PREPRINT MCS-P342-1193

An Abstract Device Definition to Support the Implementation of a High-Level Point-to-Point Message-Passing Interface

by

William Gropp
Ewing Lusk
Mathematics and Computer Science Division
Argonne National Laboratory
gropp@mcs.anl.gov
lusk@mcs.anl.gov

Mathematics and Computer Science Division
Argonne National Laboratory

March 5, 1995

Abstract

In this paper we describe an abstract device interface (ADI) that may be used to efficiently implement message-passing systems. This work was done to provide an implementation of the Message Passing Interface (MPI); however, the interface is appropriate for many message-passing systems. The ADI provides for both simple devices and those capable of significant processing. We discuss some of the issues in the implementation and provide a sample implementation for a "device" that is capable of message-passing.

1 Introduction

Our goal is to define an abstract device (ADI) on which a high-level message-passing application programmer interface (API) such as MPI can be implemented. An important requirement is to support a variety of instantiations of this device, from low-level FIFO's and streams to high-level libraries such as IBM's EUI-H, the Intel NX communication library, or portable libraries like Chameleon, p4, or PVM. Implementations of an API can thus consist almost entirely of portable code; dependencies on the low-level transport layer are encapsulated inside the implementation of the abstract device. What we have most in mind are low-level devices provided by individual MPP and workstation network vendors, and so a primary consideration is that this abstract-device approach not contribute any execution-time overhead to the real device. A wide variety of possible device protocols are envisioned. The design suggested here attempts to retain flexibility by listing a set of macros that are to be defined by each side of the interface but invoked by the other side. In other words, each side provides services to the other.

We do not discuss any of the issues related to providing reliable communications, buffer protocols, or the implementation of the API side of the interface. We assume that the global (or collective) operations are implemented with the point-to-point operations, so our ADI has no global operations. This is an area for further development; we expect to add support for collective operations in the future.

The design of the ADI is made more complex because we wish to allow for but not require a range of possible enhanced functions of the device. For example, the device may implement its own message-queuing and data-transfer functions. In addition, the specific environment in which the device operates can strongly affect the choice of implementation, particularly with regard to how data is transferred to and from the user's memory space. For example, if the device code runs in the user's address space, then it can easily copy data to and from the user's space. If it runs as part of the user's process (for example, as library routines on top of a simple hardware device), then the 'device' and the API can easily communicate, calling each other to perform services. If, on the other hand, the device is operating as a separate process and requires a context-switch to exchange data or requests, then it can be very expensive to switch between processes, and it becomes important to minimize the number of such exchanges by providing all information needed with a single call.

The original motivation for this work was the challenge of providing an implementation of MPI [?] that was both portable and efficient. Although MPI is a relatively large specification, the device-dependant parts are small. By implementing MPI using the ADI, we could provide code that could be shared among many implementations. Efficiency could be obtained by vendor-specific proprietary implementations of the abstract device. For this approach to be successful, the semantics of the ADI must not preclude maximally efficient instantiations using modern message-passing hardware. While this ADI has been designed to provide a portable MPI implementation, there is nothing about this part of the design that is specific to the MPI library; this definition of an abstract device can be used to implement any high-level message-passing library.

To help in understanding our design, it is useful to look at some abstract devices for other operations, for example, for graphical display or for printing. Most graphical displays provide for drawing a single pixel at an arbitrary location; any other graphical function can be built using this single, elegant primitive. However, high performance graphical displays offer a wide variety of additional functions, ranging from block copy and line drawing to 3-d surface shading. One approach for allowing an API (application programmer interface) to access the full power of the most sophisticated graphics devices without sacrificing portability to less capable devices is to define an abstract device with a rich set of functions, and then provide software emulations of any functions not implemented by the graphics device. We will use the same approach in defining our message-passing ADI.

A message passing ADI must provide four sets of functions: specifying a message to be sent or received, moving data between the API and the message-passing hardware, managing queues of pending messages (both sent and received), and basic information about the execution environment (e.g., how many tasks are there). The ADI will provide all of these functions; however, we expect that many message-passing hardware systems will not provide queue management or elaborate data-transfer abilities. These functions will be emulated through the use of auxiliary routines that we will define in this paper.

In Section 2 we discuss the macro prototypes. An implementation based on this interface is available by anonymous ftp from `info.mcs.anl.gov` in '`pub/mpi/mpich.tar.Z`'. At the end of this document is a complete implementation of an abstract device using an existing message-passing system (Chameleon [?]). In Section 4, we give examples of the execution sequence that the API and ADI may go through for a few common message-passing operations. Section 5 we discuss some issues in the implementation of the ADI on systems that support operations such as active messages and message-passing co-processors.

2 The Abstract Device

In this section we discuss the operations that the device may perform. These are in two categories: functions that the device must be able to perform, and functions that are not required but that a system such as MPI can exploit. In addition, there may be several ways in which an operation, such as transferring data, can be performed. A device may specify its preference.

This section describes all of the operations. We first describe the interface for sending and receiving messages because this will help introduce the ADI and motivate the additional functions such as queue management. It loosely conforms to the MPI interface; in particular, the terms blocking and nonblocking have the meaning defined in the MPI standard [**?**].

2.1 Message-passing

A message consists of a message body (the data the user wishes to send or receive), the length of the message body, a message tag (often called type) used to distinguish between messages, a context-id, and a destination (for sending) or source (for receiving). The context-id may be thought of as additional bits in the message tag that are reserved to the API; in MPI the context-id is used to implement communicators that are used in the implementation of safe libraries. Messages can be received in the API only by exactly matching the context-id, and either matching the tag and source, or specifying that any tag and/or source may be matched, in a message received by the ADI. While the context-id may seem unfamiliar, most existing systems provide at least a single-bit context-id that is used to separate user from system messages. For example, a message-passing system may use a reserved bit to distinguish between point-to-point operations made by the user and point-to-point operations used by the system to implement a collective operation.

In addition, a message may be sent in either blocking or nonblocking form. In the blocking form, the ADI will not return control to the API until the message body is available for re-use. In a send, this means that the message buffer has either been delivered or has been transferred into internal memory. In a receive, it means that the message has been received. The nonblocking form allows an API to provide the programmer with the ability to overlap communication with computation. Moreover, The API must ask the ADI about whether the message buffer is available before reusing it.

When sending a message, one of three modes may be used. These are: standard, synchronous, and ready. In the synchronous mode, the ADI must not return control to the API until the destination begins to receive the message. In the ready mode, the ADI requires that a matching receive have already been posted by the ADI at the destination; it is an error (with undefined behavior) if the receive has not been posted. The standard mode has no additional requirements.

four, with buffering, or is the API responsible for buffering?

Table 1: Fields in message handles available to the device

Field	Meaning
`handle_type`	Type of handle (`MPIR_SEND` or `MPIR_RECV`).
`dest`	Destination rank (send)
`source`	Source rank (receive)
`tag`	Tag value
`context_id`	Context id value
`completed`	Flag for whether communication opperation is completed
`mode`	Sending mode (`MPIR_STANDARD`, `MPIR_READY`, `MPIR_SYNCHRONOUS`) (send)
`dev_shandle`	Device's send handle (see Section 2.2)
`dev_rhandle`	Device's receive handle (see Section 2.3)
`datatype`	MPI-style datatype description

It is possible to implement all of these send and receive operations by with an ADI that provides only one send and one receive, e.g., the nonblocking send and nonblocking receive. The synchronous and ready send modes are relatively easily implemented on top of this; a blocking send is just a nonblocking send followed by a call to complete that send. This approach, while simplifying the ADI design, can pay a performance penalty that we wish to avoid. We will point out specific instances of these penalties below. We point out that while the ADI design contains entries for these different operations, the ADI itself can choose an implementation that trades efficiency for simplicity.

In our ADI, a message is specified by a `Request` structure. The relevant elements of this structure are shown in Table 1. This structure is used by the API and will contain additional fields.

Note that there are no fields specifying the location of the data; access to the message body is discussed separately in Section 2.5. In addition, the fields `dev_shandle` and `dev_rhandle` are provided; these provide a place for the ADI to store information about the message in the API's data-structure. These are described separately in Sections 2.2 and 2.3.

The `Request` structure is actually a union; if the type is `MPIR_SEND`, it is a `A_SHANDLE` and if the type is a `MPIR_RECV`, it is a `A_RHANDLE`.

The send and receive operations are fundamentally asymmetric. The asymmetry arises from the fact that sending is always initiated by the API on behalf of the user, whereas the most important aspects of receiving are initiated by the ADI, since data may appear there whether the user is prepared to accept it or not. We attempt to allow the device great latitude in the protocol it uses, including buffering on either the send or receive side, no buffering at all, or a mixed protocol that may depend on the particular message being dealt with. We will see this asymmetry when we discuss the

Table 2: Nonblocking send followed by wait

User Program	API	ADI
MPI_isend		
	A_alloc_send	
	allocate D_send_handle	
	A_post_send calls device layer to start send operation	
		Initiates send operation, possibly calling D_get_totallen and D_get_into_contig to transfer data (Or may just notify destination that message available)
	return	
(User code runs)	...	...
		Interrupt; message sent; calls D_mark_send_completed
	Posts send completed in MPI data structures	
	A_free_send_handle frees device's data structures	
(User code runs)		
MPI_status		
	MPI data structures show send completed	
MPI_wait		
	D_SHANDLE found marked completed	
	free D_SHANDLE	
	return	

message queues in Section 2.6.

2.2 Sending a message

Sending a message from one processor to another is the simplest operation; as we will see, the sending side of this is simpler than the receiving side. Table 2 shows one possible scenario from the point where the user calls a nonblocking send routine (at the top) to when the user completes the nonblocking send.

The API part of this process is responsible for setting up the initial request and for converting user requests into the correct ADI requests. The ADI is responsible for actually transmitting the message. Note also that the ADI and API work together to transfer the actual data; this allows the API to provide a richer set of data layouts (for example, structures or vectors with regular stride) that are not supported by the ADI. Also note that we don't specify the protocol used by the ADI to actually transfer the message; this allows the ADI to optimize for different cases. The rest of this section discusses the way in which a send operation is specified to the ADI.

The API requests the ADI to send a message by forming a `Request` structure containing the information in Table 1. The API must then initialize the ADI's data area in this structure (the `dev_shandle`). This is done with the macro `A_alloc_send_handle( Request *r )`. The API's Request structure must contain a `A_SHANDLE dev_shandle` element. The ADI defines `A_SHANDLE`; typically this is a structure but could be a pointer to a structure or an index into a private memory location.

Next, the macro `A_set_send_is_nonblocking( Request *r, int flag )` is called with `flag = 1` if the send is nonblocking and `flag = 0` otherwise. This should set the appropriate field in the ADI's `dev_shandle`.

Should this be part of the API Request?

We are now ready to post the message, that is, to ask the ADI to send it. This is done with the macro `A_post_send( Request *r )`..

an already complete return flag like post recv?

more on semantics of send?

When the API requires the send to be completed, it calls the macro `A_complete_send( Request *r, Status *s )`. The ADI does not return from this call until the send has "completed" (note that completed in the standard or ready mode means only that the message data buffer is available for reuse). Note that while the ADI is handling a `A_complete_send`, it must be prepared to handle incoming messages unrelated to this request.

Need comments on fairness, requirements that the ADI service other requests

When the API is finished with a send `Request`, it must tell the ADI to free the `dev_shandle` with the macro `A_free_send_handle( Request *r )`.

The API is allowed to use the same `Request` more than once as long as only one requested is posted and not completed at any time.

2.2.1 Rationale for allocation of ADI handles

The ADI may prefer that its private data be handled in a special way. The simplest, assuming that the ADI and API share address space, is for the ADI's information to be incorporated directly into the API `Request` structure. This is shown in Example 1.

Example 1:

```
typedef struct { ... } A_SHANDLE;
#define A_alloc_send_handle(f)
```

Another option is for the ADI to have the device handle allocated dynamically with `malloc` (or some other memory allocator); this is shown in Example 2.

Example 2:

```
  typedef struct { ... } *A_SHANDLE;
  #define A_alloc_send_handle(f)   \
f->dev_shandle=malloc(sizeof(A_SHANDLE)
```

If the ADI wishes to guarantee that the device data is hidden from the API, it can instead give the API an index that the ADI will use to access the data. This is shown in Example 3, where the function `A_PRIVATE_SEND()` (not part of the ADI definition) is used to return an integer index.

Example 3:

```
typedef int A_SHANDLE;
#define A_alloc_send_handle(f)   \
    f->dev_shandle=A_PRIVATE_SEND()
```

The same approach is used for receive handles.

2.3 Receiving a message

Receiving a message is much like sending one (the most important differences are discussed in Section 2.6 on the message queues). The progress of a nonblocking receive is shown in Table 3. A comparison with Table 2 shows that the only significant difference is the "check unexpected queue;" this handles the case of the data having arrived before the user posts the receive for the message.

The API requests the ADI to receive a message by forming a `Request` structure containing the information in Table 1. Note that the modes (e.g., `MPIR_STANDARD`) apply only to send requests; a message can be received regardless of the mode by which it was sent.[1] The API must then initialize the ADI's data area in this structure (the `dev_rhandle`). This is done with the macro `A_alloc_recv_handle( Request *r )`. The API's Request structure must contain a `A_RHANDLE dev_rhandle;` element. The ADI defines `A_RHANDLE`; typically this is a structure but could be a pointer to a structure or an index into a private memory location.

Next, the macro `A_set_recv_is_nonblocking( Request *r, int flag )` is called with `flag = 1` if the receive is nonblocking and `flag = 0` otherwise. This should set the appropriate field in the ADI's `dev_rhandle`.

Should this be part of the API Request?

We are now ready to post the message, that is, to ask the ADI to receive it. This is done with the macro `A_post_recv( Request *r, int *is_complete )`.

If the receive blocking, this does not return to the API until the message has been received. If the receive is nonblocking but the message is already

[1]This choice eliminates some possible optimizations, but it was the choice of MPI and is more general than the choice where the mode of a receive must match the mode of the send.

Table 3: Non-blocking receive

User MPI Program	MPI implementation	Device
`MPI_irecv`		
	check unexpected queue (suppose not found) `A_alloc_recv_handle` set fields, particularly "non-blocking" `A_post_recv` calls device layer to start receive operation	
		posts receive at device level
	return	

(user code)	...	...
		(message arrives, interrupt
		calls device)
		`D_msg_arrived` returns
		status of "posted"
		device transfers data using
		`D_put_from_contig`
		`D_mark_recv_completed`

	mark receive completed in MPI data structures `A_free_recv_handle` return	
`MPI_status`		
	`A_check_device` check MPI data structures for status return	
`MPI_wait`		
	(waiting on a particular receive could transfer control to device layer using `A_complete_recv`) (could poll) return	

available (see the discussion of the unexpected message queue below), the message *may* be received and the flag `is_complete` set to true to indicate that.

When the API requires the receive to be completed, it calls the macro `A_complete_recv( Request *r, Status *status )`. The ADI does not return from this call until the receive has completed and the data is available for the API and the user. The ADI must serve any other requests that arrive while waiting to complete the specified request. The data in `status` contains the tag, source, and length of the message in bytes.

this is a change from the implementation that has no status

When the API is done with a receive `Request`, it must tell the ADI to free the `dev_rhandle` with the macro `A_free_recv_handle( Request *r )`.

The API is allowed to use the same `Request` more than once as long as only one requested is posted and not completed at any time.

2.3.1 Status of posted Receives

still need some code on test and cancel

Cancel (note that cancel must affect the queues as well, and may involve some communication)

Probing for a message is discussed in Section 2.6.

2.4 Send-Receive

We should add this as an option...

Many devices can send and receive data simultaneously, and many algorithms, particularly data-parallel ones, can be arranged to take advantage of this. We allow a send and receive to be specified with macro `A_execute_send_recv( Request *rcv, Request *snd, Status *status )`.

However, we do not require that the ADI support this operation. Rather, if the device can not support this, it does not define `A_execute_send_recv`. The API is then required to submit separate send and receive requests. For example, the code in the API might look like:

```
... code to build send and receive Requests
#ifdef A_execute_send_recv
A_execute_send_recv( snd, rcv, &status );
#else
A_post_recv( rcv );
A_post_send( snd );
A_complete_send( snd );
A_complete_recv( rcv, &status );
#endif
```

```
A_free_send_handle( snd );
A_free_recv_handle( rcv );
```

This (simultaneous send and receive) is our first example of an optional functionality that the ADI can provide. We have made it the responsibility of the API to provide the functionality when the ADI does not in order to keep the ADI simpler and smaller.

2.5 Data transfer

Transfering data from the API through the ADI to another process is a critical part of any device interface design. Unless great care is taken, an interface may require that data be copied several times before being dispatched or received. In addition, the user's data may not occupy contiguous locations in memory; any full-featured API (such as MPI) will provide a way for the user to specify how the data is laid out and the ADI and API together must arrange to move it efficiently.

In describing the data transfer functions, we first describe those that relate to contiguous data. We require only that the ADI handle contiguous data but we make provisions for ADIs that can handle more elaborate data layouts. However, our interface is designed so that data that is noncontiguous in the API can be transferred by providing the ADI with contiguous data. The design is complicated by the fact that we are striving to eliminate unnecessary memory motion; this requires several different ways of moving data between the API and the ADI.

The API is required to decompose any complex data layouts into layouts that the ADI can manage. The ADI may need to make multiple calls to the API to transfer data in this case. The ADI should attempt to minimize the number of transfers, and, where possible, do them directly without using the transfer routines provided by the API.

2.5.1 Transfers from the ADI

When a message is received, the ADI must transfer the data to the API in the location that the API user has specified. There are several ways to do this; the best choice depends on the exact situation. The easiest is for the ADI to simply use the known location of the destination data and for the ADI to transfer the data directly. If, for example, the data is contiguous, then before the receive is posted, the macro `A_set_recv_contig_buffer( Request *r, void *buf, int len )` can be called to set the location into which the data should be stored. Here, `buf` is the user's buffer location and `len` is the length in bytes. The ADI then can use this information to transfer the data into the user's program.

This needs to be ADDED to the implementation (the macro only; the code already does this)

A more general interface that is capable of handling arbitrary data layouts is provided by having the ADI ask the API to perform the actual transfers. This allows the API to provide arbitarily complex datatypes without requiring the ADI to handle them. Four macros implement this interface. The first, `D_put_totallen( Request *r, int len )`, tells the API the total length of the message in bytes (as a contiguous message). `D_put_totallen` must be called before any other macro in this section. Transfers are accomplished with either `D_put_from_contig( Request *r, void *buffer, int len )`, which provides `len` bytes starting at `buffer`, or with `D_put_into_contig( Request *r, (void **)buf, int len, int *actual_len )`, which provides to the ADI a buffer `buf` of length `actual_len`; `len` is the requested length of the buffer. The ADI can use either of these functions as it finds appropriate. The value of `actual_len` may be less then the requested amount; in this case, the ADI must make repeated calls to transfer the entire message to the API (for example, the API may be using a fixed-length intermediate buffer). The API is *not* required to accept the entire message available in one call. Note that this design requires the API to keep track of where the API is in a transfer. The final macro, `D_mark_recv_completed( Request *r )`, is used to indicate that the ADI has completed any data transfers. After this point, the API should free its `Request` structure as well as using `A_free_handle( A_RHANDLE r)` to release the ADI's handle.

For example, if the ADI reads packets of a fixed length, then the code for processing data to the API might look something like

```
D_put_totallen( rcv, pkt.totallen );
while ( data_to_read )
    read packet into mybuf
    D_put_from_contig( rcv, mybuf, pkt.len )

D_mark_recv_completed( rcv );
```

However, say the that the ADI reads a packet of fixed length, and if the message is long, a single additional packet of variable length. Further assume that the API prefers to copy from contiguous data to the final data layout. In this case, the code might look something like

```
D_put_totallen( rcv, pkt.totallen );
D_put_from_contig( rcv, mybuf, pkt.len );
len = pkt.len;
while (pkt.totallen > len ) {
    D_put_into_contig( rcv, &mybuf2,
                          pkt.totallen - pkt.len, &actual_len );
    len += actual_len;
    read rest of message into mybuf2
    }
D_mark_recv_completed( rcv );
```

This interface allows the API to provide any intermediate space for holding messages, and to pick the size of those buffers. Larger buffers will probably provide better performance, but the buffers need not limit the size of message that can be received.

An alternative mechanism for the ADI to provide the data to the API is to have the ADI hand the API a contiguous buffer that the API unpacks as required. This allows the ADI to deliver the message without any further exchanges with the API; this may be important if the ADI runs in a separate process and a context switch is needed every time control is exchanged between the ADI and API. If the ADI prefers this mode of operation, it should define `A_RETURN_PACKED`. In this case, the API should allocate a buffer into which the message data can be placed by the ADI when the message arrives.

2.5.2 Transfers to the ADI

Transfers to the ADI are similar to those from the ADI. Just as for the receive case, the simplest method is for the API to use `A_set_send_contig_buffer( Request *r, void *buf, int len )` to set the location from which data should be read. Here, `buf` is the user's buffer location and `len` is the length in bytes.

This needs to be ADDED to the implementation (the macro only; the code already does this

The general interface uses routines that are the natural counterparts of the receive routines. The macro `D_get_totallen( Request *r, int *len )`, gets the length, in bytes, of the message for the ADI. This must be called before any of the other macros described in this section (it will probably also initialize some buffers). Transfers are accomplished with either `D_get_into_contig( Request *r, void *buffer, int len )`, which tells the API to transfer `len` bytes to the buffer starting at `buffer`, or with `D_get_from_contig( Request *r, (void **)buf, int len, int *actual_len )`, which provides to the ADI a buffer `buf` of length `actual_len`; `len` is the requested length of the buffer. The ADI can use either of these functions as it finds appropriate. These transfer contiguous chunks of memory from the API to the ADI. The value of `actual_len` may be less then the requested amount; in this case, the ADI must make repeated calls to get the entire message. In the case of `D_get_from_contig`, a value for `len` of `-1` requests the API to make as much data available as possible; the actual amount should be returned in `actual_len`. The API is *not* required to make the entire message available in one call even if the value of `len` is `-1`. Note that this design requires the API to keep track of where the API is in a transfer.

The final macro, `D_mark_send_completed( Request *r )`, is used to indicate that the ADI has completed any data transfers. After this point, the API should free its `Request` structure as well as using `A_free_handle( A_SHANDLE s)` to release the ADI's handle.

An alternative mechanism for the API to provide the data to the ADI is to prepack the data into a contiguous buffer. This allows the ADI to send the message without any further exchanges with the API; this may be important if the ADI runs in a separate process and a context switch is needed every time control is exchanged between the ADI and API. If the

ADI prefers this mode of operation, it should define `A_PACK_IN_ADVANCE`.

2.5.3 Noncontiguous data

An ADI that can directly handle more general layouts of data can provide enhanced performance, particularly on high-performance systems where data can be moved between processors at rates similar to the rate that data can be moved to and from local memory. Our ADI design allows an ADI to provide this functionality, and for the API to adapt itself to the ADI. An ADI that can handle more sophisticated datatypes should define the appropriate macros described in Table 4. These let the API know which datatype the ADI can handle directly.

Still to be described: getting information about the datatypes

Optional:

- Copy to/from non-contiguous buffer
  - Vector (strided)
  - Blocked (IOVEC)
  - Hindexed
  - MPI datatypes
- Provide non-contiguous buffer pointer

To handle non-contiguous data, the ADI needs to know the layout of the data and the size of each element; for heterogeneous systems, it must also know the datatype. This data is provided in the `datatype` field of the `Request`.

need more on unpacking the datatype field

MPI_PACK and MPI_UNPACK analogues as D routines. Use of source packing?

If the datatype is too complicated for the ADI (for example, it is a complex structure), the the API can force the ADI to use the `D_get_from_contig` et.al. routines.

2.6 Message queueing

In a message passing system, there are two queues: pending receives and unexpected messages (ones that have been delivered, at least in part, but for which the API has not yet issued a matching receive). Both the API and the ADI interact with these queues. For example, when the API issues a nonblocking receive, this adds an element to the posted receive queue. When the ADI receives a message that matches this posted receive, the

Table 4: Macros for asserting that the device can handle additional datatypes

Macro	Meaning
`A_DEVICE_DOES_STRIDED`	Indicates that the device can handle MPI "vector" type.
`A_DEVICE_DOES_HINDEXED`	Indicates that the device can handle MPI "hindex" type.
`A_DEVICE_DOES_ABSTRACT`	Indicates that the device can handle the abstract data types as described in the MPI subset (no recursive datatype definitions).

ADI must modify that entry in the posted-receive queue to mark that the message has been received. Because the ADI may operate asynchronously (for example, as the result of an interrupt), great care must be taken to ensure that the ADI and API do not attempt inconsistent modifications to the queues. There are a number of solutions to this problem, including the use of critical sections and multiple queues; the solution that we have chosen is to give the ADI sole responsibility for the queues. In other words, when the API needs to investigate any of the queues, it asks the ADI to do so for it. The ADI is then responsible for ensuring that all operations on the queues are safe. Since most basic message-passing devices do not provide any queue management, we provide a suite of routines that can be used to provide the required functions. All that the ADI implementor must do is to ensure that the ADI implement a critical section around queue accesses if the ADI operates asynchronously.

There are really two kinds of queues; one for posted receives and one for unexpected recieves (that is messages that have arrived for which there is no posted receive).

The rest of this section describes routines that the ADI may choose to use in implementing the message queues. Since only the ADI may call these, it need not use these. However, the discussion of these routines is a clear way to describe the sort of message queue services that the ADI needs to support, and to explain why we have the ADI perform all queue services.

Messages are added to the receive queue by the ADI with the routine `MPIR_enqueue( Queue *queue, Request *r )` and removed with `MPIR_dequeue( Queue *queue, Request *r )`.

The unexpected-receive queue is a special case because the ADI adds elements to this queue without notifying the API. Before a receive can be added to the posted receive queue, the ADI must check to see if that receive matches an already received message by using the routine `MPIR_search_unexpected_queue( int source, int tag, int context_id, int *found, int remove, Request **rcv )`.

The macro `D_msg_arrived( int from, int tag, int context_id, Request **rcv, int *is_posted )` is called by the ADI when a message arrives; this searches first the posted receive queue, and, if not found, inserts the message into the unexpected queue. The two operations are done together to ensure that there can be no race condition caused by the API posting the receive after the ADI checks the posted queue but before placing the message into the unexpected queue. A `Request` object is always returned; the value of the flag `is_posted` indicates whether the item was on the posted receive queue of was inserted into the unexpected receive queue.

If the message was not posted, then the API is responsible for allocating a `Request` and returning a pointer to it in `rcv`. When a matching receive is finally posted and the ADI finds it in the unexpected queue (with `MPIR_search_unexpected_queue`, the ADI will need to cause the `Request` to be free. It does this with the macro `D_free_unexpected( Request *rcv )`. The API is encouraged to make this locally executable; that is, this macro should simply set a flag to indicate that this `Request` should be freed later. Of course, the API can define `D_free_unexpected` to immediately free the `Request`.

2.7 Checking the queues

Many message-passing APIs provide a way to see if a message is already available to be received, and to provide some information about that message. This is called *probing* and consists of checking the unexpected receive queue to see if a message with the specified matching criteria is available. A successful probe returns the length of the message, and the values of the tag and source (in case these were unspecified). Since probing is often used to allow a user program to receive a message of unknown length, a successful probe, followed by a receive with the same criteria, must return the same message. This constrains the implementations of the unexpected receive queue.

There are two kinds of probes: blocking and nonblocking. A blocking probe does not return until a message that matches the specified conditions is received; a nonblocking probe returns immediately and indicates whether a matching message is available. A nonblocking probe is made with the macro `A_iprobe( int tag, int source, int context_id, int *found, Status *status )`. The first three arguments have the same meaning as the entries in the `Request` structure shown in Table 1. The `found` argument is set to `1` is a message exists and `0` otherwise. If a message is found, then `status` is set to contain the tag, source, and size in bytes of the message.

A blocking probe is made with the macro `A_probe( int tag, int source, int context_id, Status *status )`. The arguments have the same meaning as for `A_iprobe`. One implementation of `A_probe` is

```
do {
```

```
A_iprobe( tag, source, context_id, &found, &status );
} while (!found);
```

This implementation can be inefficient, since it causes the API to make a large number of calls to the ADI. The ADI implementation of `A_probe` should take advantage of the fact that the API is waiting for a successful probe by itself waiting for a message to arrive, yielding the CPU to other processes until a message arrives.

2.7.1 Rationale for Queue operations

An earlier specification of the ADI allowed both the API and ADI to insert and remove elements from the queues. While this system works well when the API and ADI execute in a single thread (without interrupt handlers), it is hard to avoid race conditions when they operate in separate threads. While the race conditions can be handled with the classical techniques of critical sections, the implementation of these can have a significant impact on the performance of the ADI. We choose to give the ADI sole control of the queues because it needs to be notified when any element is added or removed to the queue and thus little is saved by allowing the API to search the queues directly.

2.8 Execution environment

This section covers all of the odds and ends that are needed to round out the ADI definition. They cover both initializing the ADI and some services that are not strictly message passing but which the ADI may be in the best position to offer.

2.8.1 Controlling the ADI

Before any other ADI calls can be made, the ADI must be first started by using the macro `A_INIT( int *argc, char **argv )`. The arguments `argc` is a *pointer* to the number of command-line arguments and `argv` contains the actual command-line arguments in the usual C language format. The ADI may use some of these arguments to control its operation.

The `A_INIT` routine initializes the ADI. It is not required to start up processes or otherwise load the parallel application itself, though it may do so.

When a program is ready to terminate normally, it should call the macro `A_END()`. The ADI should return from this call; this is simply provided to allow the ADI to release any resources that it may have allocated, and to provide any additional services (such as informing the user of messages sent but not received or received into the unexpected queue but never received by the API). The default behavior of `A_END` should be to produce no output at all.

An abnormal termination is achieved by calling the macro `A_ABORT( code )`. This should terminate the program and all processes associated with it; where possible, it should have the effect of an `abort( code )`.

2.8.2 Information

The macro `A_mysize( int *size )` returns the number of processes in the parallel program. The macro `A_myrank( int *rank )` returns the rank of the calling process; this rank is in the range $0, \ldots, size - 1$.

The rest of the routines in this section are motivated by MPI; any implementation of MPI must provide these features; they are often system-specific, and need to be placed in some system-specific part of the implementation. We have chosen to combine all of these functions into the ADI in order to make the ADI the only code that contains system-specific code to port when moving a message-passing system such as MPI to a new system.

The macro `A_NODE_NAME( char *name, int max_len )` returns the name of the processor in `name`, a buffer of length `max_len`. This name should allow the identification of a particular piece of hardware.

In addition, the ADI should provide two routines to support a local time on each processor. The macro `A_WTIME()` should return, as a double value, the time in seconds since some event. This event is unspecified other than to say that it is fixed during the lifetime of the program. For example, the event can be a calendar time, such as January 1, 1970, or the time when the process started. There is no specified relationship to the values returned by `A_WTIME` on other processors.

The macro `A_WTICK()` returns the resolution of `A_WTIME`, also in seconds.

The macro `A_tag_range( int *high )` returns in `high` the maximum value of tag that the ADI provides. It is expected that most ADI implementations will provide 31 or 32 bits of tag; however, an implementation may provide fewer in exchange for greater efficiency. Note that MPI mandates a tag range of at least $2\hat{~}15 - 1$.

The macro `A_machine_name( char *str, int max_len )` provides the mane of the processor running the calling task. This should identify a particular piece of hardware. The macro `A_MAX_MACHINE_NAME_LEN` gives the maximum number of characters that may be required for `A_machine_name`.

2.8.3 Error handling

need to add here an error-handling interface and some default behaviors

2.9 Miscellaneous

talk about polled versus interupt-driven devices. Note that may want both depending on the environment

If the device can operate concurrently with the user code; in particular, if the device and the user code could access the same data structures asynchronously, then the device *must* assert `A_DEVICE_IS_ASYNCHRONOUS`.

Often, the API may need the ADI to perform an operation before it can continue; for example, completing the receipt of some message. The macro `A_check_device( int blocking )` allows the API to ask the ADI to check to see if the ADI has any work to do. If `blocking` is false, the ADI will return once there are no operations to perform. If `blocking` is true, then the ADI will not return until some event happens (for example, a message arrives). The ADI is free to define what events will cause `A_check_device` to return when called with `blocking` true.

3 Summary

These have yet to be updated

Note how small this set of routines/macros is.

In the file '`Datomic.h`':

```
nothing?
```

In the file '`A.h`':

```
typedef ... A_send_handle
typedef ... A_recv_handle
A_alloc_send_handle(D_send_handle)
A_alloc_recv_handle(D_recv_handle)
A_free_send_handle(D_send_handle)
A_free_recv_handle(D_recv_handle)

A_post_send(D_send_handle)
A_post_recv(D_recv_handle)
A_complete_recv(D_recv_handle)
A_check_device(blocking)
```

In the file '`D.h`':

```
typedef ... D_send_handle
typedef ... D_recv_handle
D_mark_send_completed(D_handle)
```

```
D_message_arrived(src, tag, context_id,
                     D_recv_handle, status)
D_get_contig(D_send_handle, address, maxlen, actual_len)
D_put_contig(D_recv_handle, address, maxlen, actual_len)
D_mark_send_completed(D_send_handle)
D_mark_recv_completed(D_recv_handle)
D_check_mpi
```

4 Example Scenarios

This section contains several examples of sequences of events and calls happening at the user, API, and device layer. It is assumed that the API layer has its own data structures consisting of handles to represent posted send and receive operations, and an "unexpected queue" to hold messages that arrive without posted receives for them. This set of examples is still incomplete, but the other sequences of events should not be difficult to infer from these. Also, these scenarios show only one possible implementation (which assumes that we can interrupt user code when the device wants service). We will use MPI as the API in these examples.

we need more discussion

add an alternate to 3: preposted nonblocking receive with no interrupts of user process

5 Comments on implementation

5.1 Packet oriented devices

The device can copy directly from user space into the message packet's payload area. If the packet is formed by writing to a device FIFO, then the data may be transferred directly to that location.

5.2 Active messages and remote copy

Active messages may be used to communicate tag, source, and context information about a message. Once a message is ready to be receive, it can be moved with remote copy. On machines with hardware support for remote copy, this can allow very fast communication.

5.3 Devices with local queues

A smart device may maintain its own queues of posted sends, receives and unexpected messages. This allows the device to reduce the number of times

Table 5: Blocking receive

User MPI Program	MPI implementation	Device
`MPI_recv`		

	check unexpected queue
	(suppose not found)
	`A_alloc_recv_handle`
	set fields, particularly "blocking"
	`A_post_recv`
	calls device layer to start receive operation	

		posts receive at device level
		(waits, handles other requests)
		(message arrives)
		`D_msg_arrived` returns status of "posted"
		device transfers data using
		`D_put_from_contig`
		`D_mark_recv_completed`

	transfer status info to user status object `A_free_recv_handle` return	

Table 6: Message arrives; receive is posted later

User MPI Program	MPI implementation	Device
		(message arrives)
		`D_msg_arrived` called
	`A_alloc_recv_handle` in unexpected queue	
		`D_msg_arrived` returns status of “not posted” Device may transfer message into unexpected queue at this point, or it may defer data transfer (to buffer on sender, for example). Assume it defers.
	...	...
`MPI_irecv`		
	checks unexpected queue; finds message `A_complete_recv`	
		device fetches messages, transfers into user space with `D_put_contig`
`MPI_status`		`D_put_from_contig`
	return “not completed”	
		`D_mark_recv_completed`
`MPI_status`		
	return “completed”	

Table 7: Chameleon routines used by sample device implementation

Routine	Action
PIbsend	Blocking send
PIbrecv	Blocking receive
PInsend	Nonblocking send
PInrecv	Nonblocking receive
PIwsend	Wait for nonblocking send
PIwrecv	Wait for nonblocking receive
PInprobe	Non-blocking probe by tag
PImytid	Rank of calling task
PInumtids	Number of tasks in parallel program
PIiInit	Start a Chameleon program
PIiFinish	End a Chameleon program

that the device interrupts the user's process when handling messages. Our design allows for this by allowing the API to post a receive to the device. The device may than handle a message that matches a posted receive (provided that it can store the message) without interrupting the user's process. The API discovers that the message has been handled when the user process requests that the message be completed (by calling `A_complete_recv`) or perhaps by looking at the message completed flag (set when the device called `D_mark_recv_completed`).

In order for this to work, the device needs to have access to the location of the destination buffer *when* `A_post_recv` is called.

6 A Portable Implementation

In this section we show a complete implementation of the device code in terms of an existing message-passing system of the abstract device interface. The code is available, with the MPI API, from `info.mcs.anl.gov` in file '`pub/mpi/mpich.tar.Z`'.

This code uses Chameleon [**?**] for the message-passing calls. Chameleon is a portable message-passing system that is allows the use of many popular transport layers, including p4, PVM, Intel NX, and IBM's EUI. Only a small set of calls is used; these are described in Table 7.

It is interesting that we can specify a portable ADI with good performance.

what more do we want to say?

For the sake of concreteness, we propose three files:

Datom.h A set of definitions of the MPI "atomic" datatypes. Many of these may be `enum` types.

A.h A set of macros (both data structures and operations) that the MPI implementation (the API layer) will rely on. Their definitions are to be provided by the device.

D.h A set of macros that the device will invoke to interact with the MPI implementation. Their definitions will be provided by the MPI implementation.

This device implementation maps all messages into messages with tag zero (for the first `A_PACKET_SIZE` bytes) and with tag `source`+1 for any part of a message that is longer.

An alternate implementation would use tags from zero to $2\hat{\ }31 - (1 + p)$ (for p processors) for any message in the initial communicator/context and tag value between 0 and $2\hat{\ }31 - (1 + p)$ and the above tag mapping for all other messages. This gives an implementation where existing message-passing programs would run with little if any overhead, since they would map directly to the underlying message-passing system. We have not implemented this since it discourages the use of contexts and gives an artificial performance advantage to "old-style" message-passing programs. We believe that native implementations to the abstract device interface will suffer only an insignificant performance impact, specifically, the cost of sending the context id and including the context id in the message matching criteria.

References