 0 and **numtask**-1, inclusive, for **taskid**.

DESCRIPTION

This subroutine returns the number of tasks in the partition (which is fixed at run time) and the task id. Each task in the partition obtains a unique task id which ranges from 0 to **numtask**-1, inclusive.

RETURN VALUES

In C, return code = 0 means the operation was successful. Return code = -1 means that an error occurred and the global variable **mperrno** contains the error number. This number, “nnn,” can be used to find a description of the error condition by looking for **0032-nnn** in *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

EXAMPLES

MP_ENVIRON is used to get the number of tasks in the partition, and the task id of the local task.

C Example

```
   #include <mpproto.h>

int environ_example()
{
   int numtask,taskid,rc;

   rc = mpc_environ(&numtask,&taskid);
   if(rc)  return(-1);  /* error */

   /* "numtask" is now equal to the number of tasks in the partition.
      "taskid" is the id of the local task  */

   return(0);
}
```

FORTRAN Example

```
      SUBROUTINE ENVIRON_EXAMPLE

      INTEGER NUMTASK,TASKID

      CALL MP_ENVIRON(NUMTASK,TASKID)

c     "NUMTASK" is now equal to the number of tasks in the partition.
c     "TASKID" is the id of the local task

      END
```

NAME

MP_GATHER, mpc_gather – Concatenates a distinct message from each task to create a single message on task **dest**.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mpproto.h>
int mpc_gather(const void *outmsg, void *inmsg, size_t blklen,
               int dest, int gid);
```

FORTRAN SYNOPSIS

```
MP_GATHER(INTEGER OUTMSG(*), INTEGER INMSG(*), INTEGER BLKLEN,
          INTEGER DEST, INTEGER GID)
```

PARAMETERS

outmsg is the application buffer that contains the outgoing message to be gathered.

inmsg is the application buffer into which the final gathered result is placed for task id equal to **dest**. For the other tasks in the same calling group, **inmsg** is used as scratch space by the system.

inmsg is of size **blklen** multiplied by the number of tasks in the group.

blklen is a non-negative integer that specifies the size, in bytes, of the outgoing message in each of the sending tasks in the group.

dest is an integer that specifies the task to which the final gathered result of the calling group will be written.

gid is a group identifier for the group of participating tasks.

DESCRIPTION

This routine gathers a distinct message of length **blklen** from each task in the calling group, including the destination task **dest**. It then places a concatenated message (according to the order in the group) in the application buffer **inmsg** in the task with id **dest**.

outmsg and **inmsg** cannot overlap in memory.

RETURN VALUES

In C, return code = 0 means the operation was successful. Return code = -1 means that an error occurred and the global variable **mperrno** contains the error number. This number, “nnn,” can be used to find a description of the error condition by looking for **0032-nnn** in *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

ERRORS

outmsg *or* **inmsg**	is not a valid buffer
blklen	is less than zero
dest	is not a valid task id or is not a member of **gid**
gid	is undefined in the calling task

Develop mode error if:

- all tasks in **gid** do not use idential values for **dest** and **blklen**.

EXAMPLES

An ordered list of task ids is placed in **inmsg** on task 0 by MP_GATHER. A partition of 4 tasks is assumed.

C Example

```
   #include <mpproto.h>

int gather_example()
{
   int numtask,taskid,dest,blklen,rc;
   int inmsg[4],outmsg;

   mpc_environ(&numtask,&taskid);

   blklen = sizeof(int);
   outmsg = taskid;
   dest = 0;

   rc = mpc_gather(&outmsg,inmsg,blklen,dest,ALLGRP);
   if(rc)  return(-1);  /* error */

   /* "inmsg" in the "dest" task (0) now contains a list of task ids in
      ascending order, i.e. {0,1,2,3} for a partition of 4 tasks  */

   return(0);
}
```

FORTRAN Example

```
      SUBROUTINE GATHER_EXAMPLE

      INTEGER NUMTASK,TASKID,DEST,INMSG(4),OUTMSG,BLKLEN,ALLGRP,NBUF(4)

      CALL MP_ENVIRON(NUMTASK,TASKID)
      CALL MP_TASK_QUERY(NBUF,4,3)      !  get "ALLGRP" value
      ALLGRP = NBUF(4)

      BLKLEN = 4
      OUTMSG = TASKID
      DEST = 0

      CALL MP_GATHER(OUTMSG,INMSG,BLKLEN,DEST,ALLGRP)

c     "INMSG" in the "DEST" task now contains a list of task ids in
c     ascending order, i.e. {0,1,2,3} for a partition of 4 tasks

      END
```

RELATED INFORMATION

Subroutines: MP_SCATTER(3), mpc_scatter(3)

NAME

MP_GETLABEL, mpc_getlabel – Returns the label assigned to a group.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mpproto.h>
int mpc_getlabel(int *label, int gid);
```

FORTRAN SYNOPSIS

```
MP_GETLABEL(INTEGER LABEL, INTEGER GID)
```

PARAMETERS

label is an integer into which the label of the specified task group is placed.

gid is an integer that specifies the identifier of the task group for which the label is returned.

DESCRIPTION

This routine returns the label of the task group **gid** in the argument **label**. The label of the predefined group, **allgrp**, is 0.

It is not necessary for each task in the group to call MP_GETLABEL.

RETURN VALUES

In C, return code = 0 means the operation was successful. Return code = -1 means that an error occurred and the global variable **mperrno** contains the error number. This number, “nnn,” can be used to find a description of the error condition by looking for **0032-nnn** in *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

ERRORS

gid is undefined in the calling task

EXAMPLES

A new task group is created, and MP_GETLABEL is used to retrieve the label given to the group.

C Example

```
   #include <mpproto.h>

int getlabel_example()
{
   int label,numtask,taskid,tasks[4],i,gid,rc;

   mpc_environ(&numtask,&taskid);

   for(i=0;i<numtask;i++)
      tasks[i] = i;

   label = 55;
   rc = mpc_group(numtask,tasks,label,&gid); /* create a new group */
   if(rc)  return(-1);  /* error */
   label = 0;                              /* erase old value */

   rc = mpc_getlabel(&label,gid);
   if(rc)  return(-1);  /* error */

   /* "label" is now set to "55"  */

   return(0);
}
```

FORTRAN Example

```
      SUBROUTINE GETLABEL_EXAMPLE

      INTEGER LABEL,NUMTASK,TASKID,TASKS(4),I,GID

      CALL MP_ENVIRON(NUMTASK,TASKID)

      DO 10 I=1,NUMTASK
   10    TASKS(I) = I - 1

      LABEL = 55
      CALL MP_GROUP(NUMTASK,TASKS,LABEL,GID)   !  create a new group
      LABEL = 0                                !  erase old value

      CALL MP_GETLABEL(LABEL,GID)

c     "LABEL" is now set to "55"

      END
```

RELATED INFORMATION

Subroutines: MP_GROUP(3), mpc_group(3), MP_PARTITION(3), mpc_partition(3)

NAME

MP_GETMEMBERS, mpc_getmembers – Returns the members of a group.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mpproto.h>
int mpc_getmembers(int glist[], int gid);
```

FORTRAN SYNOPSIS

```
MP_GETMEMBERS(INTEGER GLIST(*), INTEGER GID)
```

PARAMETERS

glist is an integer array into which the ordered membership list of the task group **gid** is placed. **glist** must be large enough to hold all of the task ids in the group.

gid is an integer that specifies the identifier of the task group for which the membership array is returned.

DESCRIPTION

This routine returns, in the argument **glist**, the array of task ids forming the task group **gid**. All tasks in the group do not have to call it.

RETURN VALUES

In C, return code = 0 means the operation was successful. Return code = -1 means that an error occurred and the global variable **mperrno** contains the error number. This number, “nnn,” can be used to find a description of the error condition by looking for **0032-nnn** in *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

ERRORS

gid is undefined in the calling task

EXAMPLES

A new task group is created, and MP_GETMEMBERS is used to retrieve the list of tasks in the group.

C Example

```
   #include <mpproto.h>

int getmembers_example()
{
   int label,numtask,taskid,tasks[4],i,gid,rc;

   mpc_environ(&numtask,&taskid);

   for(i=0;i<numtask;i++)
      tasks[i] = i;

   label = 1;
   rc = mpc_group(numtask,tasks,label,&gid); /* create a new group */
   if(rc)  return(-1);  /* error */
   for(i=0;i<numtask;i++)
      tasks[i] = -1;                          /* erase old values */

   rc = mpc_getmembers(&tasks,gid);
   if(rc)  return(-1);  /* error */

   /* "tasks" is now = {0,1,2,3} for a partition of 4 tasks  */

   return(0);
}
```

FORTRAN Example

```
      SUBROUTINE GETMEMBERS_EXAMPLE

      INTEGER LABEL,NUMTASK,TASKID,TASKS(4),I,GID

      CALL MP_ENVIRON(NUMTASK,TASKID)

      DO 10 I=1,NUMTASK
   10    TASKS(I) = I - 1

      LABEL = 1
      CALL MP_GROUP(NUMTASK,TASKS,LABEL,GID)   !  create a new group
      DO 20 I=1,NUMTASK
   20    TASKS(I) = -1                         !  erase old values

      CALL MP_GETMEMBERS(TASKS,GID)

c     "TASKS" is now = {0,1,2,3} for a partition of 4 tasks

      END
```

RELATED INFORMATION

Subroutines: MP_GROUP(3), mpc_group(3), MP_PARTITION(3), mpc_partition(3)

NAME

MP_GETRANK, mpc_getrank – Returns a task's rank within a group.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mpproto.h>
int mpc_getrank(int *rank, int taskid, int gid);
```

FORTRAN SYNOPSIS

```
MP_GETRANK(INTEGER RANK, INTEGER TASKID, INTEGER GID)
```

PARAMETERS

rank	is an integer into which the rank of the task **taskid** in the task group **gid** is placed. If the **taskid** is not a member of **gid**, **rank** will be set to -1.
taskid	is an integer that specifies the task id to check for its membership and rank in the task group **gid**.
gid	is an integer that specifies the identifier of the task group for which the task **taskid** is checked for its membership and rank.

DESCRIPTION

If a task belongs to the group identified by **gid**, this routine returns, in **rank**, the rank of **taskid**. Ranks range from 0 through the number of tasks in the group - 1. Typically, **taskid** is the id of the calling task but it can be any other task id as well.

All tasks in the group do not have to call this routine.

RETURN VALUES

In C, return code = 0 means the operation was successful. Return code = -1 means that an error occurred and the global variable **mperrno** contains the error number. This number, “nnn,” can be used to find a description of the error condition by looking for **0032-nnn** in *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

ERRORS

taskid	is not a valid task id
gid	is undefined in the calling task

EXAMPLES

A task group with descending task id order is created, and MP_GETRANK is used to determine the rank of the local task in the new group.

C Example

```
   #include <mpproto.h>

int getrank_example()
{
   int label,numtask,taskid,tasks[4],i,gid,lasttask,rank,rc;

   mpc_environ(&numtask,&taskid);
   lasttask = numtask - 1;
   for(i=0;i<numtask;i++)              /* create reversed task list */
      tasks[i] = lasttask - i;
      /* "tasks" is now = {3,2,1,0}, assuming a partition of 4 tasks */

   label = 1;
   rc = mpc_group(numtask,tasks,label,&gid); / *create a new group */
   if(rc)  return(-1);  /* error */

   rc = mpc_getrank(&rank,taskid,gid);
   if(rc)  return(-1);  /* error */

   /* "rank" is now set to "lasttask - taskid"  */

   return(0);
}
```

FORTRAN Example

```
      SUBROUTINE GETRANK_EXAMPLE

      INTEGER LABEL,NUMTASK,TASKID,TASKS(4),I,GID,LASTTASK,RANK

      CALL MP_ENVIRON(NUMTASK,TASKID)
      LASTTASK = NUMTASK - 1
      DO 10 I=1,NUMTASK                    ! create reversed task list
   10    TASKS(I) = NUMTASK - I
c     "TASKS" is now = {3,2,1,0}, assuming a partition of 4 tasks

      LABEL = 1
      CALL MP_GROUP(NUMTASK,TASKS,LABEL,GID)   ! create a new group

      CALL MP_GETRANK(RANK,TASKID,GID)

c     "RANK" is now set to "LASTTASK - TASKID"

      END
```

RELATED INFORMATION

Subroutines: MP_GROUP(3), mpc_group(3), MP_PARTITION(3), mpc_partition(3), MP_GETTASKID(3), mpc_gettaskid(3), MP_GETMEMBERS(3), mpc_getmembers(3)

NAME

MP_GETSIZE, mpc_getsize – Returns the number of tasks in a group.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mpproto.h>
int mpc_getsize(int *gsize, int gid);
```

FORTRAN SYNOPSIS

```
MP_GETSIZE(INTEGER GSIZE, INTEGER GID)
```

PARAMETERS

gsize is an integer into which the number of tasks (size) in the specified task group is placed.

gid is an integer that specifies the identifier of the task group for which the size is returned.

DESCRIPTION

This routine returns the number of tasks in the task group **gid** in the argument **gsize**. All tasks in the group do not have to call this routine.

RETURN VALUES

In C, return code = 0 means the operation was successful. Return code = -1 means that an error occurred and the global variable **mperrno** contains the error number. This number, “nnn,” can be used to find a description of the error condition by looking for **0032-nnn** in *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

ERRORS

gid is undefined in the calling task

EXAMPLES

A task group is created, and MP_GETSIZE is used to verify the number of tasks in the group. A partition size of 4 is assumed.

C Example

```
   #include <mpproto.h>

int getsize_example()
{
   int label,numtask,taskid,tasks[4],i,gid,gsize,rc;

   mpc_environ(&numtask,&taskid);

   for(i=0;i<numtask;i++)
      tasks[i] = i;

   label = 1;
   rc = mpc_group(numtask,tasks,label,&gid); / *create a new group */
   if(rc)  return(-1);  /* error */

   rc = mpc_getsize(&gsize,gid);
   if(rc)  return(-1);  /* error */

   /* "gsize" is now equal to "numtask"  */

   return(0);
}
```

FORTRAN Example

```
      SUBROUTINE GETSIZE_EXAMPLE

      INTEGER LABEL,NUMTASK,TASKID,TASKS(4),I,GID,GSIZE

      CALL MP_ENVIRON(NUMTASK,TASKID)

      DO 10 I=1,NUMTASK
   10    TASKS(I) = I - 1

      LABEL = 1
      CALL MP_GROUP(NUMTASK,TASKS,LABEL,GID)   !  create a new group

      CALL MP_GETSIZE(GSIZE,GID)

c     "GSIZE" is now equal to "NUMTASK"

      END
```

RELATED INFORMATION

Subroutines: MP_GROUP(3), mpc_group(3), MP_PARTITION(3), mpc_partition(3), MP_GETMEMBERS(3), mpc_getmembers(3)

NAME

MP_GETTASKID, mpc_gettaskid – Returns the task id for a **rank** within a group.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mpproto.h>
int mpc_gettaskid(int rank, int *taskid, int gid);
```

FORTRAN SYNOPSIS

```
MP_GETTASKID(INTEGER RANK, INTEGER TASKID, INTEGER GID)
```

PARAMETERS

rank is an integer that specifies the rank of a task in the group.

taskid is an integer into which the task id of the task with the given rank is placed.

gid is an integer that specifies the identifier of the task group to query.

DESCRIPTION

This routine returns the task id of the task which is ranked **rank** in the task group **gid**. It places this information in **taskid**.

Valid ranks range from 0 through the number of tasks in the group - 1.

All tasks in the group do not have to call this routine.

RETURN VALUES

In C, return code = 0 means the operation was successful. Return code = -1 means that an error occurred and the global variable **mperrno** contains the error number. This number, “nnn,” can be used to find a description of the error condition by looking for **0032-nnn** in *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

ERRORS

rank is less than zero or is greater than or equal to the number of tasks in **gid**

gid is undefined in the calling task

EXAMPLES

A reverse-ordered task group of 4 is created, and MP_GETTASKID is used to determine the task id of a given rank in the resulting group.

C Example

```
   #include <mpproto.h>

int gettaskid_example()
{
   int label,numtask,taskid,tasks[4],i,gid,lasttask,rank,myid,rc;

   mpc_environ(&numtask,&taskid);
   lasttask = numtask - 1;
   for(i=0;i<numtask;i++)              /* create reversed task list */
      tasks[i] = lasttask - i;
      /* "tasks" is now = {3,2,1,0}, assuming a partition of 4 tasks */

   label = 1;

   rc = mpc_group(numtask,tasks,label,&gid); /* create a new group */
   if(rc)  return(-1);  /* error */
   rank = lasttask - taskid;

   rc = mpc_gettaskid(rank,&myid,gid);
   if(rc)  return(-1);  /* error */

   /* "myid" is now equal to "taskid"  */

   return(0);
}
```

FORTRAN Example

```
      SUBROUTINE GETTASKID_EXAMPLE

      INTEGER LABEL,NUMTASK,TASKID,TASKS(4),I,GID,RANK,MYID

      CALL MP_ENVIRON(NUMTASK,TASKID)

      DO 10 I=1,NUMTASK                 ! create reversed task list
   10    TASKS(I) = NUMTASK - I
c     "TASKS" is now = {3,2,1,0}, assuming a partition of 4 tasks

      LABEL = 1
      CALL MP_GROUP(NUMTASK,TASKS,LABEL,GID)   ! create a new group
      RANK = NUMTASK - TASKID - 1              ! compute rank of local task

      CALL MP_GETTASKID(RANK,MYID,GID)

c     "MYID" is now equal to "TASKID"

      END
```

RELATED INFORMATION

Subroutines: MP_GROUP(3), mpc_group(3), MP_PARTITION(3), mpc_partition(3), MP_GETRANK(3), mpc_getrank(3), MP_GETMEMBERS(3), mpc_getmembers(3)

NAME

MP_GROUP, mpc_group – Explicitly defines a task group.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mpproto.h>
int mpc_group(int gsize, const int glist[], int label, int *gid);
```

FORTRAN SYNOPSIS

```
MP_GROUP(INTEGER GSIZE, INTEGER GLIST(*), INTEGER LABEL, INTEGER GID)
```

PARAMETERS

gsize	is an integer that specifies the number of tasks in **glist**.
glist	is the array of task ids which form a task group.
label	is a user-defined identifier to associate with the group. The predefined group **allgrp** has label 0.
gid	is an integer variable into which the task group identifier of the newly formed group is placed.

DESCRIPTION

Allows you to define a task group by explicitly specifying the list of tasks that will form the group. You do this by specifying:

- A membership array of task ids
- The size of the group
- A label to associate with the group.

This routine returns **gid**, a new task group identifier, as an argument.

You can use MP_GETSIZE, MP_GETMEMBERS, and MP_GETLABEL to obtain information of gsize, glist and label, respectively.

RETURN VALUES

In C, return code = 0 means the operation was successful. Return code = -1 means that an error occurred and the global variable **mperrno** contains the error number. This number, “nnn,” can be used to find a description of the error condition by looking for **0032-nnn** in *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

ERRORS

gsize is less than or equal to zero, or greater than the number of tasks in the partition

Develop mode error if:

- one or more of the task ids in **glist** is invalid
- a task id appears in **glist** more than once
- the id of the local task does not appear in **glist**
- all tasks in **gid** do not use identical values for **gsize** and **label**
- the task id array **glist** is not identical for all tasks in **gid**.

EXAMPLES

A task group is created consisting of 4 tasks in decreasing task id order.

C Example

```
   #include <mpproto.h>

int group_example()
{
   int label,numtask,taskid,tasks[4],i,gid,rc;

   mpc_environ(&numtask,&taskid);
   for(i=0;i<numtask;i++)              /* create reversed task list */
      tasks[i] = numtask - i - 1;
      /* "tasks" is now = {3,2,1,0}, assuming a partition of 4 tasks */

   label = 1;

   rc = mpc_group(numtask,tasks,label,&gid); /* create a new group */
   if(rc)  return(-1);  /* error */

   /* "gid" now identifies the task group {3,2,1,0}, assuming a partition
      of 4 tasks  */

   return(0);
}
```

FORTRAN Example

```
      SUBROUTINE GROUP_EXAMPLE

      INTEGER LABEL,NUMTASK,TASKID,TASKS(4),I,GID

      CALL MP_ENVIRON(NUMTASK,TASKID)

      LABEL = 1
      DO 10 I = 1,NUMTASK                    ! create reversed task list
   10    TASKS(I) = NUMTASK - I
c     "TASKS" is now = {3,2,1,0}, assuming a partition of 4 tasks

      CALL MP_GROUP(NUMTASK,TASKS,LABEL,GID)     ! create a new group
```

```
c     "GID" now identifies the task group {3,2,1,0} with a label of 1,
c     assuming a partition of 4 tasks.

      END
```

RELATED INFORMATION

Subroutines: MP_PARTITION(3), mpc_partition(3)

NAME

MP_INDEX, mpc_index – Performs an all-to-all personalized communication operation.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mpproto.h>
int mpc_index(const void *outmsg, void *inmsg, size_t blklen, int gid);
```

FORTRAN SYNOPSIS

```
MP_INDEX(INTEGER OUTMSG(*), INTEGER INMSG(*), INTEGER BLKLEN, INTEGER GID)
```

PARAMETERS

outmsg is the application buffer which contains the distinct outgoing messages.

inmsg is the application buffer which stores the incoming messages from all the tasks in the group, including itself.

blklen is a non-negative integer that specifies the size, in bytes, of each distinct outgoing message.

gid is a group identifier for the group of participating tasks.

DESCRIPTION

This routine performs an all-to-all personalized communication operation. This means that each task sends a distinct message to all the tasks in the same group. Each task in the calling group partitions its local buffers **outmsg** and **inmsg** into *N* blocks each of size **blklen**, where *N* is the number of tasks in the group. The *i*-th task in the group sends the *j*-th block in **outmsg** to the *j*-th task in the same group. The *j*-th task stores the message in the *i*-th block in **inmsg**. Therefore, the *i*-th block of **outmsg** in the *i*-th task will be copied locally to the *i*-th block of **inmsg**.

outmsg and **inmsg** cannot overlap in memory.

RETURN VALUES

In C, return code = 0 means the operation was successful. Return code = -1 means that an error occurred and the global variable **mperrno** contains the error number. This number, “nnn,” can be used to find a description of the error condition by looking for **0032-nnn** in *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

ERRORS

outmsg *or* **inmsg**	is not a valid buffer
blklen	is less than zero
gid	is undefined in the calling task

Develop mode error if:

- all tasks in **gid** do not use identical values for **blklen**

EXAMPLES

MP_INDEX is used to send a distinct message from each task in a partition of 4 to every other task in the partition. The messages originate from **outmsg** and arrive in **inmsg**.

C Example

```
   #include <mpproto.h>

int index_example()
{
   int numtask,taskid,blklen,i,rc;
   int inmsg[4],outmsg[4];

   mpc_environ(&numtask,&taskid);

   for(i=0;i<numtask;i++)
      outmsg[i] = i;
   /* "outmsg" is now {0,1,2,3}  */
   blklen = sizeof(int);

   rc = mpc_index(outmsg,inmsg,blklen,ALLGRP);
   if(rc)  return(-1);  /* error */

   /* "inmsg" now contains 4 copies of the local task id, assuming
      a partition of 4 tasks  */

   return(0);
}
```

FORTRAN Example

```
      SUBROUTINE INDEX_EXAMPLE

      INTEGER NUMTASK,TASKID,BLKLEN,ALLGRP,I
      INTEGER INMSG(4),OUTMSG(4),NBUF(4)

      CALL MP_ENVIRON(NUMTASK,TASKID)
      CALL MP_TASK_QUERY(NBUF,4,3)      !  Get "ALLGRP" value
      ALLGRP = NBUF(4)

      BLKLEN = 4
      DO 10 I=1,NUMTASK
   10    OUTMSG(I) = I - 1
c     "OUTMSG" is now {0,1,2,3}

      CALL MP_INDEX(OUTMSG,INMSG,BLKLEN,ALLGRP)

c     "INMSG" now contains 4 copies of the local task id, assuming
c     a partition of 4 tasks

      END
```

RELATED INFORMATION

Subroutines: MP_CONCAT(3), mpc_concat(3)

NAME

MP_LOCKRNC, mpc_lockrnc – Asserts or frees a lock which can block the dispatching of MP_RCVNCALL handlers and returns the old lock value.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mpproto.h>
int mpc_lockrnc(int newlock, int *oldlock);
```

FORTRAN SYNOPSIS

```
MP_LOCKRNC(INTEGER NEWLOCK, INTEGER OLDLOCK)
```

PARAMETERS

newlock is either zero to free the lock or non-zero to assert it.

oldlock is returned as zero if the lock was free and one if it was already locked.

DESCRIPTION

This routine asserts or frees a lock which can block the dispatching of MP_RCVNCALL handlers and returns the old lock value. An example of MP_LOCKRNC usage is an application which needs to update information which is shared with one or more handlers. Because of the asynchronous nature of handler dispatching, this may not be safe to do unless pending MP_RCVNCALLs can be deferred until the update is completed. While the lock is in effect any MP_RCVNCALL messages which arrive will be enqueued so that once it is released each message handler will be dispatched in turn.

The lock **does not** count asserts and frees. The application which asserts an "MP_LOCKRNC(1,OLDLOCK)" should use the value returned in **OLDLOCK** to control the lock relinquishing call to MP_LOCKRNC either by using "MP_LOCKRNC(OLDLOCK,...)" or by calling "MP_LOCKRNC(0,...)" only if **OLDLOCK** equals zero.

The MP_LOCKRNC call must never be used within handler code. When a user handler is active, the lock is in force with respect to any other pending handler.

RETURN VALUES

In C, return code = 0 means the operation was successful. Return code = -1 means that an error occurred and the global variable **mperrno** contains the error number. This number, “nnn,” can be used to find a description of the error condition by looking for **0032-nnn** in *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

ERRORS

MP_LOCKRNC called from user handler.

EXAMPLES

The MP_LOCKRNC and MP_RCVNCALL subroutines are so interrelated that they are best demonstrated by a single example using them together. The C and FORTRAN examples to illustrate MP_LOCKRNC will be found in the example code for MP_RCVNCALL.

RELATED INFORMATION

Subroutines: MP_RCVNCALL(3), mpc_rcvncall(3),

NAME

MP_PACK, mpc_pack – Packs data from a non-contiguous buffer into a contiguous buffer.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mpproto.h>
int mpc_pack(const void *begbuf, void *endbuf,
             size_t blklen, size_t offset, int blknum);
```

FORTRAN SYNOPSIS

```
MP_PACK(INTEGER BEGBUF(*), INTEGER ENDBUF(*), INTEGER BLKLEN,
        INTEGER OFFSET, INTEGER BLKNUM)
```

PARAMETERS

begbuf is the starting location of the buffer that contains the initial non-contiguous data. **begbuf** consists of **blknum** blocks, with each block consisting of **blklen** contiguous bytes. The starting addresses of consecutive blocks are separated by **offset** bytes.

endbuf is the buffer into which the packed data is to be placed.

blklen is an integer specifying the size, in bytes, of each block contained in buffer **begbuf**.

offset is an integer specifying the separation, in bytes, between the starting addresses of consecutive blocks.

blknum is an integer specifying the number of blocks contained in the buffer **begbuf**.

DESCRIPTION

Packs data from a non-contiguous buffer into a contiguous buffer. For MP_PACK, arguments **blklen**, **offset**, and **blknum** are associated with the source buffer **begbuf**.

RETURN VALUES

In C, return code = 0 means the operation was successful. Return code = -1 means that an error occurred and the global variable **mperrno** contains the error number. This number, “nnn,” can be used to find a description of the error condition by looking for **0032-nnn** in *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

ERRORS

begbuf *or* **endbuf**	is not a valid buffer
blklen	is less than zero
offset	is less than **blklen**
blknum	is less than zero

EXAMPLES

Selected data values in **begbuf** are packed into **endbuf**.

C Example

```
   #include <mpproto.h>

int pack_example()
{
 int blklen,offset,blknum,i,rc;
 int begbuf[16],endbuf[9];

 for(i=0;i<16;i++)
    begbuf[i] = i;  /* "begbuf" = {0,1,2,3,4,5,6,7,8,9,10,
                                              11,12,13,14,15} * 1

 blklen = 3 * sizeof(int);
 offset = 5 * sizeof(int);
 blknum = 3;

 rc = mpc_pack(begbuf,endbuf,blklen,offset,blknum);
 if(rc)  return(-1);  /* error */

 /* "endbuf" = {0,1,2,5,6,7,10,11,12}  */

 return(0);
}
```

FORTRAN Example

```
      SUBROUTINE PACK_EXAMPLE

      INTEGER BLKLEN,OFFSET,BLKNUM,I
      INTEGER BEGBUF(16),ENDBUF(9)

      DO 10 I=1,16
   10    BEGBUF(I) = I-1
c     "BEGBUF" = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}

      BLKLEN = 3 * 4
      OFFSET = 5 * 4
      BLKNUM = 3

      CALL MP_PACK(BEGBUF,ENDBUF,BLKLEN,OFFSET,BLKNUM)

c     "ENDBUF" = {0,1,2,5,6,7,10,11,12}

      END
```

RELATED INFORMATION

Subroutines: MP_UNPACK(3), mpc_unpack(3), MP_BVSEND(3), mpc_bvsend(3)

NAME

MP_PARTITION, mpc_partition – Partitions an existing task group.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mpproto.h>
int mpc_partition(int parent_gid, int key, int label, int *gid);
```

FORTRAN SYNOPSIS

```
MP_PARTITION(INTEGER PARENT_GID, INTEGER KEY, INTEGER LABEL, INTEGER GID)
```

PARAMETERS

parent_gid is a previously defined group to partition.

key is an integer value used to specify the initial rank in the new group of the task.

label is an integer value used to partition the parent group. Tasks with the same value of label are formed into new groups.

gid is an integer returning the task group identifier of the newly formed group.

DESCRIPTION

This routine forms multiple new groups by partitioning a previously defined task group **parent_gid**, based on the local value of **label** supplied by each task.

Each task only sees the new group which it is in. The order in each new group is determined by sorting task ids in ascending order of **key**.

RETURN VALUES

In C, return code = 0 means the operation was successful. Return code = -1 means that an error occurred and the global variable **mperrno** contains the error number. This number, “nnn,” can be used to find a description of the error condition by looking for **0032-nnn** in *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

ERRORS

parent_gid is undefined in the calling task

EXAMPLES

A partition of 4 tasks is divided into 2 groups, with the even-numbered tasks in one group and the odd-numbered tasks in the other group.

C Example

```
   #include <mpproto.h>

int partition_example()
{
   int numtask,taskid,gid,label,key,rc;
   int tasks[4];

   mpc_environ(&numtask,&taskid);

   label = taskid % 2;
   key   = -taskid;

   rc = mpc_partition(ALLGRP,key,label,&gid);
   if(rc)  return(-1);  /* error */

   /* "gid" now identifies the task group {2,0} on tasks 0 and 2, and
      the group {3,1} on tasks 1 and 3, assuming a partition of 4 tasks */

   return(0);
}
```

FORTRAN Example

```
      SUBROUTINE PARTITION_EXAMPLE

      INTEGER NUMTASK,TASKID,ALLGRP,GID,LABEL,KEY
      INTEGER TASKS(4),NBUF(4)

      CALL MP_ENVIRON(NUMTASK,TASKID)
      CALL MP_TASK_QUERY(NBUF,4,3)    !  Get "ALLGRP" value
      ALLGRP = NBUF(4)

      LABEL = MOD(TASKID,2)
      KEY   = -TASKID

      CALL MP_PARTITION(ALLGRP,KEY,LABEL,GID)

c     "GID" now identifies the task group {2,0} on tasks 0 and 2, and
c     the group {3,1} on tasks 1 and 3, assuming a partition of 4 tasks

      END
```

RELATED INFORMATION

Subroutines: MP_GROUP(3), mpc_group(3)

NAME

MP_PREFIX, mpc_prefix – Applies a parallel prefix operation with respect to a reduction operation on data distributed across a group and places the corresponding result in each task in the group.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mpproto.h>
int mpc_prefix(const void *outmsg, void *inmsg, size_t msglen,
               void (*func)(), int gid);
```

FORTRAN SYNOPSIS

```
MP_PREFIX(INTEGER OUTMSG(*), INTEGER INMSG(*), INTEGER MSGLEN,
          EXTERN FUNC, INTEGER GID)
```

PARAMETERS

outmsg is the application buffer that contains the local share of the distributed data.

inmsg is the application buffer into which the result of prefix with respect to the current task is stored.

msglen is a non-negative integer that specifies the size, in bytes, of outmsg and inmsg.

func is an external function which implements a reduction operation.

gid is a group identifier for the group of participating tasks.

DESCRIPTION

This subroutine applies a parallel prefix operation with respect to a reduction operation on data distributed across a group and places the corresponding result in each task in the group.

RETURN VALUES

In C, return code = 0 means the operation was successful. Return code = -1 means that an error occurred and the global variable **mperrno** contains the error number. This number, “nnn,” can be used to find a description of the error condition by looking for **0032-nnn** in *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

ERRORS

outmsg *or* **inmsg**	is not a valid buffer
msglen	is less than zero
gid	is undefined in the calling task

Develop mode error if:

- all tasks in **gid** do not use identical values for **msglen**

EXAMPLES

In a partition of 4 tasks, MP_PREFIX is used to generate a task-wise cumulative sum of **outmsg** values in each task in the partition. That is, each task's **inmsg** is set equal to the sum of the **outmsg** values from all tasks numerically less than or equal to itself.

C Example

```
   #include <mpproto.h>
   /* extern void i_vadd() is declared by mpproto.h */

int prefix_example()
{
   int numtask,taskid,msglen,rc;
   int inmsg,outmsg;

   mpc_environ(&numtask,&taskid);

   outmsg = taskid;
   msglen = sizeof(int);

   rc = mpc_prefix(&outmsg,&inmsg,msglen,i_vadd,ALLGRP);
   if(rc)  return(-1);  /* error */

   /* "inmsg" for tasks {0,1,2,3} is now {0,1,3,6}, respectively,
      assuming a partition of 4 tasks  */

   return(0);
}
```

FORTRAN Example

```
      SUBROUTINE PREFIX_EXAMPLE

      INTEGER NUMTASK,TASKID,MSGLEN,ALLGRP
      INTEGER INMSG,OUTMSG,NBUF(4)
      EXTERNAL I_VADD

      CALL MP_ENVIRON(NUMTASK,TASKID)
      CALL MP_TASK_QUERY(NBUF,4,3)    !  get "ALLGRP" value
      ALLGRP = NBUF(4)

      OUTMSG = TASKID
      MSGLEN = 4

      CALL MP_PREFIX(OUTMSG,INMSG,MSGLEN,I_VADD,ALLGRP)

c     "INMSG" for tasks {0,1,2,3} is now {0,1,3,6}, respectively,
c     assuming a partition of 4 tasks

      END
```

RELATED INFORMATION

Subroutines: MP_REDUCE(3), mpc_reduce(3), MP_COMBINE(3), mpc_combine(3), i_vadd(3), s_vadd(3), d_vadd(3), i_vmul(3), s_vmul(3), d_vmul(3), i_vmax(3), s_vmax(3), d_vmax(3), i_vmin(3), s_vmin(3), d_vmin(3), b_vand(3), b_vor(3), b_vxor(3), l_vand(3), l_vor(3)

See the user-defined reduction functions listed in Appendix D, “Reduction Operations” on page 115.

NAME

MP_PROBE, mpc_probe – Checks for messages which may have arrived without matching a pending receive.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mpproto.h>
int mpc_probe(int *source, int *type, int *mstat);
```

FORTRAN SYNOPSIS

```
MP_PROBE(INTEGER SOURCE, INTEGER TYPE, INTEGER MSTAT)
```

PARAMETERS

source an integer specifying the task from which the probed message has originated. If **source**=**dontcare** (a system-specific flag), a message satisfying the **type** parameter will be selected, and the sending task will be returned in **source**.

type an integer specifying the type of the message being probed. If **type**=**dontcare** (a system-specific flag), a message satisfying the **source** parameter will be selected, and the message type will be returned in **type**. If both **source**=**dontcare** and **type**=**dontcare**, the first available message will be selected, and the source task and type returned in **source** and **type**, respectively. Resetting **source** and **type** to **dontcare** and calling **MP_PROBE** again without posting any new message receives will return information on the same incoming message, not a different one.

mstat an integer which returns the message status or number of bytes for a message, satisfying the **source** and **type** parameters. If no matching message is found, **mstat** is returned as **-1**.

DESCRIPTION

This subroutine probes to see if any messages have arrived for which corresponding receives have not been previously posted.

MP_PROBE checks to see if a message that satisfies the **source** and **type** parameters and for which a corresponding receive has not been posted has arrived at the task. This message, for which a receive has not yet been posted, could have been sent using any of the send calls. **MP_PROBE** does not receive the message, it only determines that a message did arrive.

NOTES

In C, a positive value returned for **mstat** does represent a message size but it is unsafe to use a size_t parameter. This is because **mstat** must be tested for <0 and a size_t parameter is unsigned.

RETURN VALUES

In C, return code = 0 means the operation was successful. Return code = -1 means that an error occurred and the global variable **mperrno** contains the error number. This number, “nnn,” can be used to find a description of the error condition by looking for **0032-nnn** in *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

ERRORS

source is not a valid task id or wildcard value

type is not a valid message type or wildcard value

EXAMPLES

A message is sent from task 0 to task 1. Task 1 probes for any incoming message and receives the message into a temporary buffer.

C Example

```
   #include <mpproto.h>

int probe_example()
{

   int source,dest,type,msglen,mstat,rc;
   int numtask,taskid,*inmsg,outmsg[100];

   rc = mpc_environ(&numtask,&taskid);

   if(taskid == 0)  {
      /* send message to task "1" */
      dest = 1;
      type = 1;
      outmsg[0] = 700;
      msglen = sizeof(outmsg);
      rc = mpc_bsend(&outmsg,msglen,dest,type);
      if(rc)  return(-1);
   } else {
      if(taskid == 1) {
         source = DONTCARE;
         type   = DONTCARE;
         for(;;)  {
            rc = mpc_probe(&source,&type,&mstat);
            if(rc)  return(-1);
            if(mstat >= 0)  break;

         /* Other processing can be done while awaiting message arrival */

         }
         /* "mstat" is now equal to the size of the probed message,
```

```
            "source" is 0, and "type" is 1 */

         inmsg = (int *) malloc(mstat);
         msglen = mstat;
         rc = mpc_brecv(inmsg,msglen,&source,&type,&mstat);
         if(rc)  return(-1);

         /* "inmsg" now holds the probed message (inmsg[0] = 700) */

         free(inmsg);
      }
   }
   return(0);
}
```

FORTRAN Example

```
      SUBROUTINE PROBE_EXAMPLE

      INTEGER SOURCE,DEST,TYPE,MSGLEN,MSTAT,DONTCARE
      INTEGER NBUF(4),NUMTASK,TASKID,INMSG(500),OUTMSG(100)

      CALL MP_ENVIRON(NUMTASK,TASKID)
      CALL MP_TASK_QUERY(NBUF,4,3)         ! get DONTCARE value
      DONTCARE = NBUF(1)

      IF(TASKID .EQ. 0)  THEN
         ! Send message to task "1"

         DEST = 1
         TYPE = 1
         OUTMSG(1) = 700
         MSGLEN = 400
         CALL MP_BSEND(OUTMSG,MSGLEN,DEST,TYPE)
      ELSE
         IF(TASKID .EQ. 1)  THEN
            SOURCE = DONTCARE
            TYPE   = DONTCARE
            MSTAT = -1
            DO 10 WHILE (MSTAT .LT. 0)
               CALL MP_PROBE(source,type,mstat)

               ! Other processing can be done while awaiting message arrival

10          CONTINUE

         !  "MSTAT" is now equal to the size of the probed message,
         !  "SOURCE" is 0, and "TYPE" is 1

            MSGLEN = MSTAT
            CALL MP_BRECV(INMSG,MSGLEN,SOURCE,TYPE,MSTAT)

         !  "inmsg" now holds the probed message (inmsg(1) = 700)

         ENDIF
      ENDIF
      END
```

NAME

MP_RCVNCALL, mpc_rcvncall – Posts a receive buffer and a handler function for a message and returns without waiting for the message arrival.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mpproto.h>
int mpc_rcvncall(void *inmsg, size_t msglen, int *source, int *type,
                 int *msgid, void (func*)());
```

FORTRAN SYNOPSIS

```
MP_RCVNCALL(INTEGER INMSG(*), INTEGER MSGLEN, INTEGER SOURCE, INTEGER TYPE,
            INTEGER MSGID, EXTERNAL FUNC)
```

PARAMETERS

inmsg is the application buffer into which the message is received.

msglen is an integer that specifies the size, in bytes, of the buffer **inmsg**.

source is an integer that specifies the task from which the message originated. If **source**=**dontcare**, the next message satisfying the **type** parameter is read. MP_RCVNCALL does not return **dontcare** overwritten by taskid for the same reason it does not return a completed **inmsg**. MP_RCVNCALL retains the location of the **source** datum and asynchronously overwrites **dontcare** when the message arrives. This requires that any variable used to pass **dontcare** exist until after the corresponding MP_WAIT returns.

type is an integer that specifies the type of the message being received. If **type**=**dontcare**, the next message satisfying the **source** parameter is read and the message type is asynchronously reported in **type**. If both **source**=**dontcare** and **type**=**dontcare**, the next message is read, and the source task and type are asynchronously reported in **source** and **type**, respectively.

msgid This requires that any variable used to pass dontcare exist until after the corresponding MP_WAIT returns. is an integer returned as a message identifier that can be used to test the status of the message.

func is a user provided message handler function which will be invoked when the message arrives. The handler will be given the **msgid** as its only parameter.

DESCRIPTION

This routine identifies an area in memory to serve as a buffer for the message to be received and a handler to be invoked asynchronously when the message arrives. MP_RCVNCALL continues processing without waiting for the message to arrive. An application should avoid MP_RCVNCALL unless it is vital to the algorithm. Once MPL detects an MP_RCVNCALL it does additional work on every MPL call to protect itself against an illtimed handler activation. MP_RCVNCALL also opens the door to a variety of application errors which can be difficult to debug.

The handler is a function provided by the application. It will be invoked when the message has arrived and its single parameter is the message id. (**Note:** For a handler written in C the parameter is a pointer to the msgid.) A handler must abide by all the restrictions which would apply to any AIX signal handler.

The normal use of MP_RCVNCALL will depend on the handler to deal with the message whenever it may arrive. The handler must do an MP_WAIT on the msgid but this wait is guaranteed not to block because the message in known to be in. The handler routine may use only certain message passing routines (MP_BRECV, MP_BSEND, MP_BSENDRECV, MP_SEND, MP_RECV, MP_STATUS, MP_PROBE, MP_STOPALL, MP_TASK_QUERY, MP_WAIT, MP_PACK, MP_UNPACK, MP_ENVIRON, MP_GETLABEL, MP_GETMEMBERS, MP_GETRANK, MP_GETSIZE, MP_GETTASKID). It may not use collective communication or any of (MP_LOCKRNC, MP_TASK_SET, MP_GROUP, MP_PARTITION, MP_BVRECV MP_BVSEND). It can assume no additional handler will be dispatched while it is active.

The application is able to suspend handler dispatching using MP_LOCKRNC and in this situation a posted MP_RCVNCALL can be completed by an MP_STATUS or MP_WAIT which is not part of a handler. If an MP_RCVNCALL is completed this way, the handler will not be dispatched when the lock is freed.

NOTES

MP_RCVNCALL is most effective in interrupt mode which is enabled by exporting "MP_CSS_INTERRUPT=YES" before a job is started. Without interrupt mode the time delay between message arrival and handler dispatching may be excessive.

MP_RCVNCALL behaves like a very limited form of threads. An application which uses it must deal with many thread safety concerns. Data which is shared between mainline code and handler code must be managed with the understanding that handlers are asynchronously dispatched. Nontrivial loads and stores in mainline code which involve such data cannot be regarded as atomic unless the section of code is protected by calls to MP_LOCKRNC. Also, routines in libraries available for AIX are not guarenteed reentrant because AIX 3.2 does not support threads. It is unsafe for a handler to call any routine which is not reentrant unless it can be guarenteed that the handler will never be dispatched while the routine is active on the main thread.

No handler will be dispatched while another handler is active so MP_LOCKRNC is neither needed nor permitted within handler code.

Handler dispatching can only occur while the mainline code remains alive. Any MP_RCVNCALL handler still pending when the mainline terminates will be lost. A pending MP_RCVNCALL will not keep the process alive.

RETURN VALUES

In C, return code = 0 means the operation was successful. Return code = -1 means that an error occurred and the global variable **mperrno** contains the error number. This number, “nnn,” can be used to find a description of the error condition by looking for **0032-nnn** in *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

ERRORS

inmsg is not a valid buffer

msglen is less than zero

source is not a valid task id or wildcard value

type is not a valid message type or wildcard value

If the message truncation mode is not enabled and the length of the message to be received is greater than **msglen**. For information on enabling the message truncation mode, see MP_TASK_SET.

EXAMPLES

Task 0 is a client with questions and task 1 is a server maintaining up to date answers. The server task is doing work some part of which updates "answers". The server is not concerned with when the client asks a question but must be prepared to answer whenever asked. In this example, a "client" keeps asking for updates until the "server" responds that it is done generating new "answers".

C Example

```
   #include <mpproto.h>

/* Some server variables must be global because both the
   rcvncall_example() and the handler() routines need
   access to them.  They also must exist beyond the
   rcvncall_example() routine's active lifetime.              */

int answers[2];
int question,msgid,client;

void handler(int *inmsg)
{
   int rc,nbytes,type;

   rc = mpc_wait(inmsg,&nbytes);
   if(rc)  mpc_stopall(1);  /* error */
      /* Now we know "question" is available. */
   if (question != 1) {
     /* print error "Sorry, I only answer question 1". */
     mpc_stopall(2);
   }
   type = 1;
   rc = mpc_bsend(answers,sizeof(answers),client,type);
   if(rc)  mpc_stopall(1);  /* error */

   /* Question has been answered.  Set up to take another one. */

   rc = mpc_rcvncall(&question,sizeof(question),&client,
```

```
                    &type,&msgid,handler);
   if(rc)  mpc_stopall(1);  /* error */
}

int rcvncall_example()
{
   int client,server,type,msgid,nbytes,rc,i;
   int numtask,taskid,thistime;
   int newlock,oldlock,junk;

   mpc_environ(&numtask,&taskid);

   type = 1;
   client = 0;
   server = 1;

   if (taskid == client)  {
      for (answers[0]; answers[0] != -1; ) {

        /* Do some work then query "server" for some more or less
           up to date info. Eventually, "client" gets an answer
           showing "server" has finished its loop. At that point,
           "client" can exit this loop and quit asking questions.   */

        question = 1;
        rc = mpc_bsend(&question,sizeof(question),server,type);
        if(rc)  return(-1);  /* error */
        rc = mpc_brecv(answers,sizeof(answers),&server,&type,&nbytes);
        if(rc)  return(-1);  /* error */
        if (answers[0] != -1) {
          /* "client" now has the value of "thistime" as of a
             known recent iteration of the loop in "server".
             It can do something with the info and continue
             iterating this client task loop. */
        }
      }
   } else {
      if (taskid == server) {
         answers[0] = 0;
         answers[1] = 0;
         rc = mpc_rcvncall(&question,sizeof(question),&client,
                     &type,&msgid,handler);
         if(rc)  return(-1);  /* error */

         /*  Data in "question" is not available until the handler
             is dispatched and does an "mpc_wait" on the msgid.  This
             next loop does the real work of "server" which "client"
             may want to ask about at any point.  "answers" is
             updated atomically in this loop so "client" will always
             get a coherent answer.                                  */

         for (i=0; i<1000000; i++) {

           /* do some work for iteration "i" which does lots of
              important work and also produces some value in the
              variable "thistime" which could interest the client.    */

           newlock = 1;
```

```
            mpc_lockrnc(newlock, &oldlock);
            answers[0] = i;
            answers[1] = thistime;
            mpc_lockrnc(oldlock, &junk);
          }

         /* "server" is done updating "answers" so we want to catch
            the "client's" next question outside of the handler and
            respond with a shutdown flag as the answer this time.  If
            we know that whatever code this routine returns to will
            still be running on the "server" task when the "client"
            next asks we could simply leave the shutdown flag in
            answers[0] and let the next handler activation
            deliver it.  In that case the handler should refrain from
            posting another mpc_rcvncall when it sends a shutdown.     */

         newlock = 1;
         mpc_lockrnc(newlock, &oldlock);
         rc = mpc_wait(&msgid,&nbytes);
         if(rc)  mpc_stopall(1);  /* error */
         answers[0] = -1;  /* The shutdown code for client */
         type = 1;
         rc = mpc_bsend(answers,sizeof(answers),client,type);
         if(rc)  mpc_stopall(-1);  /* error */
         mpc_lockrnc(oldlock, &junk);
       }
    }
    return(0);
}
```

FORTRAN Example

```
c  Some server variables most be in common because both the
c  RCVNCALL_EXAMPLE() and the HANDLER() routines need
c  access to them.  They also must exist beyond the
c  RCVNCALL_EXAMPLE() routine's active lifetime.              */

      SUBROUTINE HANDLER(INMSG)

      INTEGER ANSWERS(2),QUESTION,MSGID,CLIENT
      COMMON/RCVARS/ANSWERS,QUESTION,MSGID,CLIENT
      INTEGER RC,NBYTES,TYPE,MSGLEN

      CALL MP_WAIT(INMSG,NBYTES)
c        Now we know "QUESTION" is available.
      IF (QUESTION .NE. 1) THEN
c       print error "Sorry, I only answer question 1".
        CALL MP_STOPALL(2)
      ENDIF
      TYPE = 1
      MSGLEN = 8
      CALL MP_BSEND(ANSWERS,MSGLEN,CLIENT,TYPE)

c       QUESTION has been answered. Set up to take another one.
c       The REREGISTER routine is used because FORTRAN will not
c       let the name HANDLER be registered within the subroutine
c       named HANDLER.  The RECUR compiler option does not help.
```

```
      CALL REREGISTER

      END

      SUBROUTINE REREGISTER
      INTEGER MSGLEN,TYPE
      EXTERNAL HANDLER
      INTEGER ANSWERS(2),QUESTION,MSGID,CLIENT
      COMMON/RCVARS/ANSWERS,QUESTION,MSGID,CLIENT

      TYPE = 1
      MSGLEN = 4
      CALL MP_RCVNCALL(QUESTION,MSGLEN,CLIENT,TYPE,MSGID,HANDLER)
      END

      SUBROUTINE RCVNCALL_EXAMPLE

      EXTERNAL HANDLER
      INTEGER ANSWERS(2),QUESTION,MSGID,CLIENT
      COMMON/RCVARS/ANSWERS,QUESTION,MSGID,CLIENT
      INTEGER SERVER,TYPE,MSGLEN,NBYTES,RC,I
      INTEGER NUMTASK,TASKID,THISTIME
      INTEGER NEWLOCK,OLDLOCK,JUNK

      CALL MP_ENVIRON(NUMTASK,TASKID)

      TYPE = 1
      CLIENT = 0
      SERVER = 1

      IF(TASKID .EQ. CLIENT)  THEN
        ANSWERS(1) = 0
        DO 10 WHILE (ANSWERS(1) .NE. -1)
c          Do some work then query "server" for some more or less
c          up to date info. Eventually, "client" gets an answer
c          showing "server" has finished its loop. At that point,
c          "client" can exit this one and quit asking questions.
           MSGLEN = 4
           QUESTION = 1
           CALL MP_BSEND(QUESTION,MSGLEN,SERVER,TYPE)
           MSGLEN = 8
           CALL MP_BRECV(ANSWERS,MSGLEN,SERVER,TYPE,NBYTES)
           IF (ANSWERS(1) .NE. -1) THEN
c            "client" now has the value of "thistime" as of a
c            known recent iteration of the loop in "server".
c            It can do something with the info and continue
c            iterating this client task loop.
           ENDIF
10      ENDDO

      ELSE
         IF (TASKID .EQ. SERVER)  THEN
           ANSWERS(1) = 0
           ANSWERS(2) = 0
           MSGLEN = 4
           CALL MP_RCVNCALL(QUESTION,MSGLEN,CLIENT,TYPE,MSGID,HANDLER)
```

```
c          Data in "QUESTION" is not available until the handler is
c          is dispatched and does an "MP_WAIT" on the msgid.  This
c          next loop does the real work of "SERVER" which "CLIENT"
c          may want to ask about at any point.  "ANSWERS" is
c          updated atomically in this loop so "CLIENT" will always
c          get a coherent answer.

           DO 20 I=1,1000000
c            do some work for iteration "I" which does lots of
c            important work and also results in some value in the
c            variable "THISTIME" which could interest the client.

             NEWLOCK = 1
             CALL MP_LOCKRNC(NEWLOCK,OLDLOCK)
             ANSWERS(1) = I
             ANSWERS(2) = THISTIME
             CALL MP_LOCKRNC(OLDLOCK,JUNK)
20         ENDDO

c         "SERVER" is done updating "ANSWERS" so we want to catch
c         the "CLIENT's" next question outside of the handler and
c         respond with a shutdown flag as the answer this time.  If
c         we know that whatever code this routine returns to will
c         still be running on the "SERVER" task when the "CLIENT"
c         next asks we could simply leave the shutdown flag in
c         ANSWERS(1) and let the next handler activation deliver
c         it.  In that case the handler should refrain from posting
c         posting another MP_RCVNCALL when it sends a shutdown.

            NEWLOCK = 1
            CALL MP_LOCKRNC(NEWLOCK,OLDLOCK)
            CALL MP_WAIT(MSGID,NBYTES)
            ANSWERS(1) = -1
            TYPE = 1
            MSGLEN = 8
            CALL MP_BSEND(ANSWERS,MSGLEN,CLIENT,TYPE)
            CALL MP_LOCKRNC(OLDLOCK,JUNK)
          ENDIF
        ENDIF
        END
```

RELATED INFORMATION

Subroutines: MP_RECV(3), mpc_recv(3) MP_LOCKRNC(3), mpc_lockrnc(3), MP_WAIT(3), mpc_wait(3), MP_STATUS(3), mpc_status(3),

NAME

MP_RECV, mpc_recv – Posts a receive buffer for a message and returns without waiting for the message arrival.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mpproto.h>
int mpc_recv(void *inmsg, size_t msglen, int *source, int *type, int *msgid);
```

FORTRAN SYNOPSIS

```
MP_RECV(INTEGER INMSG(*), INTEGER MSGLEN, INTEGER SOURCE, INTEGER TYPE,
        INTEGER MSGID)
```

PARAMETERS

inmsg is the application buffer into which the message is received.

msglen is an integer that specifies the size, in bytes, of the buffer **inmsg**.

source is an integer that specifies the task from which the message originated. If **source**=**dontcare**, the next message satisfying the **type** parameter is read. MP_RECV does not return **dontcare** overwritten by taskid for the same reason it does not return a completed **inmsg**. MP_RECV retains the location of the **source** datum and asynchronously overwrites **dontcare** when the message arrives. This requires that any variable used to pass **dontcare** exist until after the corresponding MP_WAIT returns.

type is an integer that specifies the type of the message being received. If **type**=**dontcare**, the next message satisfying the **source** parameter is read and the message type is asynchronously reported in **type**. If both **source**=**dontcare** and **type**=**dontcare**, the next message is read, and the source task and type are asynchronously reported in **source** and **type**, respectively. This requires that any variable used to pass **dontcare** exist until after the corresponding MP_WAIT returns.

msgid is an integer returned as a message identifier that can be used to test the status of the message.

DESCRIPTION

Identifies an area in memory to serve as a buffer for the message to be received and continues processing without waiting for the message to arrive.

Use calls to MP_STATUS or MP_WAIT to determine when the MP_RECV operation completes and the requested message is available in the buffer **inmsg**.

RETURN VALUES

In C, return code = 0 means the operation was successful. Return code = -1 means that an error occurred and the global variable **mperrno** contains the error number. This number, "nnn," can be used to find a description of the error condition by looking for **0032-nnn** in *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

ERRORS

inmsg is not a valid buffer

msglen is less than zero

source is not a valid task id or wildcard value

type is not a valid message type or wildcard value

If the message truncation mode is not enabled and the length of the message to be received is greater than **msglen**. For information on enabling the message truncation mode, see MP_TASK_SET.

EXAMPLES

A message is sent from task 0 using MP_SEND to task 1, where it is received using MP_RECV. Task 1 then calls MP_WAIT before using the data in **inmsg**.

C Example

```
   #include <mpproto.h>

int recv_example()
{
   int source,dest,type,msglen,msgid,nbytes,rc;
   int numtask,taskid,inmsg[2],outmsg[2];

   mpc_environ(&numtask,&taskid);

   type = 1;
   source = 0;
   dest = 1;
   msglen = sizeof(outmsg);

   if(taskid == source)  {
      outmsg[0] = 150;
      outmsg[1] = 250;
      rc = mpc_bsend(outmsg,msglen,dest,type);
      if(rc)  return(-1);  /* error */
   } else {
      if(taskid == dest) {
         rc = mpc_recv(inmsg,msglen,&source,&type,&msgid);
         if(rc)  return(-1);  /* error */

         /*  Data in "inmsg" is not available until "mpc_wait" is executed,
             but other work can continue until it is needed  */

         rc = mpc_wait(&msgid,&nbytes);
         if(rc)  return(-1);  /* error */
```

```
         /*  Task 1 now has "inmsg" = {150,250}  */
      }
   }
   return(0);
}
```

FORTRAN Example

```
      SUBROUTINE RECV_EXAMPLE

      INTEGER SOURCE,DEST,TYPE,MSGLEN,MSGID,NBYTES
      INTEGER NUMTASK,TASKID,INMSG(2),OUTMSG(2)

      CALL MP_ENVIRON(NUMTASK,TASKID)

      TYPE = 1
      SOURCE = 0
      DEST = 1
      MSGLEN = 8

      IF(TASKID .EQ. SOURCE)  THEN
         OUTMSG(1) = 150
         OUTMSG(2) = 250
         CALL MP_BSEND(OUTMSG,MSGLEN,DEST,TYPE)
      ELSE
         IF(TASKID .EQ. DEST)  THEN

            CALL MP_RECV(INMSG,MSGLEN,SOURCE,TYPE,MSGID)

c           Data in "INMSG" is not available until "MP_WAIT" is executed,
c           but other work can continue until it is needed

            CALL MP_WAIT(MSGID,NBYTES)

c           Task 1 now has "INMSG" = {150,250}
c           "NBYTES" returns the actual number of bytes sent.

         ENDIF
      ENDIF
      END
```

RELATED INFORMATION

Subroutines: MP_SEND(3), mpc_send(3), MP_WAIT(3), mpc_wait(3), MP_STATUS(3), mpc_status(3), MP_PROBE(3), mpc_probe(3), MP_BRECV(3), mpc_brecv(3)

NAME

MP_REDUCE, mpc_reduce – Performs a reduction operation on data distributed across a group.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mpproto.h>
int mpc_reduce(const void *outmsg, void *inmsg, size_t msglen, int dest,
               void (*func)(), int gid);
```

FORTRAN SYNOPSIS

```
MP_REDUCE(INTEGER OUTMSG(*), INTEGER INMSG(*), INTEGER MSGLEN, INTEGER DEST,
          EXTERN FUNC, INTEGER GID)
```

PARAMETERS

outmsg is the application buffer that contains the local share of the distributed data.

inmsg is the application buffer into which the result of the reduction operation is stored, for task id equal to **dest**. For the other tasks in the group, **inmsg** is used as scratch buffer space by the system.

msglen is a non-negative integer that specifies the size, in bytes, of **outmsg** and **inmsg**.

dest is an integer that specifies the task to which the result of a reduction operation across a group is written.

func is an external function which implements a reduction operation.

gid is a group identifier for the group of participating tasks.

DESCRIPTION

Applies a reduction operation on data distributed across a group and places the result in a designated **inmsg** of a specified task in the group. The local data set to be reduced is called **outmsg** and is of length **msglen** bytes.

The reduction is applied to one vector per task and the result is also a vector of the same length.

You can define the reduction function or you can use one of the predefined ones listed in Appendix D, “Reduction Operations” on page 115

Only the task **dest** in the group receives the final result of a reduction operation, which will be stored in the application buffer **inmsg**. **outmsg** and **inmsg** cannot overlap in memory.

RETURN VALUES

In C, return code = 0 means the operation was successful. Return code = -1 means that an error occurred and the global variable **mperrno** contains the error number. This number, "nnn," can be used to find a description of the error condition by looking for **0032-nnn** in *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

ERRORS

outmsg *or* **inmsg**	is not a valid buffer
msglen	is less than zero
dest	is not a valid task id or is not a member of **gid**
gid	is undefined in the calling task

Develop mode error if:

- all tasks in **gid** do not use identical values for **dest** and **msglen**

EXAMPLES

In a partition of 4 tasks, MP_REDUCE is used to find the sum of the **outmsg** values from all of the tasks, and places it in **inmsg** on task 0.

C Example

```
   #include <mpproto.h>
   /*  extern void i_vadd() is declared by mpproto.h  */
   /*  The constant ALLGRP is defined by mpproto.h   */

int reduce_example()
{
   int numtask,taskid,dest,msglen,rc;
   int inmsg,outmsg;

   mpc_environ(&numtask,&taskid);

   dest = 0;
   outmsg = taskid;
   /* "outmsg" for tasks {0,1,2,3} is now {0,1,2,3}, respectively */
   msglen = sizeof(int);

   rc = mpc_reduce(&outmsg,&inmsg,msglen,dest,i_vadd,ALLGRP);
   if(rc)  return(-1);  /* error */

   /* "inmsg" for task 0 is now 6, assuming a partition of 4 tasks.
   For tasks 1, 2, and 3, the value of "inmsg" is undefined.  */

   return(0);
}
```

FORTRAN Example

```
      SUBROUTINE REDUCE_EXAMPLE

      INTEGER NUMTASK,TASKID,DEST,MSGLEN,ALLGRP
      INTEGER INMSG,OUTMSG,NBUF(4)
      EXTERNAL I_VADD

      CALL MP_ENVIRON(NUMTASK,TASKID)
      CALL MP_TASK_QUERY(NBUF,4,3)    !  get "ALLGRP" value
      ALLGRP = NBUF(4)

      DEST = 0
      OUTMSG = TASKID
c     "OUTMSG" for tasks {0,1,2,3} is now {0,1,2,3}, respectively
      MSGLEN = 4

      CALL MP_REDUCE(OUTMSG,INMSG,MSGLEN,DEST,I_VADD,ALLGRP)

c     "INMSG" for task 0 is now 6, assuming a partition of 4 tasks.
c     For tasks 1, 2, and 3, the value of "inmsg" is undefined.

      END
```

RELATED INFORMATION

Subroutines: MP_COMBINE(3), mpc_combine(3), MP_PREFIX(3), mpc_prefix(3), i_vadd(3), s_vadd(3), d_vadd(3), i_vmul(3), s_vmul(3), d_vmul(3), i_vmax(3), s_vmax(3), d_vmax(3), i_vmin(3), s_vmin(3), d_vmin(3), b_vand(3), b_vor(3), b_vxor(3), l_vand(3), l_vor(3)

See the user-defined reduction functions listed in Appendix D, “Reduction Operations” on page 115.

NAME

MP_SCATTER, mpc_scatter – Distributes distinct messages to each task in the group.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mpproto.h>
int mpc_scatter(void *outmsg, void *inmsg, size_t blklen,
                int source, int gid);
```

FORTRAN SYNOPSIS

```
MP_SCATTER(INTEGER OUTMSG(*), INTEGER INMSG(*), INTEGER BLKLEN,
           INTEGER SOURCE, INTEGER GID)
```

PARAMETERS

outmsg	is the application buffer that contains the messages to scatter for task id equal to **source**. For this task, **outmsg** is not changed but for the other tasks, **outmsg** is used as a scratch space by the system.
inmsg	is the application buffer into which the incoming message is received.
blklen	is a non-negative integer that specifies the size, in bytes, of **inmsg**.
source	is an integer that specifies the task that contains the distinct messages to scatter.
gid	is a group identifier for the group of participating tasks.

DESCRIPTION

This routine distributes distinct messages to each task in the same group **gid**. Conceptually, **outmsg** in the task with task id **source** is partitioned into *N* blocks, each of length **blklen** bytes, where *N* is the number of tasks in the group. Each block is a distinct message. As a result of this call, the *i*-th block is sent to the *i*-th task in the group and stored in **inmsg**.

outmsg and **inmsg** cannot overlap in memory.

RETURN VALUES

In C, return code = 0 means the operation was successful. Return code = -1 means that an error occurred and the global variable **mperrno** contains the error number. This number, “nnn,” can be used to find a description of the error condition by looking for **0032-nnn** in *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

ERRORS

outmsg *or* **inmsg**	is not a valid buffer
blklen	is less than zero
source	is not a valid task id or is not a member of **gid**
gid	is undefined in the calling task

Develop mode error if:

- all tasks in **gid** do not use identical values for **source** and **blklen**

EXAMPLES

In a partition of 4 tasks, MP_SCATTER is used to distribute distinct messages from task 2 to all of the tasks in the partition. The data is sent from **outmsg** and is received in **inmsg**.

C Example

```
   #include <mpproto.h>

int scatter_example()
{
   int numtask,taskid,source,blklen,i,rc;
   int inmsg,outmsg[4];

   mpc_environ(&numtask,&taskid);

   blklen = sizeof(int);
   source = 2;
   if(taskid == source)
      for(i=0;i<numtask;i++)
         outmsg[i] = i;
   /* "outmsg" on task 2 is now {0,1,2,3} */

   rc = mpc_scatter(outmsg,&inmsg,blklen,source,ALLGRP);
   if(rc)  return(-1);  /* error */

   /* "inmsg" on tasks {0,1,2,3} is now {0,1,2,3}, respectively,
   assuming a partition of 4 tasks  */

   return(0);
}
```

FORTRAN Example

```
      SUBROUTINE SCATTER_EXAMPLE

      INTEGER NUMTASK,TASKID,SOURCE,BLKLEN,ALLGRP,I
      INTEGER INMSG,OUTMSG(4),NBUF(4)

      CALL MP_ENVIRON(NUMTASK,TASKID)
      CALL MP_TASK_QUERY(NBUF,4,3)       !  get "ALLGRP" value
      ALLGRP = NBUF(4)

      BLKLEN = 4
      SOURCE = 2
      IF(TASKID .EQ. SOURCE)  THEN
         DO 10 I=1,NUMTASK
   10       OUTMSG(I) = I - 1
      ENDIF
c     "OUTMSG" on task 2 is now {0,1,2,3}

      CALL MP_SCATTER(OUTMSG,INMSG,BLKLEN,SOURCE,ALLGRP)

c     "INMSG" on tasks {0,1,2,3} is now {0,1,2,3}, respectively,
c     assuming a partition of 4 tasks

      END
```

RELATED INFORMATION

Subroutines: MP_GATHER(3), mpc_gather(3)

NAME

MP_SEND, mpc_send – Identifies a message to be sent and returns without waiting for the send to complete.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mpproto.h>
int mpc_send(const void *outmsg, size_t msglen, int dest, int type,
             int *msgid);
```

FORTRAN SYNOPSIS

```
MP_SEND(INTEGER OUTMSG(*), INTEGER MSGLEN, INTEGER DEST, INTEGER TYPE,
        INTEGER MSGID)
```

PARAMETERS

outmsg is the application buffer that contains the message to send.

msglen is a non-negative integer that specifies the size, in bytes, of the message to be sent.

dest is an integer that specifies the task to which the message is to be sent.

type is an integer that specifies the type of the message to be sent.

msgid is an integer returned as a message identifier that can be used to test the status of the message.

DESCRIPTION

This subroutine identifies a message to be sent and returns without waiting for the send operation to complete. Do not modify the application buffer, **outmsg**, until subsequent calls to MP_STATUS or MP_WAIT indicate completion of MP_SEND.

RETURN VALUES

In C, return code = 0 means the operation was successful. Return code = -1 means that an error occurred and the global variable **mperrno** contains the error number. This number, “nnn,” can be used to find a description of the error condition by looking for **0032-nnn** in *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

ERRORS

outmsg is not a valid buffer

msglen is less than zero

dest is not a valid task id

type is not a valid message type

EXAMPLES

A message is sent from task 0 to task 1 using MP_SEND. Task 0 then calls MP_WAIT before modifying the data in the send buffer.

C Example

```
   #include <mpproto.h>

int send_example()
{
   int source,dest,type,msglen,msgid,nbytes,rc;
   int numtask,taskid,inmsg[2],outmsg[2];

   mpc_environ(&numtask,&taskid);

   type = 1;
   source = 0;
   dest = 1;
   msglen = sizeof(outmsg);

   if(taskid == source)  {
      outmsg[0] = 150;
      outmsg[1] = 250;
      rc = mpc_send(outmsg,msglen,dest,type,&msgid);
      if(rc)  return(-1);  /* error */

      /* Data in "outmsg" must not be modified until "mpc_wait"
      is called; meanwhile, other work can continue.  */

      rc = mpc_wait(&msgid,&nbytes);
      if(rc)  return(-1);  /* error */
      /* "outmsg" is now available for reuse */
   } else {
      if(taskid == dest) {
         rc = mpc_brecv(inmsg,msglen,&source,&type,&nbytes);
         if(rc)  return(-1);  /* error */

         /*  Task 1 now has "inmsg" = {150,250}  */
      }
   }
   return(0);
}
```

FORTRAN Example

```
      SUBROUTINE SEND_EXAMPLE

      INTEGER SOURCE,DEST,TYPE,MSGLEN,MSGID,NBYTES
      INTEGER NUMTASK,TASKID,INMSG(2),OUTMSG(2)

      CALL MP_ENVIRON(NUMTASK,TASKID)

      TYPE = 1
      SOURCE = 0
      DEST = 1
      MSGLEN = 8

      IF(TASKID .EQ. SOURCE)  THEN
         OUTMSG(1) = 150
         OUTMSG(2) = 250

         CALL MP_SEND(OUTMSG,MSGLEN,DEST,TYPE,MSGID)

c        Data in "OUTMSG" must not be modified until "MP_WAIT" is
c        called; meanwhile, other work can continue.

         CALL MP_WAIT(MSGID,NBYTES)
c        "OUTMSG" is now available for reuse
      ELSE
         IF(TASKID .EQ. DEST)  THEN
            CALL MP_BRECV(INMSG,MSGLEN,SOURCE,TYPE,NBYTES)

c           Task 1 now has "INMSG" = {150,250}

         ENDIF
      ENDIF
      END
```

RELATED INFORMATION

Subroutines: MP_RECV(3), mpc_recv(3), MP_BSEND(3), mpc_bsend(3), MP_WAIT(3), mpc_wait(3), MP_STATUS(3), mpc_status(3)

NAME

MP_SHIFT, mpc_shift – Performs linear or circular data shifts across a group.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mpproto.h>
int mpc_shift(const void *outmsg, void *inmsg, size_t msglen, int step,
              int flag, int gid);
```

FORTRAN SYNOPSIS

```
MP_SHIFT(INTEGER OUTMSG(*), INTEGER INMSG(*), INTEGER MSGLEN, INTEGER STEP,
         INTEGER FLAG, INTEGER GID)
```

PARAMETERS

outmsg	is the application buffer that contains the outgoing message.
inmsg	is the application buffer into which the incoming message is received.
msglen	is a non-negative integer that specifies the size, in bytes, of **outmsg** and **inmsg**.
step	is a positive or negative integer that specifies the number of steps of the up/down shift. Up and down refer to increasing or decreasing rank in the group.
flag	is an integer that specifies the type of shift. A value 0 indicates a rotation (circular shift) and all other integer values indicate a non-circular shift.
gid	is a group identifier for the group of participating tasks.

DESCRIPTION

This routine shifts or rotates any specified number of steps along the group in the "up" or "down" directions. The possible values of **flag** are:

- 0 indicates a rotation or circular shift
- All other values indicate a non-circular shift.

Circular shift means that the sequence of tasks "wrap-around." In other words, that following the highest ranked task is task 0 again. Therefore, for circular shift and a step of -1, the outgoing message from task 0 will go to the highest ranked task. For non-circular shift and a step of -1, the outgoing message from task 0 will not go to any other task. When **step** exceeds 0, **outmsg** at each task in the group shifts or rotates up (along the increasing rank direction) **step** steps into **inmsg**. (In the case of non-circular shift, the **inmsg** of some lower-ranked tasks is not over-written.) When **step** is less than 0, **outmsg** at each task in the group shifts or rotates down **-step** steps into **inmsg**. (In the case of non-circular shift, the **inmsg**

of some higher-ranked tasks will not be over-written.) When **step**=*0*, **outmsg** is copied into **inmsg** locally in all tasks in the group.

It is assumed that **step** is in the range of *-n* through *n* where "n" is the number of tasks in **gid**. When **flag** is 0, both **step**=*n* and **step**=*-n* indicate local copying. When **flag** is nonzero, both **step**=*n* and **step**=*-n* do nothing, as the **outmsg** in all tasks in the group have been shifted out of the group range.

For a non-circular shift (when **flag** does not equal 0), the data in **inmsg** in a task which does not need to receive will remain unchanged (as when it was passed from users to MP_SHIFT).

outmsg and **inmsg** cannot overlap in memory.

RETURN VALUES

In C, return code = 0 means the operation was successful. Return code = -1 means that an error occurred and the global variable **mperrno** contains the error number. This number, "nnn," can be used to find a description of the error condition by looking for **0032-nnn** in *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

ERRORS

outmsg *or* **inmsg**	is not a valid buffer
msglen	is less than zero
step	is less than -n or greater than n, where n is the of tasks in **gid**
gid	is undefined in the calling task

Develop mode error if:

- all tasks in **gid** do not use identical values for **step**, **flag**, and **msglen**

EXAMPLES

Each task in a partition of four sends a message to the next-highest task using a wrap-around or "ring" topology.

C Example

```
   #include <mpproto.h>

int shift_example()
{
   int numtask,taskid,msglen,step,flag,rc;
   int inmsg,outmsg;


   mpc_environ(&numtask,&taskid);

   msglen = sizeof(int);
   step = 1;
   flag = 0;             /* specify circular shift */
   outmsg = taskid;
   /* "outmsg" on tasks {0,1,2,3} is now {0,1,2,3}, respectively  */

   rc = mpc_shift(&outmsg,&inmsg,msglen,step,flag,ALLGRP);
   if(rc)  return(-1);  /* error */

   /* "inmsg" on tasks {0,1,2,3} is now {3,0,1,2}, respectively,
   assuming a partition of 4 tasks  */

   return(0);
}
```

FORTRAN Example

```
      SUBROUTINE SHIFT_EXAMPLE

      INTEGER NUMTASK,TASKID,BLKLEN,ALLGRP,STEP,FLAG, MSGLEN
      INTEGER INMSG,OUTMSG,NBUF(4)

      CALL MP_ENVIRON(NUMTASK,TASKID)
      CALL MP_TASK_QUERY(NBUF,4,3)       !  get "ALLGRP" value
      ALLGRP = NBUF(4)

      MSGLEN = 4
      STEP = 1
c     specify circular shift
      FLAG = 0
      OUTMSG = TASKID
c     "OUTMSG" on tasks {0,1,2,3} is now {0,1,2,3}, respectively

      CALL MP_SHIFT(OUTMSG,INMSG,MSGLEN,STEP,FLAG,ALLGRP)

c     "INMSG" on tasks {0,1,2,3} is now {3,0,1,2}, respectively,
c     assuming a partition of 4 tasks

      END
```

NAME

MP_STATUS, mpc_status – Returns status of a non-blocking send or receive.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mpproto.h>
int mpc_status(int msgid);
```

FORTRAN SYNOPSIS

```
MP_STATUS(INTEGER MSGID)
```

PARAMETERS

msgid is a message identifier returned by a previous call to a non-blocking send or receive subroutine.

DESCRIPTION

This routine provides a non-blocking check of the status of a specified non-blocking send or receive operation. A non-blocking send or receive operation does not wait for the message to be sent or received but continues processing.

MP_STATUS is a function and not a subroutine. This means that it returns the value of a single variable.

When this routine returns a non-negative number, this means that the non-blocking send or receive operation specified by the message identifier has completed. You can apply this routine to any active non-blocking send or receive operation. The message identifier **msgid** cannot be set to the wildcards **allmsg** or **dontcare**.

If you apply MP_STATUS to an inactive **msgid**, no error will result. However, the return value will indicate that the message id is not currently active.

NOTES

In FORTRAN, MP_STATUS is the only routine which is a function rather than a subroutine.

In C, a positive return value from mpc_status does represent a message size but it is unsafe to assign mpc_status to a size_t variable. This is because the return value must be tested for <0.

RETURN VALUES

MP_STATUS returns a value of -1 prior to completion of the non-blocking send or receive subroutine specified by **msgid**. Upon completion, it returns the number of bytes (after truncation, if the truncation mode is enabled) either sent or received by the operation associated with **msgid**. A return value of 0 means a completed send or receive of a 0-byte message.

MP_STATUS returns a value of -2 if **msgid** is currently inactive.

ERRORS

msgid is less than zero

EXAMPLES

A message is sent from task 0 to task 1. Task 1 receives the message using a non-blocking receive operation, and then enters a loop which uses MP_STATUS to determine when the message has arrived.

C Example

```
   #include <mpproto.h>

int status_example()
{
   int source,dest,type,msglen,msgid,mstat,rc;
   int numtask,taskid,inmsg,outmsg;

   mpc_environ(&numtask,&taskid);

   type = 1;
   source = 0;
   dest = 1;
   msglen = sizeof(int);

   if(taskid == source)  {
      outmsg = 5;
      rc = mpc_bsend(&outmsg,msglen,dest,type);
      if(rc)  return(-1);  /* error */
   } else {
      if(taskid == dest) {
         rc = mpc_recv(&inmsg,msglen,&source,&type,&msgid);
         if(rc)  return(-1);  /* error */

         for( ; ; )  {
            mstat = mpc_status(msgid);
            if(mstat < 0)  {
               if(mstat == -1)  {
                  /*  The message has not been received; until it is,
                      other work can continue.  */
               } else {
                  break;
                  /*  Invalid or inactive "msgid"  */
               }
            } else {
               break;
```

```
               /*  The message has been received in "inmsg" */
            }
         }
      }
   }
   return(0);
}
```

FORTRAN Example

```
      SUBROUTINE STATUS_EXAMPLE

      INTEGER SOURCE,DEST,TYPE,MSGLEN,MSGID,MSTAT
      INTEGER NUMTASK,TASKID,INMSG,OUTMSG

      CALL MP_ENVIRON(NUMTASK,TASKID)

      TYPE = 1
      SOURCE = 0
      DEST = 1
      MSGLEN = 4

      IF(TASKID .EQ. SOURCE)  THEN
         OUTMSG = 5
         CALL MP_BSEND(OUTMSG,MSGLEN,DEST,TYPE)
      ELSE
         IF(TASKID .EQ. DEST) THEN
            CALL MP_RECV(INMSG,MSGLEN,SOURCE,TYPE,MSGID)

            MSTAT = -1
            DO 10 WHILE (MSTAT .EQ. -1)
c              Until the message is received, other work can continue

               MSTAT = MP_STATUS(MSGID)
   10       END DO
            IF(MSTAT .GE. 0) THEN
c              The message has been received in "INMSG"
            ENDIF
         ENDIF
      ENDIF
      END
```

RELATED INFORMATION

Subroutines: MP_SEND(3), mpc_send(3), MP_RECV(3), mpc_recv(3)

NAME

MP_STOPALL, mpc_stopall – Forces an abnormal termination.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mpproto.h>
int mpc_stopall(int errcode);
```

FORTRAN SYNOPSIS

```
MP_STOPALL(INTEGER ERRCODE)
```

PARAMETERS

errcode is an integer, specified by the application, that is returned to standard error.

DESCRIPTION

When called from any task in the partition, this subroutine terminates all tasks, cancels the parallel job, and returns an abnormal termination condition along with **errcode**. Use this only for abnormal termination since other tasks are stopped regardless of whether or not they have finished. This is, however, a "protected" stop in the sense that buffers are flushed, files are closed, and so on, before each task is terminated.

A standard FORTRAN STOP anywhere or an END statement in the main program only terminates the individual calling task, not the whole job. Normal job termination occurs when every job processes a STOP or END statement.

errcode is a user-supplied integer that can be used in the same way as by programming 'STOP *digit_string*'. It is intended as a descriptor which can help to identify which call of MP_STOPALL was invoked by the program. See Appendix G, "Programming Considerations for User Applications in POE" on page 149 for more information on the values of the exit status for MP_STOPALL.

If this routine is called in multiple tasks, each executed MP_STOPALL will be reported. However, when multiple tasks are each on the way toward an MP_STOPALL, some tasks may be stopped from outside before getting a chance to call MP_STOPALL.

RETURN VALUES

In C, return code = 0 means the operation was successful. Return code = -1 means that an error occurred and the global variable **mperrno** contains the error number. This number, "nnn," can be used to find a description of the error condition by looking for **0032-nnn** in *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

EXAMPLES

MP_STOPALL is used to halt a parallel job after an error is detected.

C Example

```
   #include <mpproto.h>

int stopall_example(int code)
{
   /*  If any task detects an error, it can stop all of
       the tasks in the partition with "mpc_stopall".  */

   if(code < 0)
      mpc_stopall(code);
}
```

FORTRAN Example

In the example, MP_STOPALL is used to stop the job if any task encounters an error reading an input file.

```
      SUBROUTINE STOPALL_EXAMPLE(CODE)

      INTEGER CODE

c     If any task detects an error, it can stop all of
c     the tasks in the partition with "MP_STOPALL".

      IF(CODE .LT. 0) THEN
         CALL MP_STOPALL(CODE)
      ENDIF

      END
```

NAME

MP_SYNC, mpc_sync – Synchronizes tasks at a barrier.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mpproto.h>
int mpc_sync(int gid);
```

FORTRAN SYNOPSIS

```
MP_SYNC(INTEGER GID)
```

PARAMETERS

gid is a group identifier for the group of participating tasks.

DESCRIPTION

Implements barrier synchronization among a group of tasks. This means that all tasks within a group must reach a corresponding MP_SYNC before proceeding.

RETURN VALUES

In C, return code = 0 means the operation was successful. Return code = -1 means that an error occurred and the global variable **mperrno** contains the error number. This number, “nnn,” can be used to find a description of the error condition by looking for **0032-nnn** in *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

ERRORS

gid is undefined in the calling task

EXAMPLES

The tasks in a partition are synchronized by calling MP_SYNC.

C Example

```
   #include <mpproto.h>

int sync_example()
{
   int rc;


   /* The tasks in the partition may be executing at different
   points in the application code  */

   rc = mpc_sync(ALLGRP);
   if(rc)  return(-1);  /* error */

   /* The tasks in the partition have now been synchronized  */

   return(0);
}
```

FORTRAN Example

```
      SUBROUTINE SYNC_EXAMPLE

      INTEGER ALLGRP,NBUF(4)

      CALL MP_TASK_QUERY(NBUF,4,3)    !  get "ALLGRP" value
      ALLGRP = NBUF(4)

c     The tasks in the partition may be executing at different
c     points in the application code

      CALL MP_SYNC(ALLGRP)

c     The tasks in the partition have now been synchronized

      END
```

NAME

MP_TASK_QUERY, mpc_task_query – Returns information about system variables and constants.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mpproto.h>
int mpc_task_query(int nbuf[], int nelem, int qtype);
```

FORTRAN SYNOPSIS

```
MP_TASK_QUERY(INTEGER NBUF(*), INTEGER NELEM, INTEGER QTYPE)
```

PARAMETERS

nbuf is the integer array into which the returned information is received.

nelem is a non-negative integer that specifies the number of array elements to be filled in **nbuf**.

qtype is an integer that specifies the type of query. See Table 1 for further information.

DESCRIPTION

This routine returns information about system variables and constants. The information returned depends on **qtype**, which is set by the application. The **nelem** argument, also set by the application, gives the length (in number of elements) of the **nbuf** integer array where the returned information is received. The information returned is truncated if the **nelem** specified is less than the information usually returned for the given **qtype**.

The information returned for the given **qtype**s is described in the following table.

Table 1 (Page 1 of 2). Description of Qtypes.

qtype	Information Returned
qtype=1	Returns a flag which specifies whether messages are truncated when the length of the buffer identified for an incoming message is smaller than the incoming message. For FORTRAN, with **nelem** = 1, MP_TASK_QUERY returns the following: • **nbuf**(1)= 0 = messages are not truncated and overflows are treated as errors (default), or • **nbuf**(1)= 1 = messages are truncated. For C, mp_task_query returns the following: • **nbuf**[0]=0, messages are not truncated and overflows are treated as errors (default), or • **nbuf**[0]=1, messages are truncated.

Table 1 (Page 2 of 2). Description of Qtypes.

qtype	Information Returned
qtype=2	Returns information on message types. For FORTRAN with **nelem** = 2, MP_TASK_QUERY returns the following: • **nbuf**(1) = Lower bound for user message types • **nbuf**(2) = Upper bound for user message types For C, mp_task_query returns the following: • **nbuf**[0] = Lower bound for user message types • **nbuf**[1] = Upper bound for user message types
qtype=3	Returns information on wildcard values and on predefined group identifiers. It returns the **dontcare** value which can be used as **type** or **source** in many of the message passing subroutines to indicate that any type and/or source can satisfy the request. It returns the **allmsg** value which can be used as **msgid** in MP_WAIT to wait until all posted sends and receives are satisfied. It also returns the **nulltask** value which can be used to make part of a MP_BSENDRECV call inoperative. Additionally, it returns the **allgrp** value which can be used as a group identifier in any of the collective communication routines. For FORTRAN with **nelem** = 4, MP_TASK_QUERY returns the following: • **nbuf**(1) = **dontcare** value • **nbuf**(2) = **allmsg** value • **nbuf**(3) = **nulltask** value • **nbuf**(4) = **allgrp** value For C, mp_task_query returns the following: • **nbuf**[0] = **dontcare** value • **nbuf**[1] = **allmsg** value • **nbuf**[2] = **nulltask** value • **nbuf**[3] = **allgrp** value

RETURN VALUES

In C, return code = 0 means the operation was successful. Return code = -1 means that an error occurred and the global variable **mperrno** contains the error number. This number, “nnn,” can be used to find a description of the error condition by looking for **0032-nnn** in *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

ERRORS

nelem is less than 0

qtype is not 1, 2 or 3

EXAMPLES

MP_TASK_QUERY is used to obtain the values for the system wildcards **dontcare**, **allmsg**, **nulltask**, and **allgrp**.

C Example

```
   #include <mpproto.h>

int task_query_example()
{
   int nbuf[4],nelem,qtype,dontcare,allmsg,nulltask,allgrp,rc;

   qtype = 3;
   nelem = 4;

   rc = mpc_task_query(nbuf,nelem,qtype);    /*  Get wildcard values  */
   if(rc)  return(-1);  /* error */

   dontcare = nbuf[0];
   allmsg   = nbuf[1];
   nulltask = nbuf[2];
   allgrp   = nbuf[3];

   /* The wildcard values are now available for use  */

   return(0);
}
```

FORTRAN Example

```
      SUBROUTINE TASK_QUERY_EXAMPLE

      INTEGER NBUF(4),NELEM,QTYPE,DONTCARE,ALLMSG,NULLTASK,ALLGRP

      QTYPE = 3
      NELEM = 4

      CALL MP_TASK_QUERY(NBUF,NELEM,QTYPE)    !  get wildcard values

      DONTCARE = NBUF(1)
      ALLMSG   = NBUF(2)
      NULLTASK = NBUF(3)
      ALLGRP   = NBUF(4)

c     The wildcard values are now available for use

      END
```

NAME

MP_TASK_SET, mpc_task_set – Sets system parameters.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mpproto.h>
int mpc_task_set(const int nbuf[], int stype);
```

FORTRAN SYNOPSIS

```
MP_TASK_SET(INTEGER NBUF(*), INTEGER STYPE)
```

PARAMETERS

nbuf is the integer array that contains the parameters to be set in the calling task. Currently only the first value in the array is used.

stype is an integer that specifies which parameters are being set. See Table 2 on page 100 for further information.

DESCRIPTION

This routine lets you set system-dependent parameters for the task. **stype** specifies which parameters to set and **nbuf** contains the values to set. You may not need to use this routine because the default values for environment settings should satisfy most of your programming requirements.

The application should call this routine at the beginning of the job, before any communication routines are invoked. However, the system does not restrict when and how often this routine can be called. When called, it usually applies only to commands not yet started. However, if non-blocking communications are in progress the results will be unpredictable if you invoke this command. To be sure of predictable, repeatable behavior, this routine should be called very infrequently and only when it applies to non-active communications.

Whether the values set in **nbuf** need to be the same in every task or whether every task needs to invoke the routine simultaneously depends on what you are setting.

The following table lists the parameters you can set.

Table 2. Description of Stype.

stype	Description
1	Specifies whether messages will be truncated when the length of the application buffer posted for an incoming message is smaller than the incoming message. For FORTRAN, MP_TASK_SET interprets **nbuf** as follows: • **nbuf**(1)=0, messages will not be truncated (default). • **nbuf**(1)=1, messages will be truncated. For C, mpc_task_set interprets **nbuf** as follows: • **nbuf**[0]=0, messages will not be truncated (default). • **nbuf**[0]=1, messages will be truncated. All tasks do not have to call this or supply the same value.
2	Sets develop and run modes. In ***run*** mode, minimal checking is done without necessarily affecting performance. In ***develop*** mode, more detailed checking is done although all detectable user errors may not be detected. For FORTRAN, MP_TASK_SET interprets **nbuf** as follows: • **nbuf**(1) = 0 - Set ***run*** mode (default). • **nbuf**(1) = 1 - Set ***develop*** mode. For C, mpc_task_set interprets **nbuf** as follows: • **nbuf**[0] = 0 - Set ***run*** mode (default) • **nbuf**[0] = 1 - Set ***develop*** mode. If any task sets this mode, then all tasks must set it to the same value.
3	Specifies whether blocking sends will be buffered. If buffer space is available, messages will be copied into system buffers and control will return to the user. Otherwise, sends will block. For FORTRAN, MP_TASK_SET interprets **nbuf** as follows: • **nbuf**(1) = nbytes -If possible, ***nbytes*** of system buffers will be allocated. • **nbuf**(2) = nmsgs -Up to ***nmsgs*** blocking sends can be buffered at the same time. For C, mpc_task_set intreprets **nbuf** as follows: • **nbuf**[0] = nbytes -If possible, ***nbytes*** of system buffers will be allocated. • **nbuf**[1] = nmsgs -Up to ***nmsgs*** blocking sends can be buffered at the same time. These parameters can only be set once per task. All tasks do not have to call this or supply the same values.

RETURN VALUES

In C, return code = 0 means the operation was successful. Return code = -1 means that an error occurred and the global variable **mperrno** contains the error number. This number, “nnn,” can be used to find a description of the error condition by looking for **0032-nnn** in *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

ERRORS

stype is 1 and the mode is not 0 or 1

stype is 2 and the mode is not 0 or 1

stype is 3 and buffering is already turned on

stype is 3 and the buffer size requested is less than zero

stype is 3 and the number of sends to buffer is less than zero

stype is not 1, 2, or 3

Undetectable error for **stype**= 2 if all the tasks do not call MP_TASK_SET (and with the same value for the mode) if any call it.

EXAMPLES

The examples cause truncation mode to be set.

C Example

```
   #include <mpproto.h>

int task_set_example()
{
   int nbuf,stype,rc;

   stype = 1;
   nbuf  = 1;

   rc = mpc_task_set(&nbuf,stype);
   if(rc)  return(-1);  /* error */

   /* Truncation mode is now in effect */

   mpc_task_query(&nbuf,1,1);

   return(0);
}
```

FORTRAN Example

```
      SUBROUTINE TASK_SET_EXAMPLE

      INTEGER NBUF,STYPE

      STYPE = 1
      NBUF  = 1

      CALL MP_TASK_SET(NBUF,STYPE)

c     Truncation mode is now in effect

      CALL MP_TASK_QUERY(NBUF,1,1)

      END
```

RELATED INFORMATION

Subroutines: MP_TASK_QUERY(3), mpc_task_query(3)

NAME

MP_UNPACK, mpc_unpack – Unpacks data from a contiguous buffer into a non-contiguous buffer.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mpproto.h>
int mpc_unpack(const void *begbuf, void *endbuf, size_t blklen,
               size_t offset, int blknum);
```

FORTRAN SYNOPSIS

```
MP_UNPACK(INTEGER BEGBUF(*), INTEGER ENDBUF(*), INTEGER BLKLEN,
          INTEGER OFFSET, INTEGER BLKNUM)
```

PARAMETERS

begbuf is the buffer that contains the data to be unpacked.

endbuf is the starting location of the buffer into which the unpacked data is to be placed. **endbuf** consists of **blknum** blocks, with each block consisting of **blklen** contiguous bytes. The starting addresses of consecutive blocks are separated by **offset** bytes.

blklen is an integer specifying the size, in bytes, of each block contained in buffer **endbuf**.

offset is an integer specifying the separation, in bytes, between the starting addresses of consecutive blocks.

blknum is an integer specifying the number of blocks contained in the buffer **endbuf**.

DESCRIPTION

Unpacks data from a contiguous buffer into a non-contiguous buffer. For programming convenience, all arguments are presented in the same order as **mpc_pack**. For MP_UNPACK, arguments **blklen**, **offset**, and **blknum** are associated with the result buffer **endbuf**.

RETURN VALUES

In C, return code = 0 means the operation was successful. Return code = -1 means that an error occurred and the global variable **mperrno** contains the error number. This number, “nnn,” can be used to find a description of the error condition by looking for **0032-nnn** in *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

ERRORS

begbuf is not a valid buffer

endbuf is not a valid buffer

blklen is less than zero

offset is less than **blklen**

blknum is less than zero

EXAMPLES

Data in **begbuf** is unpacked into **endbuf**. Data values listed as "x" are not modified.

C Example

```
   #include <mpproto.h>

int unpack_example()
{
   int blklen,offset,blknum,i,rc;
   int begbuf[9],endbuf[15];

   for(i=0;i<9;i++)
      begbuf[i] = i;
   /* "begbuf" = {0,1,2,3,4,5,6,7,8} */

   blklen = 3 * sizeof(int);
   offset = 5 * sizeof(int);
   blknum = 3;
   rc = mpc_unpack(begbuf,endbuf,blklen,offset,blknum);
   if(rc)  return(-1);  /* error */

   /* "endbuf" = {0,1,2,x,x,3,4,5,x,x,6,7,8,x,x}  */

   return(0);
}
```

FORTRAN Example

```
      SUBROUTINE UNPACK_EXAMPLE

      INTEGER BLKLEN,OFFSET,BLKNUM,I
      INTEGER BEGBUF(9),ENDBUF(15)

      DO 10 I=1,9
   10    BEGBUF(I) = I-1
c     "BEGBUF" = {0,1,2,3,4,5,6,7,8}

      BLKLEN = 3 * 4
      OFFSET = 5 * 4
      BLKNUM = 3

      CALL MP_UNPACK(BEGBUF,ENDBUF,BLKLEN,OFFSET,BLKNUM)

c     "ENDBUF" = {0,1,2,x,x,3,4,5,x,x,6,7,8,x,x}

      END
```

RELATED INFORMATION

Subroutines: MP_PACK(3), mpc_pack(3), MP_BVRECV(3), mpc_BVRECV(3)

NAME

MP_WAIT, mpc_wait – Waits until a non-blocking send or receive has completed.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mpproto.h>
int mpc_wait(int *msgid, size_t *nbytes);
```

FORTRAN SYNOPSIS

```
MP_WAIT(INTEGER MSGID, INTEGER NBYTES)
```

PARAMETERS

msgid	is a message identifier returned by a previous call to a non-blocking send or receive function or system-specific flags **allmsg** or **dontcare**.
nbytes	is the number of bytes sent or received.

DESCRIPTION

Waits (blocks) until a specified non-blocking send or receive operation has completed. A non-blocking operation is one that does not wait until the operation completes but instead, continues processing. This routine can be applied to any active non-blocking send or receive operation.

If **msgid**=**dontcare**, the system waits until the next send or receive operation completes. If **msgid**=**allmsg**, the system waits until all pending send and receive operations complete.

For **msgid**=**dontcare** and **msgid**=**allmsg**, **msgid** returns the identifier of the last message that was sent or received and **nbytes** returns the number of bytes sent or received .

To find out the value of **allmsg**, use MP_TASK_QUERY.

RETURN VALUES

In C, return code = 0 means the operation was successful. Return code = -1 means that an error occurred and the global variable **mperrno** contains the error number. This number, “nnn,” can be used to find a description of the error condition by looking for **0032-nnn** in *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

ERRORS

msgid is less than zero and is not a valid wildcard value

EXAMPLES

A message is sent from task 0 to task 1 using non-blocking operations. MP_WAIT is then called by task 0 before the data in the send buffer is modified, and by task 1 before the data in the receive buffer is used.

C Example

```
   #include <mpproto.h>

int wait_example()
{
   int source,dest,type,msglen,msgid,nbytes,rc;
   int numtask,taskid,inmsg,outmsg;

   mpc_environ(&numtask,&taskid);

   type = 1;
   source = 0;
   dest = 1;
   msglen = sizeof(int);

   if(taskid == source)  {
      outmsg = 700;
      rc = mpc_send(&outmsg,msglen,dest,type,&msgid);
      if(rc)  return(-1);  /* error */

      /* Data in "outmsg" must not be modified until "mpc_wait"
      is executed; meanwhile, other work can continue.  */

      rc = mpc_wait(&msgid,&nbytes);
      if(rc)  return(-1);  /* error */

      /*  "outmsg" is now available for reuse.
          "nbytes" returns the actual number of bytes sent. */
   } else {
      if(taskid == dest) {
         rc = mpc_recv(&inmsg,msglen,&source,&type,&msgid);
         if(rc)  return(-1);  /* error */

         /* Incoming data is not available in "inmsg" until "mpc_wait"
         is executed; meanwhile, other work can continue.  */

         rc = mpc_wait(&msgid,&nbytes);
         if(rc)  return(-1);  /* error */

         /*  "inmsg" is now equal to 700.
             "nbytes returns the number of bytes received */
      }
   }
   return(0);
}
```

FORTRAN Example

```
      SUBROUTINE WAIT_EXAMPLE

      INTEGER SOURCE,DEST,TYPE,MSGLEN,MSGID,NBYTES
      INTEGER NUMTASK,TASKID,INMSG,OUTMSG

      CALL MP_ENVIRON(NUMTASK,TASKID)

      TYPE = 1
      SOURCE = 0
      DEST = 1
      MSGLEN = 4

      IF(TASKID .EQ. SOURCE)  THEN
         OUTMSG = 700
         CALL MP_SEND(OUTMSG,MSGLEN,DEST,TYPE,MSGID)

c        Data in "OUTMSG" must not be modified until "MP_WAIT"
c        is executed; meanwhile, other work can continue.

         CALL MP_WAIT(MSGID,NBYTES)

c        "OUTMSG" is now available for reuse.
c        "NBYTES" returns the actual number of bytes sent.

      ELSE
         IF(TASKID .EQ. DEST)  THEN
            CALL MP_RECV(INMSG,MSGLEN,SOURCE,TYPE,MSGID)

c           Incoming data is not available in "INMSG" until "MP_WAIT"
c           is executed; meanwhile, other work can continue.

            CALL MP_WAIT(MSGID,NBYTES)

c           "INMSG" is now equal to 700.
c           "NBYTES returns number of bytes received.

         ENDIF
      ENDIF

      END
```

RELATED INFORMATION

Subroutines: MP_SEND(3), mpc_send(3), MP_RECV(3), mpc_recv(3), MP_STATUS(3), mpc_status(3)

Appendix A. Subroutine Argument Quick Reference

Use this appendix to quickly find out in what order you need to enter the arguments of a subroutine. For detailed information about a subroutine, refer to Chapter 1, "Subroutines Listed Alphabetically" on page 1. Refer to the following table for information:

FORTRAN/C Subroutine	Argument order
MP_BCAST mpc_bcast	msg, msglen, source, gid
MP_BRECV mpc_brecv	inmsg, msglen, source, type, nbytes
MP_BSEND mpc_bsend	outmsg, msglen, dest, type
MP_BSENDRECV mpc_bsendrecv	outmsg, outmsglen, dest, type, inmsg, inmsglen, source, nbytes
MP_BVRECV mpc_bvrecv	inmsg, blklen, offset, blknum, source, type, nbytes
MP_BVSEND mpc_bvsend	outmsg, blklen, offset, blknum, dest, type
MP_COMBINE mpc_combine	outmsg, inmsg, msglen, func, gid
MP_CONCAT mpc_concat	outmsg, inmsg, blklen, gid
MP_ENVIRON mpc_environ	numtask, taskid
MP_GATHER mpc_gather	outmsg, inmsg, blklen, dest, gid
MP_GETLABEL mpc_getlabel	label, gid
MP_GETMEMBERS mpc_getmembers	glist, gid
MP_GETRANK mpc_getrank	rank, taskid, gid
MP_GETSIZE mpc_getsize	gsize, gid
MP_GETTASKID mpc_gettaskid	rank, taskid, gid
MP_GROUP mpc_group	gsize, glist, label, gid

© Copyright IBM Corp. 1995

FORTRAN/C Subroutine	Argument order
MP_INDEX mpc_index	outmsg, inmsg, blklen, gid
MP_PACK mpc_pack	begbuf, endbuf, blklen, offset, blknum
MP_LOCKRNC mpc_lockrnc	newlock, oldlock
MP_PARTITION mpc_partition	parent_gid, key, label, gid
MP_PREFIX mpc_prefix	outmsg, inmsg, msglen, func, gid
MP_PROBE mpc_probe	source, type, mstat
MP_RCVNCALL mpc_rcvncall	inmsg, msglen, source, type, msgid, func
MP_RECV mpc_recv	inmsg, msglen, source, type, msgid
MP_REDUCE mpc_reduce	outmsg, inmsg, msglen, dest, func, gid
MP_SCATTER mpc_scatter	outmsg, inmsg, blklen, source, gid
MP_SEND mpc_send	outmsg, msglen, dest, type, msgid
MP_SHIFT mpc_shift	outmsg, inmsg, msglen, step, flag, gid
MP_STATUS mpc_status	msgid
MP_STOPALL mpc_stopall	errcode
MP_SYNC mpc_sync	gid
MP_TASK_QUERY mpc_task_query	nbuf, nelem, qtype
MP_TASK_SET mpc_task_set	nbuf, stype
MP_UNPACK mpc_unpack	begbuf, endbuf, blklen, offset, blknum
MP_WAIT mpc_wait	msgid, nbytes

Appendix B. Subroutine Argument Definitions

The following table lists all of the arguments (parameters) that the subroutines listed in Chapter 1 use along with their definitions. The definitions listed here are general definitions. For more detailed explanations of these arguments and their relation to the subroutines, refer to Chapter 1

Argument	Description
begbuf	The starting location of the buffer that contains the initial non-contiguous data. **begbuf** consists of **blknum** blocks with each block consisting of **blklen** contiguous bytes.
blklen	A non-negative integer that specifies the size, in bytes, of each block contained in buffer **inmsg**.
blknum	An integer that specifies the number of blocks to be sent.
dest	An integer that specifies the task to which the message is to be sent.
endbuf	The buffer into which the packed data is to be placed.
errcode	An integer that is returned to standard error.
flag	An integer that specifies the type of shift. A value of 0 indicates a circular shift and all other integer values indicate a non-circular shift.
func	An external function which implements a reduction operation.
gid	A group identifier for the group of participating tasks.
glist	An integer array into which the ordered membership list of the task group is placed. **glist** must be large enough to hold the number of tasks in the group.
gsize	An integer into which the number of tasks in the specified task group is placed.
inmsg	The application buffer into which the message is to be received.
inmsglen	A non-negative integer that specifies the size, in bytes, of the buffer **inmsg**.
key	An integer value used to specify the initial rank of the local task in the new group.
label	An integer into which the label of the specified task group is placed
msg	For **task id = source**, **msg** is the application buffer that contains the message to send. For all other tasks in the same group, **msg** is the application buffer into which the incoming message is received.
msgid	An integer returned as a message identifier that can be used to test the status of the message.
msglen	A non-negative integer that specifies the size, in bytes, of the message to send.
mstat	An integer which returns status of the targeted message. When it is non-negative, it is the number of bytes.
nbuf	The integer array into which the returned information is received.
nbytes	An integer which returns the number of bytes received.
nelem	A non-negative integer that specifies the number of array elements to be filled in **nbuf**.
newlock	An integer of 0 to free the lock or 1 to assert the lock.
numtask	An integer which returns the number of tasks in the parallel partition.
oldlock	An integer value returned to indicate whether the lock is free or locked.
offset	An integer that specifies the separation, in bytes, between the starting addresses of consecutive blocks.
outmsg	The application buffer that contains the message to be sent.

© Copyright IBM Corp. 1995

Argument	Description
outmsglen	A non-negative integer that specifies the size, in bytes, of the message to be sent.
parent_gid	A previously defined group to partition.
qtype	An integer that specifies the type of query.
rank	An integer into which the rank of the task in the task group is placed. The rank of a task is its position in relation to the other tasks in the group.
source	An integer that specifies the task that originates the send.
step	An integer that specifies the number of steps of the up or down shift. Up and down refer to increasing or decreasing rank in the group.
stype	An integer that specifies which parameters are being set.
taskid	An integer which specifies a task id. Each task is assigned a unique integer between 0 and **numtask** - 1, inclusive.
type	An integer that specifies the type of the message being received.

Appendix C. MPL Size Limits

The following is a list of MPL size limits.

MPL tunables and limits

The list below includes system limits on the size of various MPL elements, and the relevant environment variable or tunable parameter.

- Number of tasks: MP_PROCS
- Maximum number of tasks: 128
- Maximum number of distinct tags: All non-negiative integers less than 2**32-1

© Copyright IBM Corp. 1995

Appendix D. Reduction Operations

Reduction Operation	Description
i_vadd	Integer vector addition
s_vadd	Single-precision floating-point vector addition
d_vadd	Double-precision floating-point vector addition
i_vmul	Integer vector multiplication
s_vmul	Single-precision floating-point vector multiplication
d_vmul	Double-precision floating-point vector multiplication
i_vmax	Integer vector maximum
s_vmax	Single-precision floating-point vector maximum
d_vmax	Double-precision floating-point vector maximum
i_vmin	Integer vector minimum
s_vmin	Single-precision floating-point vector minimum
d_vmin	Double-precision floating-point vector minimum
b_vand	Bitwise vector AND
b_vor	Bitwise vector OR
b_vxor	Bitwise vector XOR
l_vand	Logical (Boolean) vector AND
l_vor	Logical (Boolean) vector OR

Examples of user-defined reduction functions for integer vector addition.

C Example

```
/* Integer Vector Addition */
void i_vadd (in1, in2, out, len)
int in1[], in2[], out[]
int *len   /* Note that "len" is a pointer */
{
   int i,n;
   n = *len/sizeof(int); /* compute number of elements */
   for (i=0; i<n; i++)
      out[i] = in1[i] + in2[i] /* Do reduction */
}
```

FORTRAN Example

```
      SUBROUTINE I_VADD(IN1,IN2,OUT,LEN)

      INTEGER IN1(*),IN2(*),OUT(*),LEN,I,N

      N = LEN/4
      DO 10 I=1,N
10      OUT(I) = IN1(I) + IN2(I)
      END
```

User-supplied reduction operations have four arguments:

© Copyright IBM Corp. 1995

- The first argument, **in1**, is an array or scalar variable. The length, in bytes, is specified by the fourth argument, **len**.

  This argument is an input array to be reduced.

- The second argument, **in2**, is an array or scalar variable. The length, in bytes, is specified by the fourth argument, **len**.

  This argument is an input array to be reduced.

- The third argument, **out**, is an array or scalar variable. The length, in bytes, is specified by the fourth argument, **len**.

  This argument is the output array for the reduced result. It is assumed that the output array is disjoint from the two input arrays.

- The fourth argument is **len**, a pointer to an integer containing the number of bytes to be reduced.

Users may code their own reduction operations, with the restriction that the operations must be both commutative and associative. Also note that the value of **len** will be passed as a pointer. No communication calls are allowed in user-defined reduction operations.

Appendix E. Parallel Utility Functions

This chapter contains the man pages for the Parallel Utility Functions. These user-callable functions exploit features of the Parallel Environment. Included are functions for:

- updating the Program Marker Array lights
- controlling distribution of STDIN and STDOUT
- synchronizing parallel tasks without using the MPL.

There is a C and FORTRAN version for each of the functions.

The Parallel Utility Functions are:

MP_FLUSH and mpc_flush flushes output buffers to STDOUT. This is a synchronizing call across all parallel tasks.

MP_MARKER and mpc_marker requests that the numeric and text data passed in the call be forwarded to the Program Marker Array for display.

MP_NLIGHTS and mpc_nlights returns the number of Program Marker Array lights defined for this session.

MP_STDOUT_MODE and mpc_stdout_mode requests that STDOUT be set to single, ordered, or unordered mode. In single mode, only one task output is displayed. In unordered mode, output is displayed in the order received at the home node. In ordered mode, each parallel task writes output data to its own buffer; when a flush request is made all the task buffers are flushed, in order of the task id, to home node's STDOUT.

MP_STDOUTMODE_QUERY and mpc_outmode_query returns the mode to which STDOUT is currently set.

For more information on the Program Marker Array, or on controlling STDIN and STDOUT using POE, refer to *IBM Parallel Environment for AIX: Operation and Use*.

© Copyright IBM Corp. 1995

NAME

MP_FLUSH, mpc_flush—Flushes task output buffers to STDOUT.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <pm_util.h>
int mpc_flush(int option);
```

FORTRAN SYNOPSIS

```
MP_FLUSH(INTEGER OPTION)
```

PARAMETERS

option is an AIX file descriptor. The only valid value is:

1 to flush STDOUT buffers.

DESCRIPTION

This Parallel Utility Function flushes output buffers from all the parallel tasks to STDOUT at the home node. This is a synchronizing call across all parallel tasks.

If the current STDOUT mode is ordered, then when all tasks have issued this call or when any of the output buffers are full:

1. all STDOUT buffers are flushed and put out to the user screen (or redirected) in task order.
2. an acknowledgement is sent to all tasks and control is returned to the user.

If current STDOUT mode is unordered and all tasks have issued this call, all output buffers are flushed and put out to the user screen (or redirected).

If the current STDOUT mode is single and all tasks have issued this call, the output buffer for the current single task is flushed and put out to the user screen (or redirected).

NOTES

- This is a synchronizing call regardless of the current STDOUT mode.
- All STDOUT buffers will be flushed at the end of the parallel job.
- If mpc_flush is not used, standard output streams not terminated with a newline character are buffered, even if a subsequent read to standard input is made. This may cause prompt message to appear only after input has been read.

RETURN VALUES

In C and C++ calls, the following applies:

0 indicates successful completion

-1 indicates that an error occurred. A message describing the error will be issued.

EXAMPLES

C Example

The following program uses **poe** with
the **-labelio yes** option and three tasks:

```
    #include <pm_util.h>

main()
{
 mpc_stdout_mode(STDIO_ORDERED);
 printf("These lines will appear in task order\n");
 /*
  * Call mpc_flush here to make sure that one task
  * doesn't change the mode before all tasks have
  * sent the previous printf string to the home node.
  */
 mpc_flush(1);
 mpc_stdout_mode(STDIO_UNORDERED);
 printf("These lines will appear in the order received by the home node\n");
 /*
  * Since synchronization is not used here, one task could actually
  * execute the next statement before one of the other tasks has
  * executed the previous statement, causing one of the unordered
  * lines not to print.
  */
 mpc_stdout_mode(1);
 printf("Only 1 copy of this line will appear from task 1\n");
}
```

Running the above C program produces the
following output (task order of lines 4-6 may differ):

```
   0 : These lines will appear in task order.
   1 : These lines will appear in task order.
   2 : These lines will appear in task order.
   1 : These lines will appear in the order
   received by the home node.
   2 : These lines will appear in the order
   received by the home node.
   0 : These lines will appear in the order
   received by the home node.
   1 : Only 1 copy of this line will appear from task 1.
```

FORTRAN Example

```
 CALL MP_STDOUT_MODE(-2)
 WRITE(6, *) 'These lines will appear in task order'
 CALL MP_FLUSH(1)
 CALL MP_STDOUT_MODE(-3)
 WRITE(6, *) 'These lines will appear in the order received by the
xhome node'
 CALL MP_STDOUT_MODE(1)
 WRITE(6, *) 'Only 1 copy of this line will appear from task 1'
 END
```

RELATED INFORMATION

Functions: mpc_stdout_mode(3), MP_STDOUT_MODE(3), mpc_stdoutmode_query(3), MP_STDOUTMODE_QUERY(3)

NAME

MP_MARKER, mpc_marker—Passes numeric and text data to the Program Marker Array for display.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mp_marker.h>
void mpc_marker(int light, int color, char *string);
```

FORTRAN SYNOPSIS

```
MP_MARKER(INTEGER LIGHT, INTEGER COLOR, CHARACTER STRING)
```

PARAMETERS

light is the light number to be colored. The lights in each task row are numbered, left to right, from 0 to one less than the number of lights. The row on which the light is colored is that of the calling task.

If the value of light is out of range, the parameter is ignored. No light is colored by the subroutine. Setting the light to a negative number allows you to update the string only.

color is the color you want to make the light. Supported values range from 0 to 102. The range 0 to 99 is, roughly, a spectrum starting with black and going through brown, green, blue, purple, red, orange, yellow, and ending with white. 100, 101, and 102 are three shades of gray growing increasingly dark. These are approximations, as the actual colors used are requested from the default X-Windows color map. If the X-Server is not capable of providing the colors in the RGB intensities requested, it colors the light with a close approximation.

If the value of color is out of range, the parameter is ignored. The subroutine does not give the light a new color.

string is the output string to be passed to the PM Array. In C programs the string must be null-terminated. This is not necessary in FORTRAN programs. The string can be any length, although only the first 80 characters can display. A null string can be passed.

DESCRIPTION

This Parallel Utility Function requests that the numeric and text data passed in the call be forwarded to the Program Marker Array for display. This call waits for a specific acknowledgement from the Partition Manager if the number of lights (specified by the **MP_PMLIGHTS** environment variable) is positive. The program returns only after the message has been acknowledged by the POE home node. Hence, this call will slow down the user's application and approximately synchronize it with the Program Marker Array.

If **MP_PMLIGHTS** is set to *0*, no message is sent.

NOTES

- Creates a Visualization Tool(VT) trace record (marker event).
- Sends message only if PMLIGHTS is greater than 0. Each call waits for an acknowledgement from the home node.
- The Program Marker Array X-Windows display routine is distributed as a sample program.

EXAMPLES

C Example

```
The C statement:
   #include <mp_marker.h>
mpc_marker(2, 0, "Starting task 2");

gives the third light of the calling task the color black and
after the lights on that task's row, prints the string "Starting
task 2".
```

FORTRAN Example

```
The FORTRAN statements:

 MP_MARKER(2, 0, 'Starting task 2')

 gives the third light of the calling task the color black and
 after the lights on that task's row, prints the string 'Starting
 task 2'.
```

RELATED INFORMATION

Functions: mpc_nlights(3), MP_NLIGHTS(3)

Commands: pmarray(1)

NAME

MP_NLIGHTS, mpc_nlights—Gets the number of Program Marker Array lights defined for this session.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <mp_marker.h>
int mpc_nlights();
```

FORTRAN SYNOPSIS

```
MP_NLIGHTS(INTEGER NLIGHT)
```

PARAMETERS

In FORTRAN, the number of Program Marker Array lights defined for this session is returned in the variable **NLIGHT**.

DESCRIPTION

This Parallel Utility Function returns the number of Program Marker Array lights defined for this session.

NOTES

The number of Program Marker Array lights may be set using the environment variable **MP_PMLIGHTS** or the command line option **-pmlights**.

RETURN VALUES

In C, the current number of the Program Marker Array lights is the return value.

EXAMPLES

C Example

The following program uses poe with the **-pmlights 3** option and 1 task:

```
   #include <mp_marker.h>

main()
{
 printf("Number of Program Marker Array lights for this session is %d\n",
 mpc_nlights());
}
```

Running the above C program produces the following output:

```
Number of Program Marker Array lights for this session is 3.
```

FORTRAN Example

The following program using poe with the **-pmlights 3** option and 1 task:

```
      INTEGER LIGHTS

      CALL MP_NLIGHTS(LIGHTS)
      WRITE(6, *) 'Number of Program Marker Array lights for this
     xsession is ', LIGHTS
      END
```

Running the above produces the following output:

```
Number of Program Marker Array lights for this session is 3.
```

RELATED INFORMATION

Functions: mpc_marker(3), MP_MARKER(3)

Commands: pmarray(1)

NAME

MP_STDOUT_MODE, mpc_stdout_mode—Requests that STDOUT be set to single, ordered, or unordered mode.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <pm_util.h>
int mpc_stdout_mode(int mode);
```

FORTRAN SYNOPSIS

```
MP_STDOUT_MODE(INTEGER MODE)
```

PARAMETERS

mode is the mode to which STDOUT is to be set. The valid values are:

taskid specifies single mode for STDOUT, where taskid is the task identifier of the new single task. This value must be between 0 and n-1, where n is the total of tasks in the current partition. The taskid requested does not have to be the issuing task.

-2 specifies ordered mode for STDOUT. The macro STDIO_ORDERED is supplied for use in C programs.

-3 specifies unordered mode for STDOUT. The macro STDIO_UNORDERED is supplied for use in C programs.

DESCRIPTION

This Parallel Utility Function requests that STDOUT be set to single, ordered, or unordered mode. In single mode, only one task output is displayed. In unordered mode, output is displayed in the order received at the home node. In ordered mode, each parallel task writes output data to its own buffer. When a flush request is made all the task buffers are flushed, in order of task id, to STDOUT home node.

NOTES

- All current STDOUT buffers will be flushed before the new STDOUT mode is established.
- The initial mode for STDOUT is set by using the environment variable **MP_STDOUTMODE**, or by using the poe command line option **-stdoutmode**, with the latter overriding the former. The default STDOUT mode is unordered.
- This function is implemented with a half second sleep interval to ensure that the mode change request is processed before subsequent writes to STDOUT.

RETURN VALUES

In C and C++ calls, the following applies:

0 indicates successful completion.

-1 indicates that an error occurred. A message describing the error will be issued.

EXAMPLES

C Example

The following program uses poe with the **-labelio yes** option and three tasks:

```
    #include <pm_util.h>

main()
{
 mpc_stdout_mode(STDIO_ORDERED);
 printf("These lines will appear in task order\n");
 /*
  * Call mpc_flush here to make sure that one task
  * doesn't change the mode before all tasks have
  * sent the previous printf string to the home node.
  */
 mpc_flush(1);
 mpc_stdout_mode(STDIO_UNORDERED);
 printf("These lines will appear in the order received by the home node\n");
 /*
  * Since synchronization is not used here, one task could actually
  * execute the next statement before one of the other tasks has
  * executed the previous statement, causing one of the unordered
  * lines not to print.
  */
 mpc_stdout_mode(1);
 printf("Only 1 copy of this line will appear from task 1\n");
}
```

Running the above C program produces the following output (task order of lines 4-6 may differ):

```
0 : These lines will appear in task order.
1 : These lines will appear in task order.
2 : These lines will appear in task order.
1 : These lines will appear in the order received by the home node.
2 : These lines will appear in the order received by the home node.
0 : These lines will appear in the order received by the home node.
1 : Only 1 copy of this line will appear from task 1.
```

FORTRAN Example

```
 CALL MP_STDOUT_MODE(-2)
 WRITE(6, *) 'These lines will appear in task order'
 CALL MP_FLUSH(1)
 CALL MP_STDOUT_MODE(-3)
 WRITE(6, *) 'These lines will appear in the order received by the
xhome node'
 CALL MP_STDOUT_MODE(1)
 WRITE(6, *) 'Only 1 copy of this line will appear from task 1'
 END
```

Running the above program produces the following output (task order of lines 4-6 may differ):

0 : These lines will appear in task order.
1 : These lines will appear in task order.
2 : These lines will appear in task order.
1 : These lines will appear in the order received by the home node.
2 : These lines will appear in the order received by the home node.
0 : These lines will appear in the order received by the home node.
1 : Only 1 copy of this line will appear from task 1.

RELATED INFORMATION

Functions: mpc_flush(3), MP_FLUSH(3), mpc_stdoutmode_query(3), MP_STDOUTMODE_QUERY(3), mpc_synch(3), MP_SYNCH(3), mpcc(1), mpCC(1), mpxlf(1),

NAME

MP_STDOUTMODE_QUERY, mpc_stdoutmode_query—Query current STDOUT mode setting.

LIBRARY

libmpi.a

C SYNOPSIS

```
#include <pm_util.h>
int mpc_stdoutmode_query(int *mode);
```

FORTRAN SYNOPSIS

```
MP_STDOUTMODE_QUERY(INTEGER MODE)
```

PARAMETERS

mode is the address of an integer in which the current STDOUT mode setting will be returned. Possible return values are:

- **taskid** indicates that the current STDOUT mode is single, i.e. output for only task taskid is displayed.
- **-2** indicates that the current STDOUT mode is ordered. The macro STDIO_ORDERED is supplied for use in C programs.
- **-3** indicates that the current STDOUT mode is unordered. The macro STDIO_UNORDERED is supplied for use in C programs.

DESCRIPTION

This Parallel Utility Function returns the mode to which STDOUT is currently set.

NOTES

- Between the time one task issues a mode query request and receives a response, it is possible that another task can change the STDOUT mode setting to another value unless proper synchronization is used.

RETURN VALUES

In C and C++ calls, the following applies:

- **0** indicates successful completion
- **-1** indicates that an error occurred. A message describing the error will be issued.

EXAMPLES

C Example

The following program uses poe with one task:

```
#include <pm_util.h>

main()
{
 int mode;

 mpc_stdoutmode_query(&mode);
 printf("Initial (default) STDOUT mode is %d\n", mode);
 mpc_stdout_mode(STDIO_ORDERED);
 mpc_stdoutmode_query(&mode);
 printf("New STDOUT mode is %d\n", mode);
}
```

Running the above program produces the following output:

```
Initial (default) STDOUT mode is -3
New STDOUT mode is -2
```

FORTRAN Example

The following program uses poe with one task:

```
      INTEGER MODE

      CALL MP_STDOUTMODE_QUERY(mode)
      WRITE(6, *) 'Initial (default) STDOUT mode is', mode
      CALL MP_STDOUT_MODE(-2)
      CALL MP_STDOUTMODE_QUERY(mode)
      WRITE(6, *) 'New STDOUT mode is', mode
      END
```

Running the above program produces the following output:

```
Initial (default) STDOUT mode is -3
New STDOUT mode is -2
```

RELATED INFORMATION

Functions: mpc_flush(3), MP_FLUSH(3), mp_stdout-mode(3), MP_STDOUT_MODE(3), mpc_synch(3), MP_SYNCH(3), mpcc(1), mpCC(1), mpxlf(1),

MP_STDOUTMODE_QUERY

Appendix F. Tracing Routines

This chapter contains the syntax man pages for the tracing routines used in the Visualization Tool Tracing Subsystem. These user-callable routines allow the programmer to customize the trace collection from within the application being traced. Included are functions for:

- modifying trace collection parameters
- starting and stopping the trace from within the program, dynamically and selectively
- writing trace records back to the home node.

There is a C and FORTRAN version for each of the routines.

The following tracing routines will be provided:

VT_TRC_FLUSH and VT_trc_flush_c flushes the memory buffer to the trace collection directory.

VT_TRC_SET_PARAMS and VT_trc_set_params_c sets certain tracing parameters at which the dig should sample kernel statistics.

VT_TRC_START and VT_trc_start_c requests a trace from within the program, dynamically and selectively.

VT_TRC_STOP and VT_trc_stop_c requests that the collecting of trace events be discontinued.

For more information, refer to *IBM Parallel Environment for AIX: Operation and Use*.

© Copyright IBM Corp. 1995

NAME

VT_TRC_FLUSH, VT_trc_flush_c—Flushes the memory buffer to the trace collection directory.

LIBRARY

- C Library (libvtd.a)
- FORTRAN Library (libvtd.a)

  The above are automatically included by the POE functions.

C SYNOPSIS

```
#include <VT_trc.h>
int VT_trc_flush_c();
```

FORTRAN SYNOPSIS

```
VT_TRC_FLUSH(INTEGER RETURN_CODE)
```

PARAMETERS

In FORTRAN, the following applies:

return_code integer which receives the return value.

- **0** indicates successful completion
- **-1** indicates an error occurred.

DESCRIPTION

This routine may be called from the application to flush the memory buffer to the trace collection directory. This is specified by the MP_TRACE directory environment variable or -tracedir parameter and defaults to the directory from which the application was stored.

RETURN VALUES

In C and C++ calls, the following applies:

- **0** indicates successful completion
- **-1** indicates that an error occurred. A message describing the error will be issued.

ERRORS

- 0033-3051 VT_trc_capture(), write_tmp_to_usd_file(), No space is left in usd_file

  The node trace file reached the maximum size set by MP_PARMSIZE or -parmsize.
- 0033-3072 write_temp_to_usd_file(), Fork failed to the write data onto the user

  The trace writing routine could not fork. It is probable that either too many processes are active or there is not enough page space.

- 0033-3088 VT_trrtn:::write_trc_data(), Tracing continued after reducing the max size for temp file from number to number

  The temp directory was full so the size of the maximum temp trace file was reduced.

- 0033-3092 VT_trint, statfs call failed, Disk space computation failed.

  An error occurred while obtaining information about the temp file directory.

For more information about error conditions, refer to *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

EXAMPLES

C Example

```
    #include <stdlib.h>
    #include <stdio.h>
    #include <VT_trc.h>

void main()
}
  int     i;
#define SIZE 1024
  int    msgin[SIZE], msgout[SIZE];
  int    numtask, taskid;
  int    src, dest, len, send_msgid, recv_msgid, nbytes;
  int    type;

  /* Find out number of tasks/nodes. */
  mpc_environ (&numtask,&taskid );

  /************************************
    Set tracing parameters as follows:
    Memory buffer = 500K
    Temporary File = 1M
    Final Node File = 10M
    Sample every second
    Do not use wrap-around buffer
    ************************************/
  if(0!=VT_trc_set_params_c(500000,10000000,1000,0))
    }
      printf("Could not reset trace parameters\n");
    {

  /* Disable tracing while the message buffer is initialized */
  if(0!=VT_trc_stop_c())
    }
      printf("Could not stop tracing\n");
    {
  /* Flush the buffer so far to the control node */
  if(0!=VT_trc_flush_c())
    }
      printf("Could not flush trace buffer\n");
    {

  for(i=0;i<SIZE;i++)
    msgout[i] = taskid;
```

```
  /* Re-enable tracing (9 asks for everything but only events */
  /* enabled by command line or environment variable will be  */
  /* re-enabled)                                              */
  if(0!=VT_trc_start_c(9))
    }
      printf("Could not restart tracing\n");
    {


  len  = SIZE*sizeof(int);
  dest = taskid + 1;
  if( dest == numtask ) dest = 0;
  type = 1;
  mpc_send ( msgout, len, dest, type, &send_msgid );

  len  = SIZE*sizeof(int);
  src = taskid -1;
  if( src < 0 ) src = numtask-1;
  type = 1;
  mpc_recv ( msgin, len, &src, &type, &recv_msgid );

  mpc_wait (&send_msgid,&nbytes );
  mpc_wait (&recv_msgid,&nbytes );

{
```

FORTRAN Example

```
      PROGRAM  TRCDEMO
C
      IMPLICIT   NONE
      INTEGER    SIZE, I
      INTEGER    BSZ, PSZ
      INTEGER    SAMP, WRAP
      PARAMETER  ( SIZE = 1024 )
      PARAMETER  ( BSZ = 500000, PSZ = 10000000 )
      PARAMETER  ( SAMP = 1000, WRAP = 0 )
      INTEGER    MSGIN(SIZE), MSGOUT(SIZE)
      INTEGER    NUMTASK, TASKID
      INTEGER    SRC, DEST, LEN, SEND_MSGID, RECV_MSGID, NBYTES
      INTEGER    TYPE
      INTEGER    RC
C
C  FIND OUT NUMBER OF TASKS/NODES.
         CALL MP_ENVIRON ( NUMTASK, TASKID )
C
C    SET TRACING PARAMETERS AS FOLLOWS:
C    MEMORY BUFFER = 500K
C    TEMPORARY FILE = 1M
C    FINAL NODE FILE = 10M
C    SAMPLE EVERY SECOND
C    DO NOT USE WRAP-AROUND BUFFER
C
         CALL VT_TRC_SET_PARAMS ( BSZ, PSZ, SAMP, WRAP, RC )
         IF( RC .NE. 0 ) THEN
            WRITE(6,*)'Could not reset trace parameters'
```

```
         ENDIF
C
C DISABLE TRACING WHILE THE MESSAGE BUFFER IS INITIALIZED
         CALL VT_TRC_STOP ( RC )
         IF(RC .NE. 0) THEN
            WRITE(6,*)'Could not stop tracing'
         ENDIF
C
C FLUSH THE BUFFER SO FAR TO THE CONTROL NODE
         CALL VT_TRC_FLUSH ( RC )
         IF(RC .NE. 0) THEN
            WRITE(6,*)'Could not flush trace buffer stop tracing'
         ENDIF
C
         DO I = 1, SIZE
            MSGOUT(I) = TASKID
         ENDDO
C
C RE-ENABLE TRACING (9 ASKS FOR EVERYTHING BUT ONLY EVENTS
C ENABLED BY COMMAND LINE OR ENVIRONMENT VARIABLE WILL BE
C RE-ENABLED)
         CALL VT_TRC_START ( 9, RC )
         IF(RC .NE. 0) THEN
            WRITE(6,*)'Could not restart tracing'
         ENDIF
C
         LEN  = 4 * SIZE
         DEST = TASKID + 1
         IF( DEST .EQ. NUMTASK ) THEN
            DEST = 0
         ENDIF
         TYPE = 1
         CALL MP_SEND ( MSGOUT, LEN, DEST, TYPE, SEND_MSGID )
C
         LEN  = 4 * SIZE
         SRC  = TASKID - 1
         IF( SRC .LT. 0 ) THEN
            SRC = NUMTASK - 1
         ENDIF
         TYPE = 1
         CALL MP_RECV ( MSGIN, LEN, SRC, TYPE, RECV_MSGID )
         CALL MP_WAIT ( RECV_MSGID, NBYTES )
C
         CALL MP_WAIT ( SEND_MSGID, NBYTES )

         STOP
         END
C
```

RELATED INFORMATION

Functions: VT_trc_set_params_c, VT_TRC_SET_PARAMS, VT_trc_start_c, VT_TRC_START, VT_trc_stop_c, VT_TRC_STOP

For information about VT tracing philosophy, refer to *IBM Parallel Environment for AIX: Operation and Use*.

NAME

VT_TRC_SET_PARAMS, VT_trc_set_params_c—Allows applications to modify certain tracing parameters during execution.

LIBRARY

- C Library (libvtd.a) functions.
- FORTRAN Library (libvtd.a)

  The above are automatically included by the POE.

C SYNOPSIS

```
#include <VT_trc.h>
int VT_trc_set_params_c(size_t buff_size, size_t file_size,
                        int sampling_freq, int wrap_around_flag);
```

FORTRAN SYNOPSIS

```
VT_TRC_SET_PARAMS(INTEGER BUFF_SIZE, INTEGER FILE_SIZE,
                  INTEGER SAMPLING_FREQ, INTEGER WRAP_AROUND_FLAG,
                  INTEGER RETURN_CODE)
```

PARAMETERS

buff_size the size of the memory buffer

file_size the size of the node trace files

sampling_freq the interval with which to sample AIX units (microseconds)

wrap_around_flag if non-zero, a wrap around memory buffer will be used.

return_code integer which receives the return value.

- **0** indicates successful completion
- **-1** indicates an error occurred.

DESCRIPTION

This routine allows applications to set certain tracing parameters such as the size of the memory buffer and trace files, and the kernel statistics sampling interval.

NOTES

- Parameter values replace command line values.

RETURN VALUES

In C and C++ calls, the following applies:

0 indicates successful completion

-1 indicates that an error occurred. A message describing the error will be issued.

ERRORS

- 0033-3011 VT_trc_set-params(), reallocation of ... bytes failed

  The application could not obtain the memory required to change parameters.

- 0033-3090 write failed to the dig daemon, ..., error is ...

  The application could not contact the kernel statistics sampling daemon.

For more information about error conditions, refer to *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

EXAMPLES

C Example

```
   #include <stdlib.h>
   #include <stdio.h>
   #include <VT_trc.h>

void main()
}
  int     i;
#define SIZE 1024
  int    msgin[SIZE], msgout[SIZE];
  int    numtask, taskid;
  int    src, dest, len, send_msgid, recv_msgid, nbytes;
  int    type;

  /* Find out number of tasks/nodes. */
  mpc_environ (&numtask,&taskid );

  /************************************
    Set tracing parameters as follows:
    Memory buffer = 500K
    Temporary File = 1M
    Final Node File = 10M
    Sample every second
    Do not use wrap-around buffer
    ************************************/
  if(0!=VT_trc_set_params_c(500000,10000000,1000,0))
    }
      printf("Could not reset trace parameters\n");
    {

  /* Disable tracing while the message buffer is initialized */
  if(0!=VT_trc_stop_c())
    }
      printf("Could not stop tracing\n");
    {
  /* Flush the buffer so far to the control node */
  if(0!=VT_trc_flush_c())
    }
      printf("Could not flush trace buffer\n");
    {

  for(i=0;i<SIZE;i++)
    msgout[i] = taskid;
```

```
  /* Re-enable tracing (9 asks for everything but only events */
  /* enabled by command line or environment variable will be  */
  /* re-enabled)                                              */
  if(0!=VT_trc_start_c(9))
    }
      printf("Could not restart tracing\n");
    {


  len  = SIZE*sizeof(int);
  dest = taskid + 1;
  if( dest == numtask ) dest = 0;
  type = 1;
  mpc_send ( msgout, len, dest, type, &send_msgid );

  len  = SIZE*sizeof(int);
  src = taskid -1;
  if( src < 0 ) src = numtask-1;
  type = 1;
  mpc_recv ( msgin, len, &src, &type, &recv_msgid );

  mpc_wait (&send_msgid,&nbytes );
  mpc_wait (&recv_msgid,&nbytes );

{
```

FORTRAN Example

```
      PROGRAM  TRCDEMO
C
      IMPLICIT   NONE
      INTEGER    SIZE, I
      INTEGER    BSZ, PSZ
      INTEGER    SAMP, WRAP
      PARAMETER  ( SIZE = 1024 )
      PARAMETER  ( BSZ = 500000, PSZ = 10000000 )
      PARAMETER  ( SAMP = 1000, WRAP = 0 )
      INTEGER    MSGIN(SIZE), MSGOUT(SIZE)
      INTEGER    NUMTASK, TASKID
      INTEGER    SRC, DEST, LEN, SEND_MSGID, RECV_MSGID, NBYTES
      INTEGER    TYPE
      INTEGER    RC
C
C  FIND OUT NUMBER OF TASKS/NODES.
         CALL MP_ENVIRON ( NUMTASK, TASKID )
C
C    SET TRACING PARAMETERS AS FOLLOWS:
C    MEMORY BUFFER = 500K
C    TEMPORARY FILE = 1M
C    FINAL NODE FILE = 10M
C    SAMPLE EVERY SECOND
C    DO NOT USE WRAP-AROUND BUFFER
C
         CALL VT_TRC_SET_PARAMS ( BSZ, PSZ, SAMP, WRAP, RC )
         IF( RC .NE. 0 ) THEN
            WRITE(6,*)'Could not reset trace parameters'
```

```
         ENDIF
C
C DISABLE TRACING WHILE THE MESSAGE BUFFER IS INITIALIZED
         CALL VT_TRC_STOP ( RC )
         IF(RC .NE. 0) THEN
            WRITE(6,*)'Could not stop tracing'
         ENDIF
C
C FLUSH THE BUFFER SO FAR TO THE CONTROL NODE
         CALL VT_TRC_FLUSH ( RC )
         IF(RC .NE. 0) THEN
            WRITE(6,*)'Could not flush trace buffer stop tracing'
         ENDIF
C
         DO I = 1, SIZE
            MSGOUT(I) = TASKID
         ENDDO
C
C RE-ENABLE TRACING (9 ASKS FOR EVERYTHING BUT ONLY EVENTS
C ENABLED BY COMMAND LINE OR ENVIRONMENT VARIABLE WILL BE
C RE-ENABLED)
         CALL VT_TRC_START ( 9, RC )
         IF(RC .NE. 0) THEN
            WRITE(6,*)'Could not restart tracing'
         ENDIF
C
         LEN  = 4 * SIZE
         DEST = TASKID + 1
         IF( DEST .EQ. NUMTASK ) THEN
            DEST = 0
         ENDIF
         TYPE = 1
         CALL MP_SEND ( MSGOUT, LEN, DEST, TYPE, SEND_MSGID )
C
         LEN  = 4 * SIZE
         SRC  = TASKID - 1
         IF( SRC .LT. 0 ) THEN
            SRC = NUMTASK - 1
         ENDIF
         TYPE = 1
         CALL MP_RECV ( MSGIN, LEN, SRC, TYPE, RECV_MSGID )
         CALL MP_WAIT ( RECV_MSGID, NBYTES )
C
         CALL MP_WAIT ( SEND_MSGID, NBYTES )

         STOP
         END
C
```

RELATED INFORMATION

Functions: VT_trc_flush_c, VT_TRC_FLUSH, VT_trc_start_c, VT_TRC_START, VT_trc_stop_c, VT_TRC_STOP

For information about VT tracing philosophy, refer to *IBM Parallel Environment for AIX: Operation and Use*.

NAME

VT_TRC_START, VT_trc_start_c—Allows you to start the program trace from within the program, dynamically and selectively.

LIBRARY

- C Library (libvtd.a)
- FORTRAN Library (libvtd.a)

The above is automatically included by the POE functions.

C SYNOPSIS

```
#include <VT_trc.h>
int VT_trc_start_c(int flag);
```

FORTRAN SYNOPSIS

```
VT_TRC_START(INTEGER FLAG, INTEGER RETURN_CODE)
```

PARAMETERS

flag the trace control flags that define what information is traced.

return_code integer which receives the return value.

- **0** indicates successful completion
- **-1** indicates an error occurred.

DESCRIPTION

This routine may be called by the application program. This gives the program capability to start the trace from within the program, dynamically and selectively. The flag has the same meaning as the trace flag passed by PM upon startup. If the flag indicates VT_AIX_TRACE, this routine also sends a message to the monitoring daemon process.

NOTES

- Tracing is initially on when the application starts.
- If a trace event, such as kernel statistics or communication events, was not selected at the start of the application it cannot be enabled by this routine.

RETURN VALUES

In C and C++ calls, the following applies:

0 indicates successful completion

-1 indicates that an error occurred. A message describing the error will be issued.

ERRORS

- 0033-3090 write failed to the dig daemon ...., error is ...

  The application could not contact the kernel statistics sampling daemon.

For more information about error conditions, refer to *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

EXAMPLES

C Example

```
   #include <stdlib.h>
   #include <stdio.h>
   #include <VT_trc.h>

void main()
}
  int     i;
#define SIZE 1024
  int    msgin[SIZE], msgout[SIZE];
  int    numtask, taskid;
  int    src, dest, len, send_msgid, recv_msgid, nbytes;
  int    type;

  /* Find out number of tasks/nodes. */
  mpc_environ (&numtask,&taskid );

  /************************************
    Set tracing parameters as follows:
    Memory buffer = 500K
    Temporary File = 1M
    Final Node File = 10M
    Sample every second
    Do not use wrap-around buffer
    ************************************/
  if(0!=VT_trc_set_params_c(500000,10000000,1000,0))
    }
      printf("Could not reset trace parameters\n");
    {

  /* Disable tracing while the message buffer is initialized */
  if(0!=VT_trc_stop_c())
    }
      printf("Could not stop tracing\n");
    {
  /* Flush the buffer so far to the control node */
  if(0!=VT_trc_flush_c())
    }
      printf("Could not flush trace buffer\n");
    {

  for(i=0;i<SIZE;i++)
    msgout[i] = taskid;

  /* Re-enable tracing (9 asks for everything but only events */
  /* enabled by command line or environment variable will be  */
  /* re-enabled)                                              */
```

```
  if(0!=VT_trc_start_c(9))
    }
      printf("Could not restart tracing\n");
    {


  len  = SIZE*sizeof(int);
  dest = taskid + 1;
  if( dest == numtask ) dest = 0;
  type = 1;
  mpc_send ( msgout, len, dest, type, &send_msgid );

  len  = SIZE*sizeof(int);
  src = taskid -1;
  if( src < 0 ) src = numtask-1;
  type = 1;
  mpc_recv ( msgin, len, &src, &type, &recv_msgid );

  mpc_wait (&send_msgid,&nbytes );
  mpc_wait (&recv_msgid,&nbytes );

{
```

FORTRAN Example

```
      PROGRAM  TRCDEMO
C
      IMPLICIT   NONE
      INTEGER    SIZE, I
      INTEGER    BSZ, PSZ
      INTEGER    SAMP, WRAP
      PARAMETER  ( SIZE = 1024 )
      PARAMETER  ( BSZ = 500000, PSZ = 10000000 )
      PARAMETER  ( SAMP = 1000, WRAP = 0 )
      INTEGER    MSGIN(SIZE), MSGOUT(SIZE)
      INTEGER    NUMTASK, TASKID
      INTEGER    SRC, DEST, LEN, SEND_MSGID, RECV_MSGID, NBYTES
      INTEGER    TYPE
      INTEGER    RC
C
C  FIND OUT NUMBER OF TASKS/NODES.
         CALL MP_ENVIRON ( NUMTASK, TASKID )
C
C    SET TRACING PARAMETERS AS FOLLOWS:
C    MEMORY BUFFER = 500K
C    TEMPORARY FILE = 1M
C    FINAL NODE FILE = 10M
C    SAMPLE EVERY SECOND
C    DO NOT USE WRAP-AROUND BUFFER
C
         CALL VT_TRC_SET_PARAMS ( BSZ, PSZ, SAMP, WRAP, RC )
         IF( RC .NE. 0 ) THEN
            WRITE(6,*)'Could not reset trace parameters'
         ENDIF
C
C DISABLE TRACING WHILE THE MESSAGE BUFFER IS INITIALIZED
         CALL VT_TRC_STOP ( RC )
         IF(RC .NE. 0) THEN
```

```
            WRITE(6,*)'Could not stop tracing'
         ENDIF
C
C FLUSH THE BUFFER SO FAR TO THE CONTROL NODE
         CALL VT_TRC_FLUSH ( RC )
         IF(RC .NE. 0) THEN
            WRITE(6,*)'Could not flush trace buffer stop tracing'
         ENDIF
C
         DO I = 1, SIZE
            MSGOUT(I) = TASKID
         ENDDO
C
C RE-ENABLE TRACING (9 ASKS FOR EVERYTHING BUT ONLY EVENTS
C ENABLED BY COMMAND LINE OR ENVIRONMENT VARIABLE WILL BE
C RE-ENABLED)
         CALL VT_TRC_START ( 9, RC )
         IF(RC .NE. 0) THEN
            WRITE(6,*)'Could not restart tracing'
         ENDIF
C
         LEN  = 4 * SIZE
         DEST = TASKID + 1
         IF( DEST .EQ. NUMTASK ) THEN
            DEST = 0
         ENDIF
         TYPE = 1
         CALL MP_SEND ( MSGOUT, LEN, DEST, TYPE, SEND_MSGID )
C
         LEN  = 4 * SIZE
         SRC  = TASKID - 1
         IF( SRC .LT. 0 ) THEN
            SRC = NUMTASK - 1
         ENDIF
         TYPE = 1
         CALL MP_RECV ( MSGIN, LEN, SRC, TYPE, RECV_MSGID )
         CALL MP_WAIT ( RECV_MSGID, NBYTES )
C
         CALL MP_WAIT ( SEND_MSGID, NBYTES )

         STOP
         END
C
```

RELATED INFORMATION

Functions: VT_trc_flush_c, VT_TRC_FLUSH, VT_trc_set_params_c, VT_TRC_SET_PARAMS, VT_trc_start_c, VT_TRC_START, VT_trc_stop_c, VT_TRC_STOP

For information about VT tracing philosophy, refer to *IBM Parallel Environment for AIX: Operation and Use*.

NAME

VT_TRC_STOP, VT_trc_stop_c—Stops collecting trace events.

LIBRARY

- C Library (libvtd.a)
- FORTRAN Library (libvtd.a)

The above is automatically included by the POE functions.

C SYNOPSIS

```
#include <VT_trc.h>
int VT_trc_stop_c();
```

FORTRAN SYNOPSIS

```
VT_TRC_STOP(INTEGER RETURN_CODE)
```

PARAMETERS

return_code integer which receives the return value.

0 indicates successful completion

-1 indicates an error occurred.

DESCRIPTION

This routine may be called by the application program to stop tracing.

RETURN VALUES

In C and C++ calls, the following applies:

0 indicates successful completion

-1 indicates that an error occurred. A message describing the error will be issued.

ERRORS

- 0033-3090 write failed to the dig daemon ...., error is ...

  The application could not contact the kernel statistics sampling daemon.

For more information about error conditions, refer to *IBM Parallel Environment for AIX: Installation, Administration, and Diagnosis*.

EXAMPLES

C Example

```
   #include <stdlib.h>
   #include <stdio.h>
   #include <VT_trc.h>

void main()
}
  int     i;
#define SIZE 1024
  int    msgin[SIZE], msgout[SIZE];
  int    numtask, taskid;
  int    src, dest, len, send_msgid, recv_msgid, nbytes;
  int    type;

  /* Find out number of tasks/nodes. */
  mpc_environ (&numtask,&taskid );

  /************************************
    Set tracing parameters as follows:
    Memory buffer = 500K
    Temporary File = 1M
    Final Node File = 10M
    Sample every second
    Do not use wrap-around buffer
    ************************************/
  if(0!=VT_trc_set_params_c(500000,10000000,1000,0))
    }
      printf("Could not reset trace parameters\n");
    {

  /* Disable tracing while the message buffer is initialized */
  if(0!=VT_trc_stop_c())
    }
      printf("Could not stop tracing\n");
    {
  /* Flush the buffer so far to the control node */
  if(0!=VT_trc_flush_c())
    }
      printf("Could not flush trace buffer\n");
    {

  for(i=0;i<SIZE;i++)
    msgout[i] = taskid;

  /* Re-enable tracing (9 asks for everything but only events */
  /* enabled by command line or environment variable will be  */
  /* re-enabled)                                              */
  if(0!=VT_trc_start_c(9))
    }
      printf("Could not restart tracing\n");
    {


  len  = SIZE*sizeof(int);
  dest = taskid + 1;
  if( dest == numtask ) dest = 0;
  type = 1;
  mpc_send ( msgout, len, dest, type, &send_msgid );
```

```
  len  = SIZE*sizeof(int);
  src = taskid -1;
  if( src < 0 ) src = numtask-1;
  type = 1;
  mpc_recv ( msgin, len, &src, &type, &recv_msgid );

  mpc_wait (&send_msgid,&nbytes );
  mpc_wait (&recv_msgid,&nbytes );

{
```

FORTRAN Example

```
      PROGRAM  TRCDEMO
C
      IMPLICIT   NONE
      INTEGER    SIZE, I
      INTEGER    BSZ, PSZ
      INTEGER    SAMP, WRAP
      PARAMETER  ( SIZE = 1024 )
      PARAMETER  ( BSZ = 500000, PSZ = 10000000 )
      PARAMETER  ( SAMP = 1000, WRAP = 0 )
      INTEGER    MSGIN(SIZE), MSGOUT(SIZE)
      INTEGER    NUMTASK, TASKID
      INTEGER    SRC, DEST, LEN, SEND_MSGID, RECV_MSGID, NBYTES
      INTEGER    TYPE
      INTEGER    RC
C
C  FIND OUT NUMBER OF TASKS/NODES.
         CALL MP_ENVIRON ( NUMTASK, TASKID )
C
C    SET TRACING PARAMETERS AS FOLLOWS:
C    MEMORY BUFFER = 500K
C    TEMPORARY FILE = 1M
C    FINAL NODE FILE = 10M
C    SAMPLE EVERY SECOND
C    DO NOT USE WRAP-AROUND BUFFER
C
         CALL VT_TRC_SET_PARAMS ( BSZ, PSZ, SAMP, WRAP, RC )
         IF( RC .NE. 0 ) THEN
            WRITE(6,*)'Could not reset trace parameters'
         ENDIF
C
C DISABLE TRACING WHILE THE MESSAGE BUFFER IS INITIALIZED
         CALL VT_TRC_STOP ( RC )
         IF(RC .NE. 0) THEN
            WRITE(6,*)'Could not stop tracing'
         ENDIF
C
C FLUSH THE BUFFER SO FAR TO THE CONTROL NODE
         CALL VT_TRC_FLUSH ( RC )
         IF(RC .NE. 0) THEN
            WRITE(6,*)'Could not flush trace buffer stop tracing'
         ENDIF
C
         DO I = 1, SIZE
            MSGOUT(I) = TASKID
```

```
         ENDDO
C
C RE-ENABLE TRACING (9 ASKS FOR EVERYTHING BUT ONLY EVENTS
C ENABLED BY COMMAND LINE OR ENVIRONMENT VARIABLE WILL BE
C RE-ENABLED)
         CALL VT_TRC_START ( 9, RC )
         IF(RC .NE. 0) THEN
            WRITE(6,*)'Could not restart tracing'
         ENDIF
C
         LEN  = 4 * SIZE
         DEST = TASKID + 1
         IF( DEST .EQ. NUMTASK ) THEN
            DEST = 0
         ENDIF
         TYPE = 1
         CALL MP_SEND ( MSGOUT, LEN, DEST, TYPE, SEND_MSGID )
C
         LEN  = 4 * SIZE
         SRC  = TASKID - 1
         IF( SRC .LT. 0 ) THEN
            SRC = NUMTASK - 1
         ENDIF
         TYPE = 1
         CALL MP_RECV ( MSGIN, LEN, SRC, TYPE, RECV_MSGID )
         CALL MP_WAIT ( RECV_MSGID, NBYTES )
C
         CALL MP_WAIT ( SEND_MSGID, NBYTES )

         STOP
         END
C
```

RELATED INFORMATION

Functions: VT_trc_flush_c, VT_TRC_FLUSH, VT_trc_set_params_c, VT_TRC_SET_PARAMS, VT_trc_start_c, VT_TRC_START

For information about VT tracing philosophy, refer to *IBM Parallel Environment for AIX: Operation and Use*.

VT_TRC_STOP

Appendix G. Programming Considerations for User Applications in POE

Language Bindings

Both FORTRAN and C bindings for MPL are contained in the same library (libmpi.a) and can be freely intermixed.

The following documents various limitations, restrictions, and programming considerations for user applications written to run under the Parallel Environment.

These considerations apply only to C and C++ programs.

Environment Overview

The end user is encouraged to think of the Parallel Operating Environment(POE) (also referred to as the **poe** command) as an ordinary (serial) command. It accepts redirected I/O, can be run under the **nice** and **time** commands, interprets command flags, and can be invoked in shell scripts.

An n-task parallel job running in the Parallel Operating Environment actually consists of the n user tasks, an equal number (n) of instances of the Parallel Environment **pmd** daemon (which is the parent process of the user's task), and the POE "home node" on which the **poe** command is executed. A **pmd** daemon is started by the POE home node on each machine on which each task runs, and serves as the point of contact between the home node and the user's tasks.

The POE home node routes standard input, standard output and standard error streams between the home node and the user's tasks via the **pmd** daemon, using TCP/IP sockets for this purpose. The sockets are created when the POE home node starts the **pmd** daemon for each task of a parallel job. The POE home node and **pmd** also use the sockets to exchange control messages to provide task synchronization, exit status and signaling. These capabilities do not depend upon the Message Passing library and are available to control any parallel program run by the **poe** command.

Exit Status

Exit status (a value between 0 and 255 inclusive) is returned from POE on the home node reflecting the composite exit status of the user's parallel application, as follows:

- If MPI_ABORT(comm,nn>0,ierror) or MPI_Abort(comm,nn>0) is called, the exit status is nn (mod 256).
- If MP_STOPALL(nn>=0) or mpc_stopall(nn>=0) is called, the exit status is nn (mod 256).
- If all tasks terminate via exit(MM> =0) or STOP MM> =0 and MM is not equal to 1 for any node, then POE provides a synchronization barrier at the exit. The exit status is the largest value of MM from any parallel job (mod 256).
- If any task terminates via exit(MM =1) or STOP MM =1, then POE will immediately terminate the parallel job, as if MP_STOPALL(1) or

© Copyright IBM Corp. 1995

MPI_Abort(MPI_COMM_WORLD,1) had been called. This may also occur if a FORTRAN I/O library error occurs.

- If any task terminates via a signal (for example, a segment violation), the exit status is 128+signal and the entire job is immediately terminated.
- If POE terminates before the start of the user's application, the exit status is =1.
- If the user's application cannot be loaded or fails during initialization of the message passing library (before the user's main() is called), the exit status is =255.
- The user should explicitly call exit(MM) or STOP MM to set the desired exit code. A program exiting without an explicit exit value returns unpredictable status, and may result in causing premature termination of the parallel application.

POE Job Step Function

The POE job-step function is tailored for the execution of a sequence of separate yet inter-related dependent programs. Therefore, it provides you with a job control mechanism that allows both job-step progression and job-step termination. The job control mechanism is the program's exit code.

- Job-step progression:

  POE will continue the job-step sequence if the task exit code is 0 or in the range of 2 - 127.

- Job-step termination:

  POE will terminate the parallel job, and not execute any remaining user programs in the job-step list if the task exit code is 1 or greater than or equal to 127.

- Default termination:

  Any POE infrastructure detected failure (such as failure to open pipes to the child process or an exec failure to start the user's executable) will terminate the parallel job, and not execute any remaining user programs in the job-step queue.

POE Additions To The User Executable

When the user's executable is compiled with mpcc, mpxlf or mpCC, POE adds the following routines:

- POE's own version of *crt0*.

  POE's version of *crt0* is statically bound to the application executable. It calls the message passing initialization, and then the application program main().

- Message passing initialization module.

  POE initializes the parallel message passing library and determines that all nodes can communicate successfully before the user main() program gains control. As a result, any program compiled with the POE compiler scripts must be run under the control of POE and is not suitable as a serial program.

  If communication initialization fails, the parallel task is terminated with the exit code described above.

- Signal handlers.

  POE installs signal handlers for most signals that cause program termination in order to complete the VT trace file, if enabled, and to notify the other tasks of termination. POE then causes the program to exit normally with a code of (128+signal). If the user installs a signal handler for any of these signals, it should call the POE registered signal handler if the process decides to terminate. (See the section following entitled Let POE Handle Signals When Possible.)

  The following are the signals specifically handled by POE or the message passing library:

  – SIGHUP

    Caught and exits with an exit code of 128+SIGHUP.

  – SIGINT

    Caught and exits with an exit code of 128+SIGINT.

    Note: This signal may be caught by user or by dbx, in which case this usage is ignored.

  – SIGQUIT

    Caught, sets the default signal handler and calls exit handler with an exit code of 128+SIGQUIT. The exit handler will dump user's context and take the default signal action.

  – SIGFPE

    Caught, sets the default signal handler and calls exit handler with an exit code of 128+SIGFPE. The exit handler will dump user's context and take the default signal action.

  – SIGSEGV

    Caught, sets the default signal handler and calls exit handler with an exit code of 128+SIGSEGV. The exit handler will dump user's context and take the default signal action.

  – SIGPIPE

    Caught by message passing library to manage message traffic. If the user application catches this signal, it should call the registered message passing signal handler before returning to the main code.

  – SIGALRM

    Caught by message passing library to manage message traffic. If users provide their own interval timing mechanism, they should arrange to call the POE message passing signal handler approximately every 200-800 milliseconds. Message passing calls from user programs may be blocked until the POE message passing signal handler is called.

    If the user application catches this signal but doesn't do interval timing, it should call the registered message passing signal handler before returning to the main code.

  – SIGTERM

    Caught and exits with an exit code of 128+SIGTERM. This is also used by POE to signal orderly termination of a parallel job. If it must be caught by the user, please read carefully the section on program termination (below).

- SIGSTOP

  Default action

- SIGTSTP

  Default action

- SIGCONT

  Default action

- SIGIO

  Caught by message passing library to manage message traffic. If the user application catches this signal, it should call the registered message passing signal handler before returning to the main code.

- SIGPWR

  Caught, sets the default signal handler and calls exit handler with an exit code of 128+SIGPWR. The exit handler will dump user's context and take default signal action.

- SIGDANGER

  Caught and exits with an exit code of 128+SIGDANGER.

- Replacement exit/atexit

  POE requires its own versions of the library exit()/atexit() functions, and expects to load them dynamically from its own version of libc.a in /usr/lpp/ppe.poe/lib, so the user should not code his/her own exit function to override the library function. This is to synchronize profiling and to provide barrier synchronization upon exit.

Program Interrupts Must Be Handled

The message passing library uses an interval timer to manage message traffic, specifically to ensure that all messages are eventually sent and received. When this interval timer expires, a SIGALRM signal is sent to the program, interrupting whatever computation is in progress. The message passing library has a signal handler set, and normally handles the signal and returns to the user's program without the program's knowledge. However, the following library and system calls are interrupted and do not complete normally. The user is responsible for testing whether an interrupt occurred and recovering from the interrupt. In many cases, this is accomplished by just retrying the call.

- sleep(see note below)/usleep/nsleep
- select
- open/close/fopen/fclose
- pause
- sigpause
- accept
- connect
- recv/recvfrom/recvmsg
- send/sendto/sendmsg
- aio_read/aio_write/aio_suspend
- fork
- system
- exec/execv/...
- msem_lock/semop

- AIX msg... routines
- poll

Note: The normal timer interval is less than 500 milliseconds. So a sleep call (with time specified in seconds) will return the original sleep interval, due to rounding, and can't be used to determine how much time remains in the interval. The functions usleep and nsleep should be used. See also the sample SLEEP program described in Appendix H, “Using Signals and Parallel Environment Programs” on page 159.

With the exception of sleep, system and exec, the routines listed above set the system error indicator (the variable errno) to EINTR, which can be tested by the user's program. See the sample SELECT function described in Appendix H, “Using Signals and Parallel Environment Programs” on page 159.

Normal file read and write are restarted automatically by AIX, and should not require any special treatment by the user.

The **system** and **fork** calls create a new process in which the interval timer is still running. If a fork is followed by an exec (which is what **system** does), the signal handler for the timer is overlaid, and the process will be terminated when the interval timer expires. To handle this for the **system** call, temporarily turn the interval timer off (using the **alarm**(0) call) before the call, and turn it on again (**ualarm**(500000) will do) after the **system** call.

To handle the interval timer for a forked child, merely turn off the interval timer via **alarm**(0) in the child.

There are other restrictions on fork described below.

Let POE Handle Signals When Possible

Programs that handle signals must coordinate with POE's handling of most of the common signals (see above).

The message passing library sets up signal handlers for SIGALRM, SIGIO and SIGPIPE to manage message passing activity. A user program may install a handler for any or all of these signals, but should save the address of and invoke the POE signal handler before returning to the interrupted code. The sigaction() function returns the required structure. Also, the user should code the SA_RESTART flag in his/her signal handler.

Users may NOT issue message passing calls from signal handlers. Also, many AIX library calls are not "signal safe", and should not be issued from signal handlers. Check the AIX Technical Reference (function sigaction()) for a list of AIX functions callable from signal handlers.

POE sets up signal handlers for all the signals that normally terminate program execution. It does this so that it can terminate the entire parallel job in an orderly fashion if one task terminates abnormally (via signal). A user program may install a handler for any or all of these signals, but should save the address of the POE signal handler. If the user program decides to terminate, it should call the POE signal handler. If the user program decides not to terminate, it should just return to the interrupted code. SIGTERM is used by POE to shutdown the parallel job in a variety of abnormal circumstances, and should be allowed to terminate the job.

The POE home node converts a user's SIGTSTP signal (Ctrl-z) to a SIGSTOP signal to all the remote nodes, and passes the SIGCONT signal sent by the **fg** or **bg** command to all the remote nodes to restart the job.

Don't Hard Code File Descriptor Numbers

Do not use hard coded file descriptor numbers beyond those specified by STDIN, STDOUT and STDERR.

POE opens several files and uses file descriptors as message passing handles. These are allocated before the user gets control, so the first file descriptor allocated to a user is unpredictable.

POE Gets Control First And Handles Process Initialization

User's main program is called after POE sets up its environment via the entry point mp_main(). mp_main() initializes the message passing library, sets up signal handlers, sets up an atexit routine, and initializes VT trace data collection before calling the user main. The user should not implement his/her own initialization procedure to replace the crt0.o functions provided by POE. POE does not provide re-entrant initialization functions. This means the user may not use the thread library or any functions that require use of the re-entrant C library (libc_r.a).

Termination Of A Parallel Job

POE provides for orderly termination of a parallel job, so that all tasks terminate at the same time. This is accomplished in the atexit routine registered by the initialization function, mp_main(). For normal exits (codes 0, 2-127), the atexit routine sends a control message to the POE home node, and waits for a positive response. For abnormal exits and those which don't go through the atexit routine, the **pmd** daemon catches the exit code and sends a control message to the POE home node.

For normal exits, when POE gets a control message for each task, it responds to each node, allowing that node to exit normally with its individual exit code. The **pmd** daemon monitors the exit code and passes it back to the POE home node for presentation to the user.

For abnormal exits and those detected by **pmd**, POE sends a message to each **pmd** asking that it send a SIGTERM signal to its task, thereby terminating the task. When the task finally exits, **pmd** sends its exit code back to the POE home node and exits itself.

User-initiated termination of the POE home node via SIGINT (Ctrl-c) and/or SIGQUIT (Ctrl-\) causes a message to be sent to **pmd** asking that the appropriate signal be sent to the parallel task. Again, **pmd** waits for the task to die then terminates itself.

POE Does Not Support Threads And The Re-entrant C Library

POE provides its own version of crt0.o, and is compiled with the non re-entrant library (libc.a). This is true for the message passing library (libmpi.a) as well. As a result, thread programs and those that use libc_r.a (the re-entrant library) are not supported.

Forks Are Limited

As described above, if a process forks, the forked child inherits the running timer. It should turn the timer off before exec'ing another program. If the forked child does not exec another program, it should be aware that an atexit routine has been registered for the parent which is also inherited by the child. In most cases, the atexit routine will request POE to terminate the task (parent). A forked child should terminate with an _exit(0) system call to prevent the atexit routine from being called. Also, if the forked parent terminates before the child, the child process will not be cleaned up by POE.

A forked child MAY NOT call the message passing library.

The User's Program Can't Run As Root

To prevent uncontrolled root access to the entire SP system, POE checks to see that the user is not root as part of its authentication.

AIX Function Limitations

The use of the following AIX functions may be limited, but no formal testing has been done:

- wide character sets
- shared memory - The message passing library uses shared memory for adapter mapping. The user can use the remaining data segments as desired.
- getuinfo will not show terminal information, since the user's program running in the parallel partition does not have an attached terminal.

Programs using RPC should install the fix for AIX APAR #IX49770.

Shell Execution

You can have POE run a shell script which is loaded and run on the remote nodes as if it were a binary file.

If the POE home node process is not started under the Korn shell, mounted file system names may not be mapped correctly to the names defined for the Automounter running on the POWERparallel SP. See the *IBM Parallel Environment for AIX: Operation and Use Guide* for a discussion of alternative name mapping techniques.

The program executed by POE on the parallel nodes does not run in a shell on those nodes. Redirection and piping of STDIO applies to the POE home node (**poe** binary), and not the user's code. If shell processing of a command line is desired on the remote nodes, invoke a shell script on the remote nodes to provide the desired preprocessing before the user's application is executed.

Standard I/O Requires Special Attention

The user's program running on the remote nodes has no controlling terminal. STDIN and STDOUT, STDERR are always piped.

Programs that depend on piping standard input or standard output as part of a processing sequence may wish to bypass the home node poe binary. Running the **poe** command (or starting a program compiled with mpcc or mpxlf or mpCC) causes the poe binary to be loaded on the machine on which the command was

typed (the POE home node). The poe binary, in turn, starts a daemon named pmd on each parallel node assigned to run the job, and then requests **pmd** to run the user's executable (via fork and exec). The poe binary reads STDIN and passes it to each of the parallel tasks via a TCP/IP socket connection to the **pmd** daemon, which pipes it to the user. Similarly, STDOUT and STDERR from the user are piped to **pmd** and sent on the socket back to the home node, where it is written to the poe binary's STDOUT and STDERR descriptors. In cases where the user knows that the task reading STDIN or writing STDOUT must be on the same node (processor) as the poe binary (the poe home node), named pipes can be used to bypass poe's reading and forwarding STDIN and STDOUT.

If STDIN is piped to the poe binary (via ordinary pipes), and the user's application is linked with the message passing library, (via mpcc, mpxlf, or mpCC), the environment variable MP_HOLD_STDIN should be set to "yes", to allow poe to initialize the message passing library before handling the STDIN file.

STDIN/STDOUT piping example

The following two scripts show how STDIN and STDOUT can be piped directly between prior and post processing steps, bypassing the POE home node process. This example assumes that parallel task 0 is known or forced to be on the same node as the POE home node.

The script compute_home runs on the home node; the script compute_parallel runs on the parallel nodes (those running tasks 0 through n-1).

```
compute_home:
#! /bin/ksh
# Example script compute_home runs three processes:
#    data_generator creates/gets data and writes to stdout
#    data_processor is a parallel program that reads data
#      from stdin, processes it in parallel, and writes
#      the results to stdout.
#    data_consumer reads data from stdin and summarizes it
#
mkfifo poe_in_$$
mkfifo poe_out_$$
export MP_STDOUTMODE=0
export MP_STDINMODE=0
data_generator >poe_in_$$ | \n
     poe compute_parallel poe_in_$$ poe_out_$$ data_processor |  \n
     data_consumer <poe_out_$$
 rc=$?
 rm poe_in_$$
 rm poe_out_$$
 exit rc
```

```
compute_parallel:
#! /bin/ksh
# Example script compute_parallel is a shell script that
#    takes the following arguments:
#    1) name of input named pipe (stdin)
#    2) name of output named pipe (stdout)
#    3) name of program to be run (and arguments)
#
poe_in=$1
poe_out=$2
shift 2
$*   <$poe_in   >$poe_out
```

Network Tuning Considerations

Programs generating large volumes of STDOUT or STDERR may overload the home node. As described previously, standard output and standard error files generated by a user's program are piped to **pmd**, then forwarded to the poe binary via a TCP/IP socket. It is possible to generate so much data that the IP message buffers on the home node will be exhausted, and the poe binary will hang (and possibly the entire node will hang). Note that the option -stdoutmode (environment variable MP_STDOUTMODE) controls which output stream is displayed by the poe binary, but does not limit the standard output traffic received from the remote nodes, even if set to display the output of just one node.

The POE environment variable MP_SNDBUF may be used to override the default network settings for the size of the TCP buffers used.

Users with large volumes of standard I/O should work with their network administrator to establish appropriate TCP/IP tuning parameters. They may also wish to examine if using named pipes as described above is appropriate for their application.

Reserved environment variables

Environment variables starting with MP_ are intended for use by POE, and should be set only as instructed in the documentation. POE also uses a handful of MP_... environment variables for internal purposes, which should not be interfered with.

AIX Message Catalog considerations

POE assumes that NLSPATH contains the appropriate POE message catalogs, even if LANG is set to "C" or is unset. Duplicate message catalogs are provided for languages "En_US" , "en_US", and "C".

Appendix H. Using Signals and Parallel Environment Programs

Any Parallel Environment(PE) call that is interruptible by a signal may not behave as expected because the message passing subsystem uses timer signals. For example, the user's program does not sleep for the full time but returns quickly with an error code of EINTR. This indicates the sleep was interrupted by a signal. This happens for select system call as well.

The following are some sample programs to replace sleep and select.

Sample Replacement Sleep Program

The following sample replacement program for sleep guarantees to sleep for the specified interval, even if interrupted.

Sleep Example

```
#include <errno.h>
#include <sys/time.h>
int SLEEP(int amount)
{
 struct timestruc_t Requested, Remaining;
 double famount = amount;
 int rc;
 while (famount < 0.0) {
  Requested.tv_sec = (int) famount;
  Requested.tv_nsec =
   (int) ((famount - Requested.tv_sec)*1000000000.);
  rc = nsleep ( &Requested, &Remaining );
  if ((rc == -1) && (errno == EINTR)) {
   /* Sleep interrupted.  Resume it */
   famount = Remaining.tv_sec + Remaining.tv_nsec /
       1000000000.;
   continue;
  }
  else /* Completed sleep.  Set return to zero */
  {
    return (0);
  }
 }    /* end of while */

 /* famount = 0; exit */
 return (0);
}
```

Sample Replacement Select Program

The following is a sample replacement program for select. Select worries about restoring the status of the file descriptor bit masks, as well as handling the remaining time after an interrupt.

Select Example

© Copyright IBM Corp. 1995

```
#include <stdio.h>
#include <sys/select.h>
#include <sys/types.h>
#include <sys/time.h>
#include <errno.h>

int SELECT(int maxfds, fd_set *reads, fd_set *writes, fd_set *errors,
  struct timeval *timeout)
{
 struct timestruc_t Timer1, Timer2;
 struct timeval timetogo;
 static fd_set readcopy;
 static fd_set writecopy;
 static fd_set errcopy;
 int rc;
 double worktime;
 double remaining;

 /* If we get interrupted, will need to restore select bits */
 if (reads) bcopy(reads,&readcopy,sizeof(fd_set));
 if (writes) bcopy(writes,&writecopy,sizeof(fd_set));
 if (errors) bcopy(errors,&errcopy,sizeof(fd_set));

 /* two cases: if timeout specifies a time structure, we
  need to worry about timeouts.  Otherwise, we can
  ignore it */

 if (timeout == NULL) {
  while (TRUE) {
   rc = select(maxfds,reads,writes,errors,NULL);
   if ((rc == -1) && (errno == EINTR)) { /* interrupted */
    if (reads) bcopy(&readcopy,reads,sizeof(fd_set));
    if (writes) bcopy(&writecopy,writes,sizeof(fd_set));
    if (errors) bcopy(&errcopy,errors,sizeof(fd_set));
    continue;
   }
   else return(rc);
  }
 }
 else  { /* timeout is not null */
  timetogo.tv_sec = timeout->tv_sec;
  timetogo.tv_usec = timeout->tv_usec;
  remaining = timetogo.tv_sec + timetogo.tv_usec/1000000.;
/*
  fprintf(stderr,"remaining time = %f\n",remaining);
  fflush(stderr);
*/
  gettimer(TIMEOFDAY, &Timer2);
  while (TRUE) {
   Timer1.tv_sec = Timer2.tv_sec;
   Timer1.tv_nsec = Timer2.tv_nsec;
   rc = select(maxfds,reads,writes,errors,&timetogo);
   if ((rc == -1) && (errno == EINTR)) { /* interrupted */
    gettimer(TIMEOFDAY, &Timer2);
    /* compute amount remaining */
    worktime = (Timer2.tv_sec - Timer1.tv_sec) +
     (Timer2.tv_nsec - Timer1.tv_nsec)/1000000000.;
    remaining = remaining - worktime;
```

```
    timetogo.tv_sec = (int) remaining;
    timetogo.tv_usec = (int) ((remaining - timetogo.tv_sec)*
        1000000.);
    /* restore the select bits */
    if (reads) bcopy(&readcopy,reads,sizeof(fd_set));
    if (writes) bcopy(&writecopy,writes,sizeof(fd_set));
    if (errors) bcopy(&errcopy,errors,sizeof(fd_set));
    continue;
   }
   else return(rc);
  }
 }
}
```