Distribution Category:
Mathematics and
Computer Science (UC-405)

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439

**ANL-000**

# MPICH ADI Implementation Reference Manual

## Draft

by

*William Gropp*
*Ewing Lusk*
*Mathematics and Computer Science Division*
*Argonne National Laboratory*

February 1, 1996

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

# Contents

**1 Introduction** **1**

**2 ADI routines** **1**
2.1 Include file structure . . . 1
2.2 Core message-passing routines . . . 1
MPID_Cancel . . . 1
MPID_Check_device . . . 2
MPID_Complete_send . . . 2
MPID_Complete_recv . . . 3
MPID_End . . . 3
MPID_Init . . . 3
MPID_Iprobe . . . 4
MPID_Myrank . . . 5
MPID_Mysize . . . 5
MPID_Post_send . . . 5
MPID_Post_send_ready . . . 6
MPID_Post_send_sync . . . 6
MPID_Test_recv . . . 7
MPID_Test_send . . . 7
MPID_Test_request . . . 8
MPID_PACK_IN_ADVANCE . . . 8
MPID_RETURN_PACKED . . . 8
2.3 Core service routines . . . 9
MPID_Abort . . . 9
MPID_Alloc_recv_handle . . . 9
MPID_Alloc_send_handle . . . 9
MPID_Free_recv_handle . . . 10
MPID_Free_send_handle . . . 10
MPID_Reuse_recv_handle . . . 11
MPID_Reuse_send_handle . . . 11
2.4 Routines used in the Chameleon versions . . . 11
MPID_Check_incoming . . . 11
MPID_Clr_completed . . . 12
MPID_Set_completed . . . 12
2.5 Extension routines . . . 12
2.5.1 Point-to-point . . . 13
MPID_Blocking_recv . . . 13
MPID_Blocking_send . . . 13
MPID_Blocking_send_ready . . . 13
MPID_Probe . . . 14
2.5.2 Collective . . . 14
Collective . . . 15
MPID_Barrier . . . 15
MPID_Comm_free . . . 16
MPID_Comm_init . . . 16
MPID_Reduce_sum_double . . . 16
MPID_Reduce_sum_int . . . 17
2.5.3 Environment . . . 17
MPID_Node_name . . . 17
MPID_Version_name . . . 18
MPID_Wtick . . . 18
MPID_Wtime . . . 18

MPID_Wtime_is_global . . . . . 19
2.6 Heterogenity support . . . . . 19
MPID_Dest_byte_order . . . . . 19
MPID_HAS_HETERO . . . . . 20
MPID_IS_HETERO . . . . . 20
MPID_Procinfo . . . . . 20
2.7 Miscellanenous discussion . . . . . 20
ADI . . . . . 20
Chameleon-MPI . . . . . 22
Packet . . . . . 22
Threads . . . . . 23
MPID_Flags . . . . . 24

# 1 Introduction

This document contains detailed documentation on the routines that are part of the ADI part of the MPICH model MPI implementation. The ADI is a multi-level interface; there are some routines (the *core*) that must be implemented, and an additional set (the *extensions*) that can be implemented as necessary and desired for improved performance (see the collective operations below). In addition, the ADI may itself be implemented on a lower-level interface called the "channel" interface; this is detailed in a separate report.

As an alternate to this manual, the reader should consider using the script `mpiman`; this is a script that uses `xman` to provide a X11 Window System interface to the data in this manual.

The ADI definition presented here is still evolving; while we intend to make no major incompatible changes, we can not rule them out. Unlike MPI, the ADI is not a standard but is rather part of a research project.

# 2 ADI routines

The ADI routines are divided into several categories. The core message-passing routines provide the basic support for sending and receiving messages, as well as testing for messages, initializing and terminating the package, and getting information about the participating processes. A set of "extension" routines provides access to more advanced operations when the underlying hardware and software can support them. This split allows implementers to choose a relatively small set for a first implementation and then add features to get additional performance. We emphasize that all of MPI is supported by the core set of routines; the extension routines are present to provide better performance, not to provide more complete functionallity.

The first argument of most of these routines requires some explanation. This argument is a void pointer that is returned from the initialization routine `MPI_Init`. In many cases, this pointer can be null and ignored. Its purpose is to provide a way for multiple ADI implementations to be used by a single MPI program. For example, a program running on a cluster consisting of an MPP and an external graphics workstation would like to use the MPP's high-performance system within the MPP and TCP/IP to the workstation. This initial variable allows this if it contains pointers to the appropriate ADI functions to use. Currently, there is no implementation that uses this feature, but one is planned for the end of 1994.

## 2.1 Include file structure

The file `dm.h` defines the interface to the ADI for the MPICH code. It contains definitions of the form

```
#define MPID_Test_request( ctx, request ) MPID_MY_Test_request( request )
```

that define the interfaces (e.g., `MPID_Test_request`) in terms of particular routines (e.g., `MPID_MY_Test_request`). It also contains information about the capabilities of the ADI, such as whether the ADI can operate in a heterogeneous environment.

The file `mpid.h` contains any definitions that are specific to the ADI itelf; some MPICH routines may include this file as well.

Neither of these files will be accessed by users compiling MPI programs; in other words, the include file `mpi.h` does not reference these files.

## 2.2 Core message-passing routines

**MPID_Cancel** **MPID_Cancel**

**MPID_Cancel** — Cancels a pending communication operation.

### Input Parameter

**ctx** ADI context returned by MPID_Init
**r** pointer to an API message request

### Synopsis

```
void MPID_Cancel( void * ctx, MPIR_CHANDLE *r )
```

### Notes

The actual freeing of internal space must be done with the appropriate `MPID_FREE_SEND/RECV_HANDLE` calls.

This routine may have little effect. If the message has already been delivered, it has no effect.

---

**MPID_Check_device** **MPID_Check_device**

---

**MPID_Check_device** — Checks for pending operations and handles if found.

### Input Parameter

**blocking** Indicates whether the operation is blocking or not. The values `MPID_BLOCKING` and `MPID_NOT_BLOCKING` can be used.
**ctx** ADI context returned by `MPID_Init`

### Synopsis

```
void MPID_Check_device( void * ctx, int blocking )
```

### Notes

This routine should be called by the API frequently *if* the ADI requires it. For example, if the ADI is not a separate thread and does not use signals to operate asynchronously.

Some ADI routines may also use this to process pending ADI events.

---

**MPID_Complete_send** **MPID_Complete_send**

---

**MPID_Complete_send** — Completes a send of a message.

### Input Parameter

**ctx** ADI context returned by MPID_Init
**dmpi_send_handle**
APIs send handle

### Synopsis

```
int MPID_Complete_send( void * ctx, MPIR_SHANDLE *dmpi_send_handle )
```

### Notes

This routine completes a send begun with `MPID_Post_send`, `MPID_Post_send_ready`, or `MPID_Post_send_sync`.

---

**MPID_Complete_recv** **MPID_Complete_recv**

---

**MPID_Complete_recv** — Completes a receive of a message.

### Input Parameter

**ctx** ADI context returned by `MPID_Init`

**dmpi_recv_handle**
APIs receive handle

### Synopsis

```
int MPID_Complete_recv( void * ctx, MPIR_RHANDLE *dmpi_recv_handle )
```

### Notes

This routine completes a send begun with `MPID_Post_recv`.

---

**MPID_End** **MPID_End**

---

**MPID_End** — Ends the operation of the ADI

### Input Parameter

**ctx** ADI context returned by `MPID_Init`

### Synopsis

```
void MPID_End( void * ctx )
```

### Notes

This routine must be called at the end of the parallel part of a program. Normally, this is just before the program exits. Once `MPID_End` is called, no other MPID routines, including `MPID_Init`, may be called. Note that by the time this routine is called, much of the MPI environment may already have been dismantled. For example, the communicators and datatypes may have been freed before this routine is called.

If the macro `MPID_END_NEEDS_BARRIER` is defined, the code that calls `MPID_End` must call `MPI_BARRIER( MPI_COMM_WORLD )` before calling `MPID_End`. This helps

---

**MPID_Init** **MPID_Init**

---

**MPID_Init** — Initializes the ADI

### Input Parameters

| | |
|---|---|
| **argc** | Pointer to an integer containing the number of arguments |
| **argv** | Pointer to the argument vector |

### Returns

Pointer that may be passed back to these routines (context).

### Synopsis

```
void *MPID_Init( int *argc, char ***argv )
```

### Notes

If called from a C program with main(argc,argv), the correct calling sequence is `MPID_Init(&argc,&argv)`.

This routine may (or may not) also initialize the parallel processes themselves. That is, when this routine is called, it may create the requested number of parallel processes, or it may simply make use of an already existing set. What the case is, and how the number of processes is specified, is specific to any particular implementation and environment.

As an example, the Chameleon version of the ADI has these properties

```
Transport Layer             -   MPID_Init starts processes
------------------------------------------------------------
p4, pvm, EUIH               -   yes
Paragon                     -   depends on Paragon
Intel i860, Delta, CM5, POE -   no
```

The pointer returned by this routine may be null; it is provided to allow for multiple, simultaneous ADIs. For example, if the pointer returned contains a structure with pointers to the ADI routines, and the macros for the operations choose the actual function from this structure, then a single user-application can easily switch between ADIs, depending, for example, on the MPI Communicator in use.

---

**MPID_Iprobe** **MPID_Iprobe**

---

**MPID_Iprobe** — Returns whether a message of the specified type is available

### Input Parameters

| | |
|---|---|
| **ctx** | ADI context returned by `MPID_Init` |
| **tag** | Message tag |
| **source** | Message source (sending processor). Measured in terms of rank of the communicator |
| **context_id** | Context id value of communicator. |

### Output Parameters

| | |
|---|---|
| **found** | Set to true if a message is available, false otherwise |
| **status** | MPI status object; has values set if found is true |

### Synopsis

```
int MPID_Iprobe( void * ctx, int tag, int source, int context_id,
int *found, MPI_Status *status )
```

### Notes

Since this routine does not have a communicator argument, there is no easy way to convert the `source` argument from the value here (local to the communicator that `context_id` belongs to) and to the source relative to `MPI_COMM_WORLD`. To force `source` to be relative to `MPI_COMM_WORLD` instead of the communicator, the ADI should define `MPID_NEEDS_WORLD_SRC_INDICES` in `dm.h`.

---

**MPID_Myrank** **MPID_Myrank**

---

**MPID_Myrank** — Provides the rank of the calling process

### Input Parameter

**ctx** ADI context returned by `MPID_Init`

### Output Parameter

**rank** rank, in the range 0 to `Mysize-1` (see `MPID_Mysize`) of the calling process among all of the processes.

### Synopsis

```
void MPID_Myrank( void * ctx, int *rank )
```

---

**MPID_Mysize** **MPID_Mysize**

---

**MPID_Mysize** — Provides the number of processes

### Input Parameter

**ctx** ADI context returned by `MPID_Init`

### Output Parameter

**size** number of processes in the set of processes.

### Synopsis

```
void MPID_Mysize( void * ctx, int *size )
```

---

**MPID_Post_send** **MPID_Post_send**

---

**MPID_Post_send** — Starts a send of a message

### Input Parameter

**ctx** ADI context returned by `MPID_Init`
**dmpi_recv_handle**
APIs send handle

### Synopsis

```
int MPID_Post_send( void * ctx, MPIR_SHANDLE *dmpi_send_handle )
```

### Notes

This routine only begins the send. `MPID_Complete_send` must be called to finish the send, even if it was a blocking operation

---

**MPID_Post_send_ready** **MPID_Post_send_ready**

---

**MPID_Post_send_ready** — Starts a send of a message with ready mode

### Input Parameter

**ctx** ADI context returned by `MPID_Init`
**dmpi_recv_handle**
APIs send handle

### Synopsis

```
int MPID_Post_send_ready( void * ctx, MPIR_SHANDLE *dmpi_send_handle )
```

### Notes

This routine only begins the send. `MPID_Complete_send` must be called to finish the send, even if it was a blocking operation. The matching receive should already be posted.

It is legal to implement this with `MPID_Post_send`.

---

**MPID_Post_send_sync** **MPID_Post_send_sync**

---

**MPID_Post_send_sync** — Starts a send of a message with synchronous mode

### Input Parameter

**ctx** ADI context returned by `MPID_Init`
**dmpi_recv_handle**
APIs send handle

### Synopsis

```
int MPID_Post_send_sync( void * ctx, MPIR_SHANDLE *dmpi_send_handle )
```

### Notes

This routine only begins the send. `MPID_Complete_send` must be called to finish the send, even if it was a blocking operation. This send will not complete until the destination *begins* the matching receive.

---

**MPID_Test_recv** **MPID_Test_recv**

---

**MPID_Test_recv** — Tests for the completion of a receive request.

### Input Parameter

**ctx** ADI context returned by `MPID_Init`
**dmpi_recv_handle**
APIs recv handle

### Return value

True (!= 0) if the request has completed, False (0) otherwise.

### Notes

This routine must return true if the request either has completed or would complete if `MPID_Check_device` were called. In other words, it must be possible to have a loop

```
while (!MPID_Test_recv( ctx, request )) ;
```

that will terminate when the message arrives, without requiring an explicit call to another ADI routine.

---

**MPID_Test_send** **MPID_Test_send**

---

**MPID_Test_send** — Tests for the completion of a send request.

### Input Parameter

**ctx** ADI context returned by `MPID_Init`
**dmpi_send_handle**
APIs send handle

### Return value

True (!= 0) if the request has completed, False (0) otherwise.

### Notes

This routine must return true if the request either has completed or would complete if `MPID_Check_device` were called. In other words, it must be possible to have a loop

```
while (!MPID_Test_send( ctx, request )) ;
```

that will terminate when the message has been sent, without requiring an explicit call to another ADI routine.

---

**MPID_Test_request** **MPID_Test_request**

---

**MPID_Test_request** — Tests a request for completion.

## Input Parameter

**ctx** ADI context returned by `MPID_Init`
**request** MPI_Request handle to test.

## Return value

True (!= 0) if the request has completed, False (0) otherwise.

## Notes

This routine must return true if the request either has completed or would complete if `MPID_Check_device` were called. In other words, it must be possible to have a loop

```
while (!MPID_Test_request( ctx, request )) ;
```

that will terminate when the message has been sent or received, without requiring an explicit call to another ADI routine.

---

**MPID_PACK_IN_ADVANCE** **MPID_PACK_IN_ADVANCE**

---

**MPID_PACK_IN_ADVANCE** — Indicates that the ADI prefers to be given a single buffer

## Notes

This is a macro that indicates that the data will be provided in the data-area specified by the "start" field of the device handle. If the message is not contiguous, the API is responsible for packing the message (it may use `MPIR_Pack`).

More advanced ADI implementations may inform the calling routines (the API code) that the ADI can handle more complicated data, such as `MPID_DOES_STRIDED`. The exact interface for this has not been determined.

---

**MPID_RETURN_PACKED** **MPID_RETURN_PACKED**

---

**MPID_RETURN_PACKED** — Indicates that the ADI prefers to return a single buffer

## Notes

This is a macro that indicates that the data will be returned in the data-area specified by the "start" field of the device handle. If the message is not contiguous, the API is responsible for unpacking the message (with the routines MPIR_SetupUnPackMessage and MPIR_UnPackMessage)

## 2.3 Core service routines

These routines provide some required services; principly with the allocation and manipulation of the ADI part of the device data structures.

---

**MPID_Abort** **MPID_Abort**

---

**MPID_Abort** — Terminates a parallel program

### Input Parameter

| | |
|---|---|
| **code** | Value to return to invocation environment as exit status |
| **ctx** | ADI context returned by `MPID_Init` |

### Synopsis

```
void MPID_Abort( void * ctx, int code )
```

### Notes

This call terminates all processes in a parallel program.

It is not always possible to return code as the status.

---

**MPID_Alloc_recv_handle** **MPID_Alloc_recv_handle**

---

**MPID_Alloc_recv_handle** — Allocates and initializes device recv handle

### Input Parameter

| | |
|---|---|
| **ctx** | ADI context returned by `MPID_Init` |
| **a** | Address of device recv handle in APIs send structure |

### Synopsis

```
void MPID_Alloc_recv_handle( void * ctx, MPID_SHANDLE *mpid_recv_handle )
```

### Notes

In systems where the device handle is statically allocated, this routine may be used to initialize the fields of that handle. In particular, it may set fields such as "completed" to false.

---

**MPID_Alloc_send_handle** **MPID_Alloc_send_handle**

---

**MPID_Alloc_send_handle** — Allocates and initializes device send handle

### Input Parameter

| | |
|---|---|
| **ctx** | ADI context returned by `MPID_Init` |
| **a** | Address of device send handle in APIs send structure |

### Synopsis

```
void MPID_Alloc_send_handle( void * ctx, MPID_SHANDLE *mpid_send_handle )
```

### Notes

In systems where the device handle is statically allocated, this routine may be used to initialize the fields of that handle. In particular, it may set fields such as "completed" to false.

---

**MPID_Free_recv_handle** **MPID_Free_recv_handle**

---

**MPID_Free_recv_handle** — Frees device recv handle

### Input Parameter

| | |
|---|---|
| **ctx** | ADI context returned by `MPID_Init` |
| **a** | Address of device recv handle in APIs recv structure |

### Synopsis

```
void MPID_Free_recv_handle( void * ctx, MPID_RHANDLE *mpid_recv_handle )
```

### Notes

In systems where the device handle is statically allocated, this routine may be used to free any areas that may have been created by the recv operation.

---

**MPID_Free_send_handle** **MPID_Free_send_handle**

---

**MPID_Free_send_handle** — Frees device send handle

### Input Parameter

| | |
|---|---|
| **ctx** | ADI context returned by `MPID_Init` |
| **a** | Address of device send handle in APIs send structure |

### Synopsis

```
void MPID_Free_send_handle( void * ctx, MPID_SHANDLE *mpid_send_handle )
```

### Notes

In systems where the device handle is statically allocated, this routine may be used to free any areas that may have been created by the send operation.

**MPID_Reuse_recv_handle** **MPID_Reuse_recv_handle**

**MPID_Reuse_recv_handle** — Reinitializes device recv handle

### Input Parameter

**ctx** ADI context returned by `MPID_Init`
**a** Address of device recv handle in APIs send structure

### Synopsis

```
void MPID_Reuse_recv_handle( void * ctx, MPID_RHANDLE *mpid_recv_handle )
```

### Notes

In systems where the device handle is statically allocated, this routine may be used to reset fields such as "completed" to false. This routine is used for persistent message-passing requests

**MPID_Reuse_send_handle** **MPID_Reuse_send_handle**

**MPID_Reuse_send_handle** — Reinitializes device send handle

### Input Parameter

**ctx** ADI context returned by `MPID_Init`
**a** Address of device send handle in APIs send structure

### Synopsis

```
void MPID_Reuse_send_handle( void * ctx, MPID_SHANDLE *mpid_send_handle )
```

### Notes

In systems where the device handle is statically allocated, this routine may be used to reset fields such as "completed" to false. This routine is used for persistent message-passing requests

## 2.4 Routines used in the Chameleon versions

**MPID_Check_incoming** **MPID_Check_incoming**

**MPID_Check_incoming**

### Input Parameter

**ctx** ADI context returned by `MPID_Init`
**blocking** Indicates whether the operation is blocking or not. The values `MPID_BLOCKING` and `MPID_NOT_BLOCKING` can be used.

### Returns

-1 if nonblocking and no messages pending

### Synopsis

```
int MPID_Check_incoming( void * ctx, MPID_BLOCKING_TYPE blocking )
```

### Notes

This routine handles testing for a single incoming message. For people interested in active messages, this routine is essentially the handler for the incoming active messages.

---

**MPID_Clr_completed** **MPID_Clr_completed**

---

**MPID_Clr_completed** — Mark a request as not completed

### Input Parameter

| | |
|---|---|
| **ctx** | ADI context returned by `MPID_Init` |
| **request** | MPI_Request handle to set. |

### Notes

This is used within the Chameleon ADI implementation only

---

**MPID_Set_completed** **MPID_Set_completed**

---

**MPID_Set_completed** — Mark a request as completed

### Input Parameter

| | |
|---|---|
| **ctx** | ADI context returned by `MPID_Init` |
| **request** | MPI_Request handle to set. |

### Notes

This is used within the Chameleon ADI implementation only

## 2.5 Extension routines

The routines in this section do not need to be provided since they can either be formed from other routines (e.g., blocking recv) or return a "not available" (e.g., version name).

Some of these *must* be defined as macros because the code that calls the ADI will use `#ifdef` to test for the existence of a feature at compile time. For example, the code to conditionally use the ADI-supplied barrier is

```
#ifdef MPID_Barrier
  if (comm->ADIBarrier) {
      MPID_Barrier( comm->ADIctx, comm );
      return MPI_SUCCESS;
      }
#endif
```

Others can be defined in terms of the core routines. For example

```
#define MPID_Probe( tag, source, context_id, status ) { \
int found; \
while (1) { \
    MPID_Iprobe( tag, source, context_id, &found, status ); \
    if (found) break; \
    MPID_Check_incoming( MPID_BLOCKING ); } }
```

### 2.5.1 Point-to-point

---

**MPID_Blocking_recv** **MPID_Blocking_recv**

---

**MPID_Blocking_recv** — Performs a blocking receive

#### Input Parameter

**ctx** ADI context returned by `MPID_Init`

**dmpi_recv_handle**
APIs receive handle

#### Synopsis

```
int MPID_Blocking_recv( void * ctx, MPIR_RHANDLE *dmpi_recv_handle )
```

---

**MPID_Blocking_send** **MPID_Blocking_send**

---

**MPID_Blocking_send** — Performs a blocking send.

#### Input Parameter

**ctx** ADI context returned by `MPID_Init`

**dmpi_send_handle**
APIs send handle

#### Synopsis

```
int MPID_Blocking_send( void * ctx, MPIR_SHANDLE *dmpi_send_handle )
```

---

**MPID_Blocking_send_ready** **MPID_Blocking_send_ready**

---

**MPID_Blocking_send_ready** — Performs a ready blocking send

### Input Parameter

**ctx** ADI context returned by `MPID_Init`
**dmpi_send_handle**
APIs send handle

### Synopsis

```
int MPID_Blocking_send_ready( void * ctx, MPIR_SHANDLE *dmpi_send_handle )
```

### Notes

This may be implemented as `MPID_Blocking_send`

---

**MPID_Probe** **MPID_Probe**

---

**MPID_Probe** — Returns status for a matching message; waits if message is not available.

### Input Parameters

**ctx** ADI context returned by `MPID_Init`
**tag** Message tag
**source** Message source (sending processor). Measured in terms of rank of the communicator
**context_id** Context id value of communicator.

### Output Parameters

**status** MPI status object; has values set if found is true

### Synopsis

```
int MPID_Probe( void * ctx, int tag, int source, int context_id,
MPI_Status *status )
```

### Notes

Since this routine does not have a communicator argument, there is no easy way to convert the `source` argument from the value here (local to the communicator that `context_id` belongs to) and to the source relative to `MPI_COMM_WORLD`. To force `source` to be relative to `MPI_COMM_WORLD` instead of the communicator, the ADI should define `MPID_NEEDS_WORLD_SRC_INDICES` in `dm.h`.

#### 2.5.2 Collective

There are a number of systems on which some relatively simple code can substantially improve the performance of important collection operations. One example is the TMC CM-5, where a special network allows some operations in `MPI_COMM_WORLD` to be done more efficiently. Another important case is systems with shared memory or a global address space, where collective operations can be done in and with shared memory.

**Collective** **Collective**

**Collective** — Support in the ADI for collective operations

## Discussion

On some systems, there is special hardware for some collective operations. On others, there are special techniques that can be applied to significantly speed up some collective operations.

It is not necessary to support ANY collective operations within the ADI. However, the ADI does provide a way for the ADI to perform some operations. In addition, these operations may be provided for just MPI_COMM_WORLD or for any communicator.

The ADI provides for the following classes of collective operations

**MPID_Barrier**
: Barrier

**MPID_Reduce_op_type**
: Reduction (op == sum, prod, etc, type = double, int, etc). Define MPID_Reduce if ANY are defined

**MPID_Scan_op_type**
: Scan reduction. Define `MPID_Scan` if *any* are defined.

**MPID_Bcast** Broadcast (1 to all)

**MPID_AlltoAll**
: All to all

These routines all handle a *single element*, since this is an important and common case.

Since these may require the creation of special data structures, the ADI routine `MPID_Comm_init` must be called before these may be used. `MPID_Comm_free` must be called when a communicator is destroyed to ensure that any resources are returned.

All of these "routines" are actually macros, allowing the API code to contain code like

```
#ifdef MPID_Barrier
if (comm->AdiBarrier) MPID_Barrier( comm->AdiCtx, comm );
else
#endif
{
... code using pt-2-pt
}
```

Not yet decided: are the reduction and scan reduced to a single root or to all (I"d prefer to all).

**MPID_Barrier** **MPID_Barrier**

**MPID_Barrier** — Performs a barrier synchronization

## Input Parameters

**ctx** ADI context returned by `MPID_Init`
**comm** MPI Communicator

## Synopsis

```
void MPID_Barrier( void *ctx, MPI_Comm comm )
```

---

**MPID_Comm_free** **MPID_Comm_free**

---

**MPID_Comm_free** — Frees the ADI"s use of a communicator.

## Input Parameters

| | |
|---|---|
| **ctx** | ADI context returned by `MPID_Init` |
| **comm** | MPI Communicator |

## Synopsis

```
void MPID_Comm_free( void *ctx, MPI_Comm comm )
```

## Notes

This routine is responsible for freeing any data areas needed to perform any ADI-supported collective operations (such as `MPID_Barrier`).

---

**MPID_Comm_init** **MPID_Comm_init**

---

**MPID_Comm_init** — Initializes the ADI"s use of a communicator

## Input Parameters

| | |
|---|---|
| **ctx** | ADI context returned by `MPID_Init` |
| **parent** | MPI Communicator of parent |
| **comm** | MPI Communicator |

## Synopsis

```
void MPID_Comm_init( void *ctx, MPI_Comm comm )
```

## Notes

This routine is responsible for initializing any data areas needed to perform any ADI-supported collective operations (such as `MPID_Barrier`). In addition, this routine is collective in the parent communicator. If the parent is `NULL`, comm must be `MPI_COMM_WORLD` and all processes must call this routine.

---

**MPID_Reduce_sum_double** **MPID_Reduce_sum_double**

---

**MPID_Reduce_sum_double** — Reduce a single double value by sum on all processes

## Input Parameters

| | |
|---|---|
| **ctx** | ADI context returned by `MPID_Init` |
| **sendbuf** | Contribution from each process |
| **comm** | MPI Communicator |

### Output Parameter

**recvbuf** Contains (on ALL processes) the output value

### Synopsis

```
void MPID_Reduce_sum_double( void *ctx, double *sendbuf, double *recvbuf,
MPI_Comm comm )
```

---

**MPID_Reduce_sum_int** **MPID_Reduce_sum_int**

---

**MPID_Reduce_sum_int** — Reduce a single int value by sum on all processes

### Input Parameters

**ctx** ADI context returned by `MPID_Init`
**sendbuf** Contribution from each process
**comm** MPI Communicator

### Output Parameter

**recvbuf** Contains (on ALL processes) the output value

### Synopsis

```
void MPID_Reduce_sum_int( void *ctx, int *sendbuf, int *recvbuf,
MPI_Comm comm )
```

#### 2.5.3 Environment

---

**MPID_Node_name** **MPID_Node_name**

---

**MPID_Node_name** — Provides the name of the processor.

### Input Parameter

**ctx** ADI context returned by `MPID_Init`
**len** Length of "name" argument

### Output Parameter

**name** Name of the processor on which this call took place. It should identify the specific hardware when possible

### Synopsis

```
void MPID_Node_name( void * ctx, char *name, int len )
```

---

**MPID_Version_name** **MPID_Version_name**

---

**MPID_Version_name** — Provides a text string describing the version of the ADI

### Input Parameter

**ctx** ADI context returned by `MPID_Init`

### Output Parameter

**name** Name of the version of the ADI. Must be a string of at least 128 characters

### Synopsis

```
void MPID_Version_name( void * ctx, char *name )
```

---

**MPID_Wtick** **MPID_Wtick**

---

**MPID_Wtick** — Returns the resolution of the timer `MPID_Wtime`

### Input Parameter

**ctx** ADI context returned by `MPID_Init`

### Synopsis

```
double MPID_Wtick( void * ctx )
```

---

**MPID_Wtime** **MPID_Wtime**

---

**MPID_Wtime** — Returns the time, relative to some arbitrary point.

### Input Parameter

**ctx** ADI context returned by `MPID_Init`

### Synopsis

```
double MPID_Wtime( void * ctx )
```

### Notes

This routine may return a different value on eacy processor even when called at the same "time". The only quantity that is meaningful is the difference between two results of `MPID_Wtime`. The resolution of this time is provide by `MPID_Wtick`; while `MPID_Wtime` should provide a high-resolution time, this is not required or guarenteed.

The time measured is elapsed or wall-clock time, not CPU time.

This may also be provided as a macro in the file `dm.h`.

---

**MPID_Wtime_is_global** **MPID_Wtime_is_global**

---

**MPID_Wtime_is_global** — Returns the resolution of the timer `MPID_Wtime`

### Return value

True (1) if the value from `MPID_Wtime` is synchronized on all processes, False (0) otherwise.

### Synopsis

```
int MPID_Wtime_is_global( void )
```

### Notes

This is often implemented with a definition

```
#define MPID_Wtime_is_global() 1
```

If this is not defined, it is assumed that `MPID_Wtime` is not synchronized.

## 2.6 Heterogenity support

The routines in this section provide support for heterogeneous systems. They do not need to be provided on homogeneous systems.

---

**MPID_Dest_byte_order** **MPID_Dest_byte_order**

---

**MPID_Dest_byte_order** — Returns the byte ordering desired by the destination

### Input Parameter

**dest** Absolute rank or destination

### Returns

One of `MPID_H_NONE`, `MPID_H_LSB`, `MPID_H_MSB`, `MPID_H_XDR`

### Synopsis

```
int MPID_Dest_byte_order( int dest )
```

---

**MPID_HAS_HETERO** **MPID_HAS_HETERO**

---

**MPID_HAS_HETERO** — Indicates that heterogeneous processors may be present

### Notes

This is a macro that is defined when the ADI supports heterogeneous processing elements. APIs that are to operate in a heterogeneous environment need to ensure that message are properly prepared. See also the discussion on heterogeneous support, `MPID_IS_HETERO`

---

**MPID_IS_HETERO** **MPID_IS_HETERO**

---

**MPID_IS_HETERO** — Indicates that heterogeneous processors are present

### Notes

This is a global variable that can be used to determine if an ADI that could contain heterogeneous processors actually does. For example, if the ADI can connect disimilar workstations but the active collection contains only similar workstations, then `MPID_HAS_HETERO` would be defined, but `MPID_IS_HETERO` would be false.

See also the discussion on heterogeneous support.

---

**MPID_Procinfo** **MPID_Procinfo**

---

**MPID_Procinfo** — Data structure that contains information on the processes

### Notes

This structure is undergoing development. See `MPID_Dest_byte_order`.

## 2.7 Miscellanenous discussion

This section contains comments on various aspects of implementations of the ADI.
Parameters - Compile-time parameters for the ADI

The ADI makes use of a number of compile-time parameters both to decide what to do and to indicate to the MPICH code what capabilities it has. In addition, specific implementations of the ADI also make use of compile-time parameters. They are all documented here.

### Parameters for the ADI

These parmeters apply to any ADI, independant of implementation.

**MPID_CAN_SEND_CONTIG**
Obsolete; ignored

**MPID_HAS_HETERO**

If set, environment is heterogeneous

**MPID_DEVICE_CODE**
If set, indicates compiling the ADI itself. Should NOT be set when compiling the routines in mpich/src/* .

**MPID_NEEDS_WORLD_SRC_INDICES**
If set, indicates that the Probe routines need "rank in world" instead of "rank in communicator". This is used in mpid/meiko .

**MPID_PACK_IN_ADVANCE**
If set, asks the MPI routines to give the ADI only contiguous data. Currently, all devices set this.

**MPID_RETURN_PACKED**
Receive counterpart of `MPID_PACK_IN_ADVANCE`

### Parameters for the Chameleon version of the ADI

These parameters apply to the version of the ADI in mpid/chameleon, and to the ones derived from it (mpid/ch_*).

**MPID_DEBUG_ALL**
If set, include code (switchable at runtime) for debugging

**MPID_HAS_SHARED_MEM**
If set, environment has shared memory. Intended only for collective routines

**MPID_H_INC**
Used to protect include file if included twice

**MPID_HAS_DEBUG**
If set, ADI has some debugging features

**MPID_LIMITED_BUFFERS**
Indicates an underlying message-passing system with small buffers (like IBM MPL); causes the ADI to be more aggressive about polling for incoming messages

**MPID_TEST_SYNC**
For debugging only (of synchronous messages)

**MPID_TINY_BUFFERS**
If set, underlying message passing system has essentially no buffering (this is more strict that `MPID_LIMITED_BUFFERS`)

**MPID_USE_GET**
If set, use shared memory operations

**MPID_USE_SEND_BLOCK**
If set, use a special send routine that may block when possible. Used in mpid/ch_cmmd to conserve tports.

### Parameters for the channel version of the ADI

The following all refer to the "channel" interface and control details of the message packets that are sent.

**MPID_CACHE_ALIGN**
Used for shared memory; requests that shared memory locks be cached aligned

**MPID_PKT_DEF**
Protects include file

**MPID_PKT_DYNAMIC_RECV,MPID_PKT_DYNAMIC_SEND**
These two indicate whether send/receive routines take address of packet or packet. Used only in the shared memory version

**MPID_PKT_GALLOC**
If set, provides a global allocation of a packet

**MPID_PKT_GET_NEEDS_ACK**
If set, the shared-memory "get" operation requires an acknowledgement (for example, if a single address space is used)

**MPID_ADI_MUST_SENDSELF**
: If set, the ADI handles the case of sending a message from one process to the same process with special code, bypassing the channel interface code.

## Packet Features

These provide optional declarations in a packet for the length of a packet, a link (shared memory), and the source (sending process). Used in shared memory and raw tcp. The "INCLUDE" versions define DECL/SET/GET versions.

**MPID_PKT_INCLUDE_LEN**
: Include a packet size field (`int pkt_len:32`) in the packet definition

**MPID_PKT_INCLUDE_LINK**
: Include a link field (`union _MPID_PKT_T *next`) in the packet definition

**MPID_PKT_INCLUDE_SRC**
: Include a source in `MPI_COMM_WORLD` field (`int src:32`) in the packet definition.

**MPID_PKT_LEN_DECL**
: Appears in the packet definition; contains the actual declaration (if any). For internal use only.

**MPID_PKT_LINK_DECL**
: Appears in the packet definition; contains the actual declaration (if any). For internal use only.

**MPID_PKT_SRC_DECL**
: Appears in the packet definition; contains the actual declaration (if any). For internal use only.

**MPID_PKT_PRIVATE**
: Used to add "private" data at the *top* of the packet. Used in some shared memory versions.

**MPID_PKT_XDR_DECL**
: Used to define a flag for heterogeneous support (message is using XDR)

---

**Chameleon-MPI** **Chameleon-MPI**

---

**Chameleon-MPI** transport (device) implementation

This simple Chameleon implementation is meant to work with most message passing systems and does not rely on large amounts of message buffering. Messages are sent in one or two packets, depending on the length. If the message is short (< `MPID_PKT_DATA_SIZE`), then the header and message are placed into a single message and sent together. Long messages are sent without copying the message by sending the header in a separate packet.

This Chameleon implementation depends on adequate buffering by the message-passing system. Because there is no asynchronous notification of messages in Chameleon (because most transport layers do not provide this important functionality), it is difficult to implement a packet-based system (i.e., one that divides messages into multiple packets and manages all buffering).

## Comments on the implementation

There are many things that can be done to improve the performance of the implementation of the ADI on various platforms. Some of these have been added to the current implementation with ifdef"s (see the MPICH configure for more discussion of what is currently available).

---

**Packet** **Packet**

---

**Packet** implementation - Describes the packets used the the Chameleon and Channel implementations

of the ADI

## Overview

The Chameleon and Channel implementations send message in *packets* containing either header information, data, or both.

The design here is fairly general, allowing both eager and reluctant (rendevous) protocols for each of the MPI modes. Some simplification can be achieved if, for example, the MPI Synchronous mode is only implemented with the reluctant protocol.

This is discussed in more detain in *Creating a new MPICH device using the Channel interface*, and is included here to give a quick summary of the approach.

## Eager

In the eager protocol, the message data is sent without first asking the destination node (for example, to check on buffer space). In order to achieve better use of the underlying message-passing systems, there are two major categories: long and short. In a short message, the message data and message envelope (context, tag, lrank) are sent together. In a long message, the data is sent separately from the envelope (avoiding any data copies). If the MPI mode is Synchronous or Ready, then some additional information is included. In the synchronous case, it is necessary for the receiver to tell the sender when the receive has started (but see Rendevous below); this is done with the `MPID_PKT_SYNC_ACK`. In the ready case, if the matching receive is *not* available, then an error message can be sent back to the sender with `MPID_PKT_READY_ERROR`.

In the long message case, the *sender* may use nonblocking send operation. On some systems, it may be necessary to ask the sender to execute an MPI_Wait-like operation on the nonblocking send in order to get a matching receive to complete. On these systems, the macro `MPID_WSEND_BEFORE_WRECV` is defined, and the MPI_Wait on the send is requested with the `MPID_PKT_COMPLETE_SEND` packet. (The systems that require this tend to use rendevous or reluctant protocols for long messages). *THIS IS UNUSED.*

## Rendevous

In the Rendevous protocol, a *long* send is not begun until the receiver oks it. The request is made with `MPID_PKT_REQUEST_SEND`, and is acknowledged with `MPID_PKT_OK_TO_SEND`. Note that a message-passing implementation may make use of a "ready-receiver" send once the `OK_TO_SEND` has been received. Also note that the rendevous protocol handles standard and synchronous sends identically; no separate `SYNC_ACK` packet is required.

Depending on the system, it may or may not be advantageous to compress the size of the data in the packet. If the macro `MPID_PKT_COMPRESSED` is defined, then the envelope data is encoded into 2 fewer ints.

## Get

The *get* protocol is used on shared memory systems. The behavior of this protocol is very much like the *rendevous* protocol, except that the packets contain information needed to access the data directly from memory shared by the sender and the receiver (thus requiring a different packet type).

---

**Threads** **Threads**

---

**Threads** in the ADI - Support for multithreaded applications

MPI is thread-friendly in that the MPI specification does not specify anything that is not thread safe. However, the MPI specification does not mandate any particular thread support. It is our

intent that the MPICH ADI provide a thread-safe version; this section details the current state of that interface. This is currently untested.

### Communicator-based locks

Some operations on a communicator must be single threaded. These macros can be defined to provide the appropriate locks. The first parameter (`ctx`) is the ADI context (often ignored); the second is the communictor to lock.

**MPID_THREAD_LOCK(ctx,comm)**
: Lock a thread in a communicator

**MPID_THREAD_UNLOCK(ctx,comm)**
: Unlock a thread in a communicator

**MPID_THREAD_LOCK_INIT(ctx,comm)**
: Initialize a thread in a communicator

**MPID_THREAD_LOCK_FINISH(ctx,comm)**
: Free a thread in a communicator

### Datastructure-based locks

These four are for locking individual data-structures. The data-structure should contain something like

```
typedef struct {
   MPID_THREAD_DS_LOCK_DECLARE
   other stuff
   } foo;
```

and then use

```
foo *p;
MPID_THREAD_DS_LOCK(p)
MPID_THREAD_DS_UNLOCK(p)
```

**MPID_THREAD_DS_LOCK_DECLARE**
: The declaration of the lock

**MPID_THREAD_DS_LOCK_INIT(p)**
: Initializes the lock for datastructure `p`

**MPID_THREAD_DS_LOCK(p)**
: Lock datastructure `p`

**MPID_THREAD_DS_UNLOCK(p)**
: Unlock datastructure `p`

Be very careful that thread locks are not nested or that nesting is handled carefully (for example, with a nest level).

---

**MPID_Flags** **MPID_Flags**

---

**MPID_Flags** — Describes the run-time data collection and debugging flags

### Notes

The MPID/ADI implementation collects a number of statistics and provides a variety of debugging options.

This interface is currently under development