An Implementation of a Myrinet GM and API for Alpha Digital Unix

Digital™ Unix

Date

David Panariti
R.S. Nikhil

Cambridge Research Laboratory

Technical Report Series

Cambridge Research Laboratory

The Cambridge Research Laboratory was founded in 1987 to advance the state of the art in both core computing and human-computer interaction, and to use the knowledge so gained to support the Company's corporate objectives. We believe this is best accomplished through interconnected pursuits in technology creation, advanced systems engineering, and business development. We are actively investigating scalable computing; mobile computing; vision-based human and scene sensing; speech interaction; computer-animated synthetic persona; intelligent information appliances; and the capture, coding, storage, indexing, retrieval, decoding, and rendering of multimedia data. We recognize and embrace a technology creation model which is characterized by three major phases:

Freedom: The life blood of the Laboratory comes from the observations and imaginations of our research staff. It is here that challenging research problems are uncovered (through discussions with customers, through interactions with others in the Corporation, through other professional interactions, through reading, and the like) or that new ideas are born. For any such problem or idea, this phase culminates in the nucleation of a project team around a well articulated central research question and the outlining of a research plan.

Focus: Once a team is formed, we aggressively pursue the creation of new technology based on the plan. This may involve direct collaboration with other technical professionals inside and outside the Corporation. This phase culminates in the demonstrable creation of new technology which may take any of a number of forms - a journal article, a technical talk, a working prototype, a patent application, or some combination of these. The research team is typically augmented with other resident professionals---engineering and business development---who work as integral members of the core team to prepare preliminary plans for how best to leverage this new knowledge, either through internal transfer of technology or through other means.

Follow-through: We actively pursue taking the best technologies to the marketplace. For those opportunities which are not immediately transferred internally and where the team has identified a significant opportunity, the business development and engineering staff will lead early-stage commercial development, often in conjunction with members of the research staff. While the value to the Corporation of taking these new ideas to the market is clear, it also has a significant positive impact on our future research work by providing the means to understand intimately the problems and opportunities in the market and to more fully exercise our ideas and concepts in real-world settings.

Throughout this process, communicating our understanding is a critical part of what we do, and participating in the larger technical community---through the publication of refereed journal articles and the presentation of our ideas at conferences--is essential. Our technical report series supports and facilitates broad and early dissemination of our work. We welcome your feedback on its effectiveness.

Robert A. Iannucci, Ph.D.
Director

An Implementation of a Myrinet MCP and API for Alpha Digital Unix

David Panariti

28 September 1998

Abstract

Myricom has produced a high-bandwidth, low-latency interconnect for the PCI bus called Myrinet. In this paper we describe an implementation of the control program which runs on the interface card and a host-side API to access it.

CRL Technical reports are available on the CRL's web page at
http://www.crl.research.digital.com.

Digital Equipment Corporation
Cambridge Research Laboratory
One Kendall Square, Building 700
Cambridge, Massachusetts 02139
USA

Table Of Contents

History and Context

CRL has been engaged in systems research including parallel systems and clustering for some time. One result of this has been CLF, the Cluster Language Framework. CLF provides job control, debugging, terminal I/O control, debugger access and a reliable packet protocol (RPP). The RPP provides reliable, in-order delivery of packets using a sliding window scheme. It also uses sequence numbers, acks and provides for retransmission of unacked packets after a suitable timeout interval. The existence of RPP is the reason that the Myrinet software does not proved reliable and in-order delivery. CLF has been designed to allow new transports to be hooked into the system relatively easily. CLF currently supports UDP, Memory Channel, Shared Memory, Myrinet and MPI as transports. MPI can use a number of underlying transports itself, and also provides job control. MPI was implemented to facilitate a port of CLF to Windows NT. Soon to be added is support for Compaq's ServerNet.

CLF was chosen as the logical place to initially add Mryinet support. Along the way to this support, we also implemented a copy-free modification to CLF to allow transports to provide clients with buffers which need not have their data copied before the transport can send it. As an example, the Myrinet adapter can only send data that resides in specially allocated DMA memory. The API allocates a buffer out of this memory so that no extra copy is required. We first prototyped this copy-free mechanism using CLF's shared memory transport. Then we added Myrinet support that also uses the copy-free mechanism. The needs of CLF had a fairly large impact on the design of the Myrinet software.

CLF acts as a substrate for the Stampede system, also developed here at CRL. Stampede is a cluster programming facility that abstracts data communications and provides a memory model called Space Time Memory (STM). Other papers from CRL and Georgia Tech are available that describe Stampede and STM.

Myrinet Overview

Myrinet is a high-speed switched interconnect capable of full duplex operation at 160 MB/s in each direction. Network topologies are flexible: from point-to-point to arbitrarily connected switched configurations. Myrinet packets can essentially be any size (there is a timeout for tail flit that becomes an effective limit). Packets are source routed, with each switch consuming a route byte and recalculating the CRC. As depicted in the figure below, each Myrinet board contains an onboard LANai RISC microprocessor, a block of SRAM that is accessible by both the host and the LANai, three DMA engines and a network interface. It is possible to DMA between the network and the SRAM and to DMA between the SRAM and host memory. In addition, the LANai can perform PIO to and from the network on a byte halfword (16 bits) and word (32 bits) basis. The card is controlled by a so-called Myrinet Control Program (MCP). Myricom provides a very robust and flexible MCP that was, unfortunately, too slow for our needs. However, Myricom has addressed this issue by providing a customized version of the GNU C++ compiler for the LANai processor so that it was possible for us to create our own MCP.

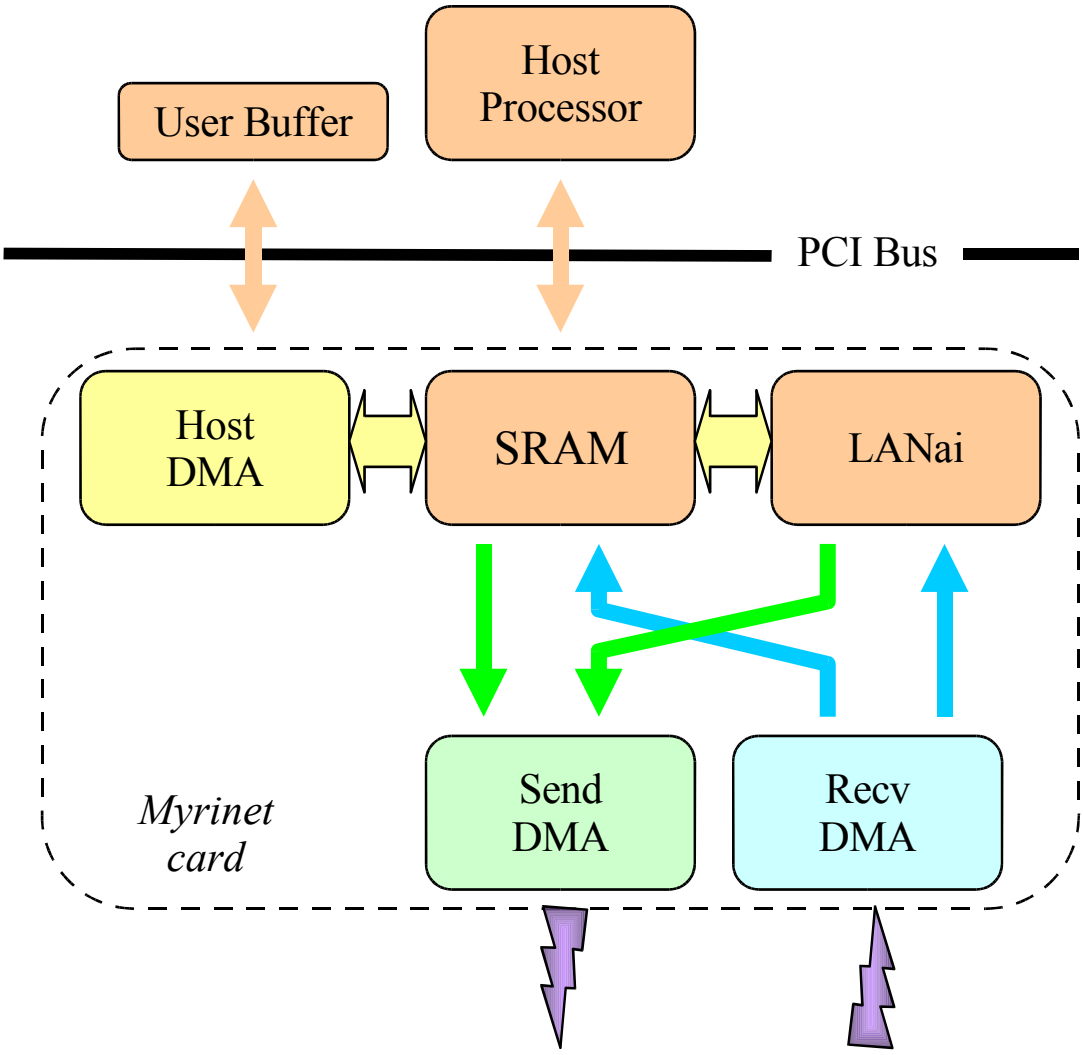


Figure 1, Myrinet Hardware

System Overview

The system consists of three main components:

- The Myrinet Control Program, or MCP – This component resides on the Myrinet card in SRAM and runs on the LANai processor. It controls the card and its basic job is to move packets between the host and the myrinet. It is written in GNU C++ and was written here at CRL.
- The API – this runs on the host platform and controls the MCP by making requests to it through the shared SRAM over the PCI bus. Client applications talk only to the API. This is written in C++ (with C bindings) here at CRL.
- The Driver – this runs on the host platform and mmap()s the SRAM into the API's address space. It also allocates a single large block of DMA memory for packet buffers. This DMA memory is also mapped into the API's address space. This driver allows the API to communicate directly with the MCP without the need to go through the kernel. It is written in C. This was produced by a large cast:
  - Copyright (c) 1997 Duke University
  - Copyright (c) 1996, University of Colorado
  - Copyright (c) 1996, Myricom, Inc.
  - Portions of this Myrinet driver were derived from the BSDI Myrinet driver, written by Ilia Gilderman of The Hebrew University of Jerusalem.

The following figure illustrates the main software components of the system. The MCP is downloaded into the Myrinet SRAM by the API initialization routines. The API is provided as a statically linkable library that is linked with the client program. The driver lives in the kernel, and requires a kernel rebuild in order to function.

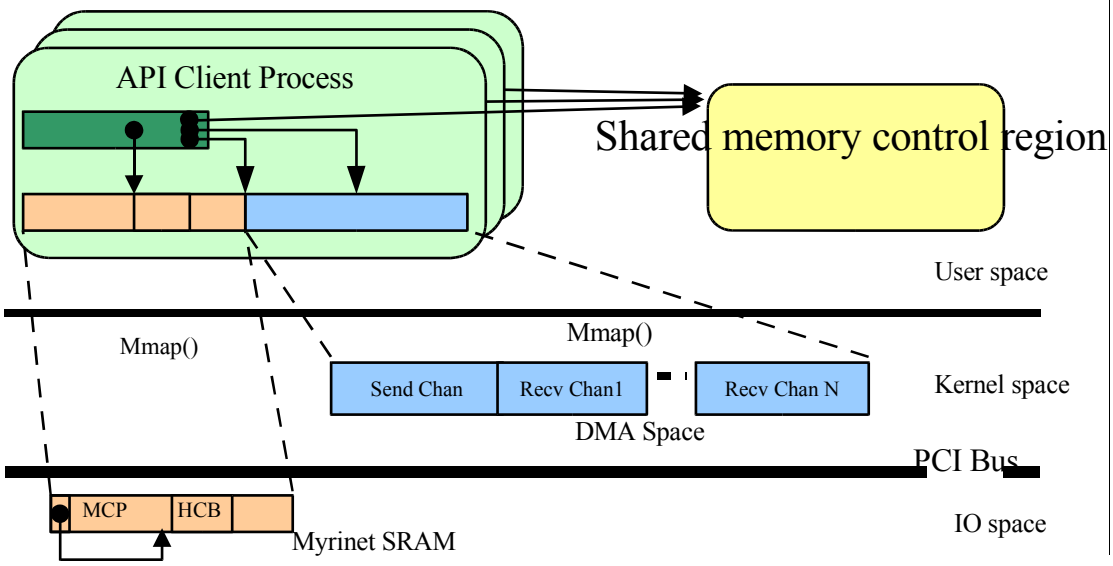


Figure 1, Memory Organization

Protocol

The API and MCP implement no real transmission protocol. It is most like UDP in that packets are not guaranteed to arrive, nor is the ordering of the packets guaranteed to be maintained. It does do its best to provide reliable, in-order delivery, but upper level software is required in order to guarantee this. The Myrinet is a very reliable transport, with a very low error rate. The Myrinet board provides a weak CRC-8 for each packet so that corruption can be detected. Packets with CRC errors are dropped by the MCP and are never seen by a client application. An application may wish to provide additional error checking to ensure that the MCP itself does not harm the data.

Frame Structure

The MCP does impose a frame structure on data packets, adding a header containing packet type and subtype fields, as well as a destination channel number. This subtracts from the total payload available to applications. The route is sent separately and does not subtract from the application payload size. The frame structure is presented below.

Number of bytes:

1	1	2	1	1	n
$Route_0$	$Route_{n-1}$	Type	Subtype	Chan	Application payload

Figure 2, Frame Format

$Route_0$ through $route_{n-1}$ are routing bytes. These are consumed by the myrinet switches along the route to the destination. The *type* field is a two byte field and is assigned by Myricom to allow multiple packet types to exist on a single network. CRL has been assigned the packet type 0x370. Anything after the type is purely implementation dependent. Technically, the type is app dependent, too, but the packet type would not be compatible on a shared network. Currently the *subtype* we use is **`FST_API0_DATA`**, which is defined in **`frame.h`** to be 1. *Chan* is the destination process's channel number.

MCP Overview

The MCP has been designed to provide low latency and high bandwidth. The MCP is loaded into the SRAM on the Myrinet card over the host's PCI bus, which helps to greatly speed up the development cycle. It executes on the LANai processor, a 33MHz pipelined 32-bit microprocessor. In addition to the LANai and SRAM, the Myrinet board holds three DMA engines: one handles DMA between the host memory and SRAM (from-host-DMA & to-host-DMA), one between the network and the SRAM (recv-DMA) and one between SRAM and the network (send-DMA). All three can be running simultaneously, with the host-DMA engine running either to or from host memory. The main function of the MCP is to ensure that all of the DMA engines are running as efficiently as possible in order to move packets through the system.

The LANai is a dual context machine. It has a user context, which is interruptible, and a system context, which is not. Special machine instructions switch between modes. The LANai resets into system context and begins running code in **`crts0.s`**. This code performs some very basic initialization and the jumps to the standard **`main()`** entry point of C and C++ programs. The standard parameters of **`argc`** and **`argv`** are not provided. After initializing the hardware, enabling recv-DMA, unmasking recv-DMA interrupts and handshaking with the host, the MCP issues a "punt" instruction to put it into user context. Interrupts cause a switch back into system context causing the system code to resume executing immediately after the "punt" instruction. After servicing the interrupt, another "punt" instruction allows the user context to run again.

The main polling loops run in user context, servicing the send and host-DMA engines and the interrupt driven recv code runs in system context. The MCP is written in GNU C++ as modified by Myricom. We had to perform some minor modifications to get it to run under 64-bit Digital Unix.

The MCP and the API communicate via the SRAM on the card, which is mapped into the process space by a small device driver. A special area of SRAM is designated as the Host Communication Block (HCB). The HCB contains the send channel, multiple recv channels, route tables, etc. Using shared SRAM for communication allows the API to control the MCP and to send packets without going through the Unix kernel (once the SRAM pages are mapped). This yields much lower latencies, but requires that the API poll the MCP in order to determine if packets are available. Access to the SRAM is across the PCI bus, and is relatively slow, so even though there is direct access to it, it should be accessed as little as possible.

The MCP is structured primarily as three interconnected state machines, one for each DMA engine. The receive DMA engine can be run in two modes: interrupt driven or polled. The interrupt driven mode yields lower latency, but does not allow for overlapped recv-DMA and to-host-DMA, due to synchronization issues. Running in polled mode allows for this overlap and allows for significant improvements in large single packet latencies and corresponding bandwidths, with an increase in latency for small packets. Most benefits of overlapped DMA are lost when moving many packets, due to the pipelining which does a good job of overlapping multiple sends and receives to hide latency.

The MCP was originally designed to be interrupt driven on receive, with one state machine for the host-DMA and one for send-DMA. After initialization, the MCP entered a main polling loop, which called the host-DMA service routine and then the send-DMA service routine. Received packets triggered interrupts, which caused an immediate context switch to the system mode of the LANai processor. The receive ISR checked the incoming packet's CRC, placed it into the rx pool (if room, see below) and reset the receive DMA engine to allow more packets to come in off of the network. It is very important that the network always be enabled in order to prevent network deadlocks[1]. Received packets are immediately attended to, even if that attention is simply to drop them. Receive is then immediately enabled again.

MCP Features

The MCP incorporates a number of features to enhance its latency, bandwidth and usability. Many features such as small packet PIO and overlapped DMA were inspired by the Duke team's Trapeze Project (http://www.cs.duke.edu/ari/trapeze/index.html) which runs on Myrinet on Alpha processors under Digital Unix. Many thanks go out to that team, especially Andrew Gallatin who was of tremendous help in getting things working here.

- Latency and bandwidth enhancing features:
  - Interrupt driven recv
  - PIO send on small packets (insp: Trapeze)
  - Pipelined and buffered (standard, + insp: Trapeze)
  - DMA recv notification
  - PCI crossings minimized
  - DMA buffer allocation – enables copy free operation
  - Overlapped (wormhole/cut-through) send and receive (insp: Trapeze)
- Usability Features
  - Multiple API managed recv channels
  - Simple, coordinated multiprogrammed initialization
  - Route table management
  - Route file support

API Overview

The API runs on the host machine and its job is to allow client applications to access the Myrinet card in a simple and efficient manner. The API communicates with the MCP via the Host Control Block (HCB) which is described in more detail below. The API allows the client to allocate send buffers, to send such buffers, to

poll for send completion, to allocate, initialize, poll and receive from recv channels. It also provides functions to set and maintain source routes used by the myrinet.
The API manages and coordinates access to the MCP via the HCB. In addition, the API maintains a shared memory region to hold information needed by multiple clients but not by the MCP. Since the SRAM is a slow and limited resource, these common variables, data structures, locks, etc have been removed from the HCB.
An important feature of the API is the management of send buffers. Since the API provides clients with actual send buffers, extra care must be taken when managing these buffers. For example, it is essential that buffers are never modified until they have been sent to their destination. It is also essential that a freed buffer is not reallocated to a client until it has been sent. These issues are described in more detail under the **Send Channel** section.

API Programming Model

The basic programming model for the API is as follows.

1. System initialization
   1.1. Init/load MCP
   1.2. Register routes
   1.3. Allocate recv channel (receiver only)
   1.4. Init recv channel (receiver only)
2. Sender:
   2.1. Allocate send buffer
   2.2. Fill send buffer
   2.3. Send buffer
   2.4. Free send buffer
3. Receiver
   3.1. Receive buffer from channel
   3.2. Use data
   3.3. Free receive buffer

Host Communications Block

The API and the MCP communicate via a structure in SRAM called the host communications block or HCB. The HCB contains one send channel, multiple recv channels, the MCP's state, the PIO buffers and the route table. The Myrinet card is a shared resource, which implies that the SRAM, MCP, HCB, etc. are all shared, too. It is possible, for example, to have the API load a new MCP into SRAM during initialization. This would be catastrophic if other processes were already using the card. Therefore, the API uses a system of semaphores to identify first access to the card. Only the first process can load the MCP or do other critical initialization tasks. The semaphores are set up with undo operations at process termination so that no lingering references to the card remain. Once all processes using the card have exited, the next process to access it will be considered the "first" and can initialize the card.

The HCB can reside anywhere in SRAM, but its address is stored in a fixed location in SRAM. The MCP's crt0.s code contains the following fragment:

(Not

This
locat
alwa
mair
SRA
the a
The

```
.text

.global start
.global _hcb_ptr
.global _init_state

start:
!                                       Addr in SRAM
      mov     0x40000, %sp      !           0x0
      bt      around            !           0x4
      nop                       ! shadow    0x8
                                6
_hcb_ptr:
      .long 0                   !           0xC

      .
```

```
        mov     %sp,%fp             ! frame pointer
        add     %pc,8,%rca
        bt      _main               !call _main
        st      %rca,[--%sp]        !push return address during shadow
```

Figure 2, crts0.s

multi-byte data items. Also, it is essential to maintain alignment in the HCB. The HCB class is used directly by both the MCP and API and each has differing alignment requirements. Conditional code is included to ensure correct alignment.

A slight wrinkle is that, since the asynchronously running MCP stores the HCB pointer, we need to ensure that the MCP has set up this pointer before we allow the API to access any data within the MCP. We simply loop until the pointer becomes non-null, since it is initialized to 0. Once the pointer is obtained, we perform a handshake operation with the MCP via the MCP state variable in the HCB. After the handshake completes, we know that the API and the HCB are in known states. We then perform a simple sanity check. The HCB contains a few signature ints within it, at the beginning, end and somewhere near the middle. These are initialized by the HCB's constructor and are checked by the API after handshaking. This helps to prevent problems caused if the size or locations of HCB elements change. This is primarily useful when doing MCP development, since a mismatched API and MCP can cause mysterious failures. Its not bullet proof, but it sure helps.

Structures in the HCB are sized to multiples of two where feasible, or the sum of two powers of two. This makes accessing arrays much faster since the LANai has no hardware multiply. The PIO buffers and queue items must be longword aligned since we use an optimized long copy function in the API.

The following figure shows the basic structure of the HCB and the arrows indicate the direction of data flow into and out of the HCB as driven by the two processors.

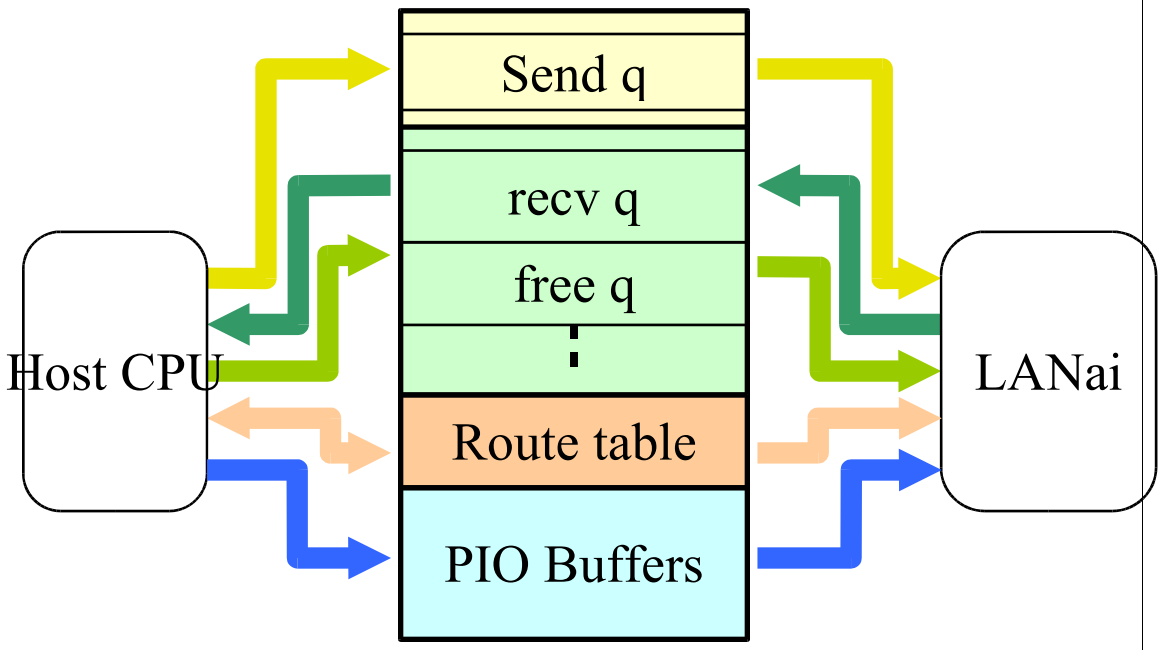


Figure 2, HCB

Buffers

The API and MCP allow for the development of copy-free clients by allowing packets to be allocated directly out of DMA mapped memory. The host-DMA engine can only DMA to and from this kind of memory and allowing API users to access the memory directly can eliminate a copy operation. Received packets are simply pointers into the DMA buffer space, so again, there is no copy operation required.

The device driver which maps SRAM into user space also allocates a region of DMA enabled memory and maps it at the end of the SRAM. This memory is used for packet buffers. It is divided up into one array of

send buffers, *NUM_CHANS* arrays of recv buffers and *NUM_CHANS* recv poll structures. From the API's perspective, it looks like the DMA space follows immediately after the SRAM, but this is just the way the driver happens to map it. Just the buffers and poll structures live in the DMA memory. The channels and their corresponding queues live in the HCB in SRAM (see HCB, above). The DMA region is a limited resource that is allocated when the device driver initializes and cannot be increased while the OS is running. The size of the DMA space can be increased by changing a constant in the driver and recompiling the driver, the Myricom libraries and the API. A possible future enhancement is to allow the API to request the driver to allocate addition buffer regions and map them into user space. The library would need to be modified to use variables provided by the driver rather than using compile time constants. Also, no clients could be active at the time the buffer space is being increased due to the rather simplistic division of DMA space into send, recv and poll regions. Non-contiguous DMA regions would be no real problem since the drive could map them contiguously for the processes. No single buffer could span DMA regions, however.

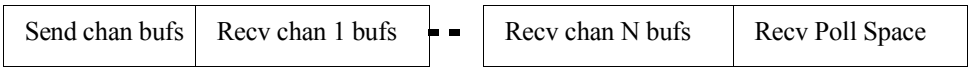


Figure 3, DMA Memory Usage

Send packets are allocated directly out of the DMA memory. Once a packet has been given to the API, it cannot be modified until the MCP has sent it. The API provides a function to poll a send packet, or you can just free the packet to the API, which will hold on to the packet until it has been sent.

The API can only send packets from the DMA region. It is best, therefore to allocate and send packets quickly without holding on to them. Received packets returned by the API are simply pointers into the DMA space**.** Due to their nature, it is also best to free recv packets as quickly as possible, since they are needed to receive new packets into. This DMA memory incurs no additional access time penalties, unlike SRAM. Received packets can be turned around and sent as send packets simply by passing the recv buffer to the send routine. The same restrictions hold as for regular send packets. Allowing direct access to these buffers removes the need for extra copies, but does add some usage restrictions. Applications that wish to provide the illusion of infinite or at least a very large number of packets will have to provide their own buffer management, and it may introduce extra copies. For example, CLF has a copy-free mechanism which allows a packet to be allocated directly from a transport's buffer memory. CLF can therefore allocate a packet from the API and pass it on to the client. When CLF see this kind of packet is being sent, it simply passes the buffer on to the API. However, it may be that CLF cannot allocate a buffer from the API. In this case, it malloc()s a buffer and hands that to the client. Later, when it is time to send that packet, CLF will again attempt to allocate a DMA buffer from the API. If it cannot get one, it will try to steal one from its send queue. If there are none there (i.e. an app is holding onto too many packets), it will give up for the time being and try to send again later. Eventually CLF will get a packet from the API and then it will copy data from the malloc()ed buffer into the DMA buffer and send the new buffer to the API for sending.

Channels

The HCB contains one send and multiple recv channels. The channels contain queues for elements that describe I/O buffers and some handshaking information for initialization. The queue elements contain pointers to the actual buffers in DMA space and to the length of the data in the buffer. Each buffer is sized to some maximum value defined in **`myri_api0_k.h`**. This value should be adjusted so that client payloads can fit within the payload space in the myrinet packet or else the client application must perform fragmentation and reassembly on the payload.

Send Channel

The HCB contains one shared send channel that is used by all of the API's clients. A single send channel was chosen to help reduce MCP latency when determining if new packets are present to be sent. If there are multiple send channels, then the MCP must search through all of them to determine which channel has an

item to send. Unused channels could therefore slow down the ones that are in use. The API provides any necessary locking of the send channel to ensure consistency. For applications that are guaranteed to use only one sender or apps that have already been serialized before sending, there are special non-locking functions that can save a few cycles.

The send channel is a simple dual pointer shared queue, with the API enqueuing items and the MCP dequeuing them. Associated with the send channel and located in the HCB is an array of PIO buffers (q.v.) used for small packets. There is a PIO a buffer associated with each regular send buffer and it is essentially used in its place once a PIO-able packet is sent to the API.

Since the send channel lives in slow SRAM on the far side of the PCI bus, the queue pointers are shadowed in main memory. This opens the API up to seeing a stale head pointer, which is updated by the MCP when it dequeues an item. However, this only makes us think the queue is full when it isn't, which is harmless, except that we'd never send anything more. Therefore, when we see a full queue we simply access (and cache) the current queue head. This is an SRAM access, but it only occurs once every N-1 sends, where N is the size of the send queue.

The initial design of the send channel had the MCP returning buffers to a free list after they had been sent by the MCP, which was quite convenient. We knew exactly when a packet was sent, and hence when we could reuse it. However, the primary initial client for the API, CLF, uses a sliding window for sending packets and assumed that it maintained ownership of a buffer until it explicitly freed it. CLF could not tolerate having buffers that it thinks it owns being freed by another party. CLF is aware of the copy-free potential of the API, and will allocate buffers from the API and pass them on to its clients. These are the buffers that are placed into CLF's RPP send window. If CLF detects a timeout or a lost packet, it will attempt to resend the buffers in its send window. If these buffers have been reused, then random data will be sent in place of the original packet contents. Therefore it is a requirement that all API clients free their send packets rather than assuming that they will be freed after they are sent to the API. The API provides a function for freeing packets that is smart enough to take any send packet and deal with it. If the MCP has not sent the packet yet (i.e. it is still queued, being DMAed from the host or being DMAed to the network), then the API will enqueue the buffer on a busy list and not reallocate it until it has been sent. If the packet is already sent, then it is added to the send channel's free list. To determine when packets are actually sent, the API places send packets onto a FIFO list when they are sent to the MCP. In addition, the API keeps a count of the total number of packets it has enqueued. The MCP keeps a count of the number of packets that it has sent in the HCB. The API can use its enqueue count and the MCP's send count to determine how many packets have actually been sent. We preserve the ordering of packets once they are enqueued in the send channel so we can assume the first *n* (n = num enqueued – num sent) packets in the list are the ones that have been sent. These packets are available for reallocation. The API also provides functions to poll a packet's completion status and to get the number of pending I/Os. This last is just an approximation, since the number is changed asynchronously by the MCP, but it will be in error by showing too many pending, which is a "safe error," as opposed to too many sent, which can cause an unsent buffer to be reused too soon.

Recv Channel

The HCB also contains multiple recv channels to support multiprogramming. The channel is intended to provide similar functionality to the TCP or UDP port number: i.e. it identifies the process within the machine. There is no API support for multiple processes to share a single recv channel. The API performs no locking on recv channels before accessing the data. If a client wishes to allow multiple threads or processes to access a recv channel, then the client must perform locking on its own. API clients send packets to a particular recv channel by passing the channel number as a parameter to the send function. The MCP places incoming packets into the appropriate recv channel. Receiving clients poll a particular recv channel to see if there are any packets available.
The number of recv channels is a compile time constant, **`MYRI_API0_NUM_CHANNELS`** that lives in **`myri_api0_k.h`**. Changing this value requires a rebuild of the API and any applications that use the constant.
Since it is critical that no two processes use the same recv channel, the API has routines that allow multiple processes to request and get unique channel numbers. The API opens a socket with a port number that is a function of the channel number (e.g. **K** + chan_num) for each channel, providing a way of detecting used

channels. The socket is not used for any kind of data communication, we just use it to provide a convenient means of telling us when a channel is in use. The Unix kernel will not allow us to open a socket on a port that is already opened. The Unix kernel will also close a socket when a process exits, even if it terminates abnormally, thus releasing the channel. Sockets were chosen since, in this application, they were much easier to use than Unix's rather cumbersome semaphores. The base port is **`BASE_CHAN_PORT`** in **`chan_mgr.h`.**

Each recv channel contains two queues: one for received items and one for free items. Each is a simple dual pointer queue. When a recv channel is initialized, the received item queue is set to be empty, and the free item queue is is filled with I/O elements that are initialized to point at DMA buffers (q.v.) previously assigned to this recv channel. Once the channel is initialized it can begin to receive packets. Packets sent to an uninitialized channel are dropped by the MCP. The MCP dequeues buffers from the free queue and enqueues them onto the receive queue. The API dequeues packets from the receive queue and enqueues them onto the free queue when it is done with them.

Routes

The route table lives in the HCB, too. Myrinet uses source routing, so the complete route to the destination must precede each packet. Routes are registered with the API, which returns a unique route identifier. The route is stored in the HCB's route table (in SRAM). When packets are sent, they use just the route id. The MCP uses the route id to locate the actual route in the route table. This helps to reduce, by a small amount, the traffic over the PCI bus to the SRAM. Clients can request that a route be assigned to a specific route id. However, if this route is in use it will be changed from this point on. Packets being sent using that route id will now go to the new destination. This is probably not a good idea. The best thing to do is to ask the API to assign a route id to the route you provide. This route, however, may already exist in the system. For example, if a process is sending to some node on one channel, and then a new process wants to send to the same node (on a new channel), then the new route will exist in the route table already. When the new route is registered, the API will detect the route already in existence and return that route id. This is done to conserve space in the route table that must exist in SRAM.

Myrinet routes can be any length, but the API limits them to be **`MYRI_API0_MAX_ROUTE_LEN`**, which lives in **`myri_api0_k.h`**. The number of routes the API can remember is set as a **`const int NUM_ROUTES`** in **`class route_table_t`** in **`route_tab.h`**. The API provides some simple route table management using a simple text file. Nodes can be referred to by a symbolic name. A simple API call is available that, given a source and destination node name, returns the route from the source to the destination. The source routes are fully specified in the route file. This file is maintained by the user. There is currently no means of automatic route discovery. This was purposely left out of the MCP for speed considerations. However, it should be possible to perform route discovery on top of the API as a client application that maps the network and writes a route table.

```
# Current switch setup:
#
# port  host
#-----------
# 0:    east
# 1:    scout
# 2:    pong
# 3:    valencia
# 4:    ping
# 5:    rowdy
# 6:    wishbone
# 7:    gil
#
# src_comm_host          dst_comm_host        unit_num route_bytes...
#
east                     gil                     0    7
east                     ping                    0    4
east                     pong                    0    2
east                     rowdy                   0    5
east                     scout                   0    1
east                     valencia                0    3
east                     wishbone                0    6
#
gil                      east                    0    -7
gil                      ping                    0    -3
gil                      pong                    0    -5
gil                      rowdy                   0    -2
gil                      scout                   0    -6
```

Figure 3, Route File Fragment

Data Flow

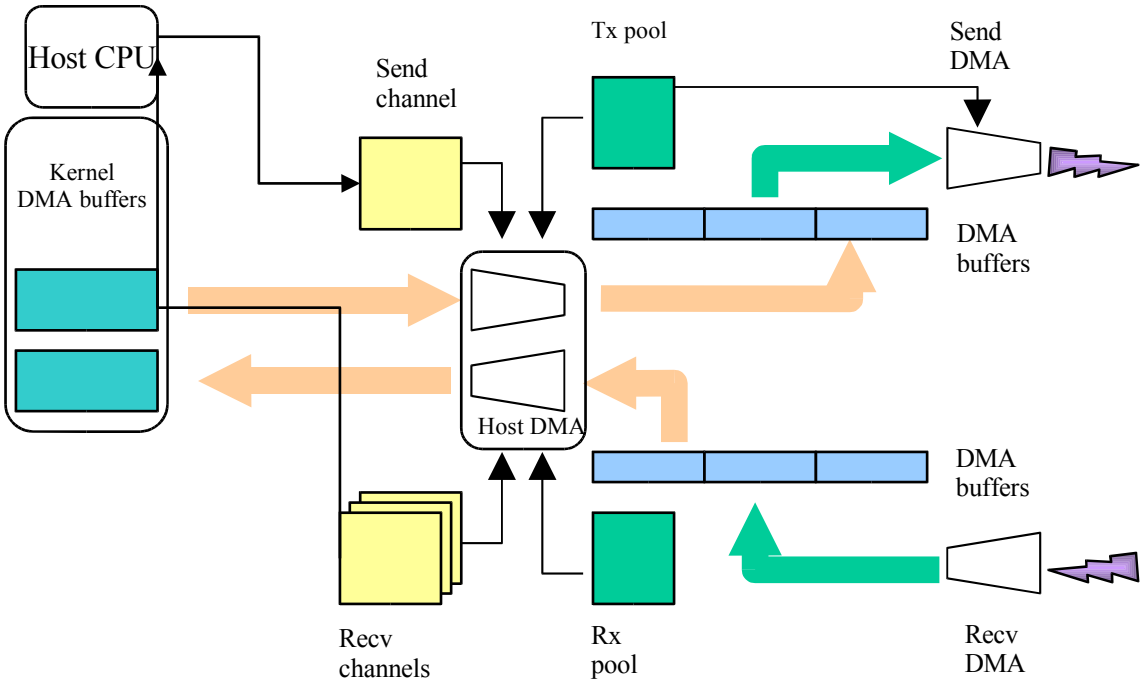


Figure 3, Data Flow

The three DMA engines are driven by three state machines, each implemented by a different class. The host-DMA and send-DMA engines are always polled. The recv-DMA can be either polled or interrupt driven. When interrupt driven, only a small amount of the state machine is needed. A slight bottleneck in the system is the host-DMA engine. It can be running either to or from the host memory, but not both. This means that the host-DMA engine is active twice as often as either the send or recv DMA engines. When sending a packet, the host-DMA engine moves data from the host into the tx pool. The send-DMA engine moves packets from the tx pool to the network. This is illustrated in the top half of figure 3. Data from the network are moved by the recv DMA engine into the rx pool. The host-DMA engine then moves data from the rx pool to the host's memory. This is illustrated in the bottom half of figure 3.

To increase bandwidth, the MCP works very hard to pipeline DMA operations and to keep all DMA engines busy. The tx pool serves as a buffer between the from-host-DMA operations and send-DMA operations, allowing us to start a new from-host-DMA operation even if the send-DMA is busy. Since the send-DMA is much faster than the host-DMA, this tends to happen only if a packet is blocked in the network by another packet. Back ups can also occur when, for example, the send-DMA is sending a very large packet and the host-DMA is moving some small packets. The tx pool also allows the send-DMA to remain busy if there are pooled sends available even if the host-DMA engine is busy. The rx pool allows packets to be received by the MCP when the host-DMA engine is busy. Since the host-DMA is slower than network DMA, this can happen quite often. Also, since there is only one host-DMA engine, it can be busy on sends as well as receives and so recv buffering is even more important. The rx and tx pool classes are template instantiations of a generic io pool class. The template parameter is the number of buffers in the pool. This allows us to easily set the size of the rx pool to be much larger than the tx pool. The tx pool only needs to be large enough to keep the send-DMA engine busy, but the larger the rx pool, the less likely we are to drop packets.

The host-DMA class is responsible for selecting the next operation for the host-DMA engine to perform. Originally the algorithm was, in order of importance:

1. Select an rx pool packet if the rx pool was over some high-water mark,
2. Service any pending sends from the host if the tx pool was not full,
3. Select any pending rx pool packet.

Now, we simply service any pending recv, then any pending host send if the tx pool is not empty. This provides the lowest latency for small, single packets and loses very little in other cases. Once the next DMA operation is selected, the host state machine shifts to the state that handles that operation. Once the DMA to or from the host is started, we loop polling for the DMA's completion, servicing the other DMA engines inside the polling loop. When DMA completes, we finish the transaction and then shift back to the IDLE state.

Sending

Sending data has a few variants:

- We can do a PIO send whereby we do not DMA from the host, but allow the host CPU to copy data into the SRAM directly.
- We can DMA data into the tx pool.
  - Optionally, we can overlap this DMA with send-DMA to the Myrinet.

In all cases, sending starts with the API putting a packet into the send channel. In the first case, when the API sees a small enough packet, it simply copies the packet into the corresponding PIO buffer inside the HCB. This avoids the delay and overhead of starting up the host-DMA engine. It does add some load to the host machine, but is a win overall. The PIO threshold is determined by a compile time constant. When the MCP dequeues and detects a PIO packet, it sets a pointer to the PIO buffer and shifts to the state following DMA from the host. Since there is no DMA from the host, no DMA overlap is possible with PIO buffers. PIO and DMA packets can be freely intermixed.

In the case of larger packets, DMA is the better alternative. The MCP, after having decided to perform a from-host-DMA operation, dequeues a send channel request and begins a DMA operation from the host. During the DMA it loops waiting for the engine to finish the transfer. While looping, it services the other DMA engines. This arrangement made it quite easy to add overlapped send-DMA. If the tx pool queue is empty when the from-host-DMA operation begins, then we are in a position to perform overlapped DMA. It is not enough that the send-DMA engine is idle, since the tx pool may have other packets queued in it and we wish to preserve the order of packets as given by the client. We set a flag indicating that overlapped DMA is possible, and pass in that flag when servicing the send-DMA engine. When the send-DMA engine sees this flag, it enters the overlapped DMA state. It just basically checks the live host-DMA pointer and DMAs a chunk to the network when it is available. The size of this chunk is a variable. It then moves an internal pointer to the end of the just sent chunk. In this way, the send-DMA follows the host-DMA in a chunk-wise fashion. The MCP will send all of the data that are available, as long as it above the threshold amount. The only exception is the last chunk of the packet which will be sent regardless of its size.

The following figure shows the overlapped DMA process. The host DMA pointer shows the next position to be filled with data from the host-DMA engine. The threshold is the minimum amount of data we wish to send to the network. In the figure, we show the DMA process sampled at three places during the transfer. Each time, we send the amount available as shown in the "DMA to network" blocks. The final sample shows that we send the remainder of the buffer even if the amount is less than the threshold amount.

If we are not doing overlapped send-DMA, then when the from-host-DMA completes we simply update the tx pool's incoming pointer to enqueue the packet. When something calls the send-DMA service routine and the send-DMA is idle, then it will notice the packet in the tx pool. The send-DMA code will then begin a send-DMA operation. First, the route must be sent. The API only tells us the route ID, so we have to look up the route in the route table. This is a simple indexing operation. We have a special case of route 0 meaning no route at all, i.e. a point-to-point link. The route can be any number of bytes, but the LANai send-DMA engine

can only send full 32 bit words to the network. Early versions of the MCP simply used PIO to send the route bytes to the network. The LANai can PIO 4, 2 or 1 byte quantities to the network. The route sending routine was optimized to send as many 4 byte, then 2 byte then 1 byte pieces of the route as possible. Recently, however, we began to use a special feature of the LANai send-DMA engine. There is a register, called the send alignm ing of a DMA transfer. After skipping this many bytes, it DMAs the rest as usual. This just requires us to determine how many bytes need to be s et the register. The API makes sure to offset the route into the first 32bit word so that the correct bytes are skipped. This is the number of bytes that the send DMA engine will not send.

The figure below illustrates 4 routes of lengths 1, 2, 3 and 5. x's denote bytes which will be skipped as the DMA engine sends the route to the network. The DMA source address is set up to point to the leftmost byte (4 byte wor

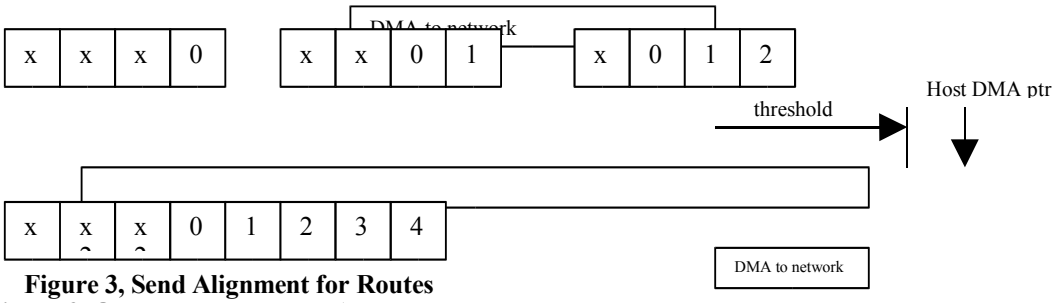


Figure 3, Send Alignment for Routes
Figure 3, Overlapped send-DMA

The MCP sets the SA register, starts the DMA engine and returns. The calling code can then perform actions while the DMA engine is sending the route. The DMA engine can send route bytes faster than the LANai can, and does so without using the LANai CPU. We have the opportunity to do some useful work in the "shadow" of the DMA operation. When the MCP is ready to send the rest of the packet it polls the DMA engine to ensure the route is sent and then it sends the rest of the packet. In the case of overlapped DMA, when the first chunk is ready, the route is sent in an identical fashion before the actual data in the first chunk.

Receiving

There are two recv modes: interrupt driven and polled. Here, we refer to interrupts on the LANai processor. The host processor is never interrupted when using the MCP and API. Interrupt driven is the original mode and was implemented in order to lower latency. By interrupting directly to the recv complete state, overhead is eliminated and there is no latency in waiting for the other polling operations to complete. This is verified by the fact that changing to polled DMA adds approximately 5μS to latency. Polled DMA, however, is required in order to allow overlapped recv and to-host-DMA operation. Polling prevents races between the "foreground" task of to-host-DMA and the "background" completion of a recv. Since we are using the live DMA counters and the live rx pool it is best if they are not changed in a manner unknown to us.

Interrupt driven recv is pretty straightforward. The LANai is configured to be interrupted when a packet is received, or when a packet overrun occurs. The recv-DMA engine is pointed to a recv buffer from the rx pool and set up to receive a maximum of *bufferlen* bytes. The network hardware can detect the last byte of a packet and posts an interrupt when it is detected, or when it is not detected before *bufferlen* bytes are received. This causes the system context to resume executing after the "punt" instruction. First, we see what kind of interrupt it was. If it is not a normal recv-DMA complete interrupt, we drop the packet. Packets with non-zero CRCs are also dropped. The rx pool is a dual pointer queue, with the recv ISR using and writing to the incoming pointer and the main polling routine using and writing to the outgoing pointer. Both sides read both pointers to determine fullness and emptiness. Due to the nature of the dual pointer queue, we seem to lose one buffer. However, this is buffer still useful, since we need a buffer to recv into at all times in order to keep the network alive[1]. The recv-DMA engine is always setup to point into some rx pool buffer. It is the nature of the dual pointer queue that you can use the last buffer, you just cannot tell anyone about it since it is impossible to tell the difference between full (head == tail) and empty (head == tail). So we use (tail + 1 mod QSIZE == head) and lose a buffer. In this case, we really don't lose anything since we'd need some space

somewhere to recv into to keep the network moving packets. So after a packet is received, if the rx pool is full, we effectively drop the packet by resetting the recv-DMA to reuse this last "unusable" buffer. If it is not full, then we move the outgoing pointer to the next buffer and set up the recv-DMA using that value. We then switch back to the user context. The user-context polling loop will eventually detect that there is an incoming packet in the rx pool (head != tail) , and decide to DMA it to the host. In preparation for this, we verify the packet header for type and subtype and then do a bounds check on the recv channel number which is stored in the packet header. If all is well, we use the channel number to select the correct recv channel in the HCB (q.v.). After finding the recv channel, we check for space in it. If there is none, we abort the DMA operation, but do not drop the packet. It may be that later there will be room in the recv channel after the API client has consumed some packets. Additional incoming packets will be queued in the rx pool until it is full, and then dropped there. This way, we do not drop packets until absolutely necessary. Once there is room in the recv channel, the buffer at the outgoing pointer in the rx pool is used as the source of a to-host-DMA operation. We then loop polling for DMA completion, while servicing the other DMA engines. Upon completion of a to-host operation, we update the outgoing pointer and shift to a state that DMAs the current count of DMAs completed to the host. The host polls this number to see if any new packets are available.
We use DMA notification because on certain classes of machines, it was seen that the host's rapid polling of SRAM waiting for new packets was interfering with DMA from the host to the SRAM. To deal with this, the MCP stores the current number of I/Os completed per channel and sends this number to the API via DMA. The API can then poll this number and process I/Os whenever it increases. This adds a little to the latency of each packet, but on certain machines can increase the DMA rate by 50%. Also, since we DMA the total number of received packets, it may be that multiple packets are available when the poll routine is called. We will not have to poll again until we've processed all of the received packets.

Overlapped recv-DMA is just like overlapped send-DMA, except we need to follow the recv-DMA engine and send chunks through the to-host-DMA engine. Another small difference is that we must make sure that the threshold size is larger than the frame header, since we need information from it in order to select the recv channel.

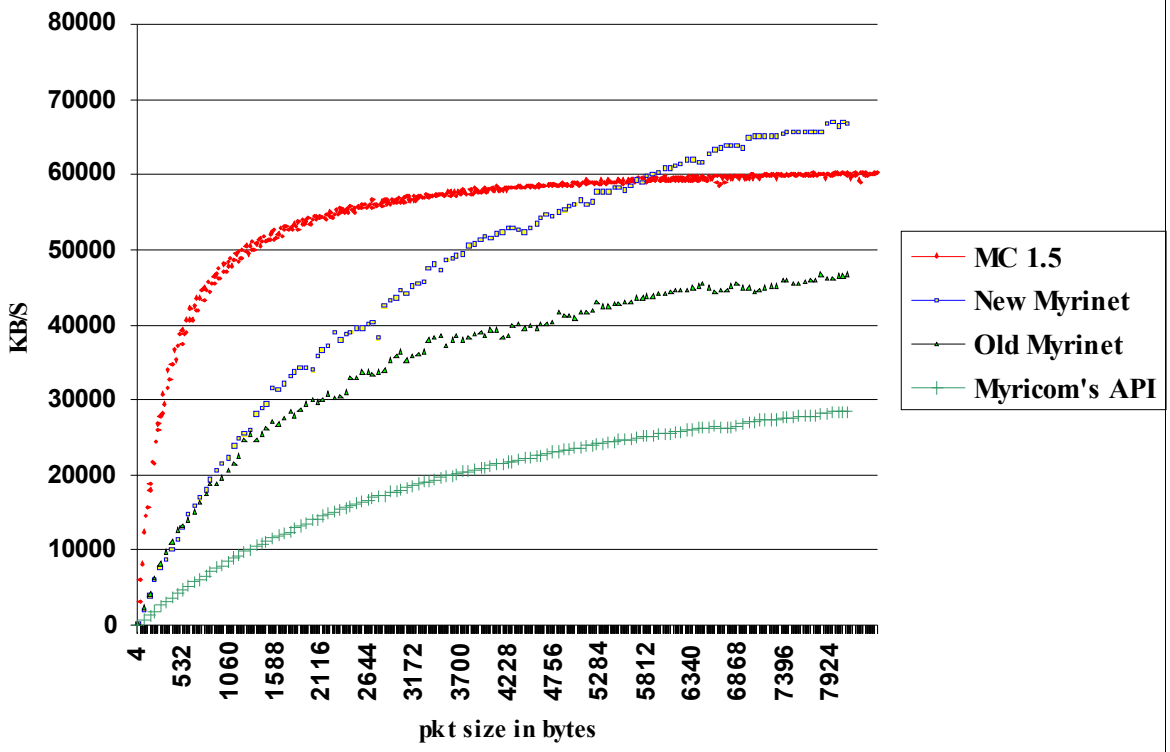
Performance Results

Performance results were obtained on two Digital 1200 AlphaServers with dual 533 MHz Alpha processors and 512M of memory. One PCI Myrinet card was used in each machine and they were connected through a Myrinet 8-way switch.
Performance numbers for 400MHz Digital 4100 AlphaServers were a few μ seconds faster in packet latency. This is due, we believe, to the superior memory subsystem on the 4100s. DMA performance on both systems, as measured by a special purpose MCP from Myricom, shows identical DMA performance. The gain in latency is apparently made in the areas where the processor writes across the PCI bus in a PIO fashion. This is done when enqueuing a packet, and can be done with packet data too if the packet size is below the PIO threshold.

The transports compared are Digital's Memory Channel v1.5, the old Myrinet system which did not use overlapped DMA on receives, the new Myrinet system which does do overlapped DMA on receives and Myricom's API and MCP. Also, the Memory Channel tests were run on 4100s. Since all of the MC writes are done via PIO, they gained the benefit of the 4100's better memory performance. We now have additional MC cards for use in the 1200s, and if time permits we can get some numbers for MC on this platform, too.

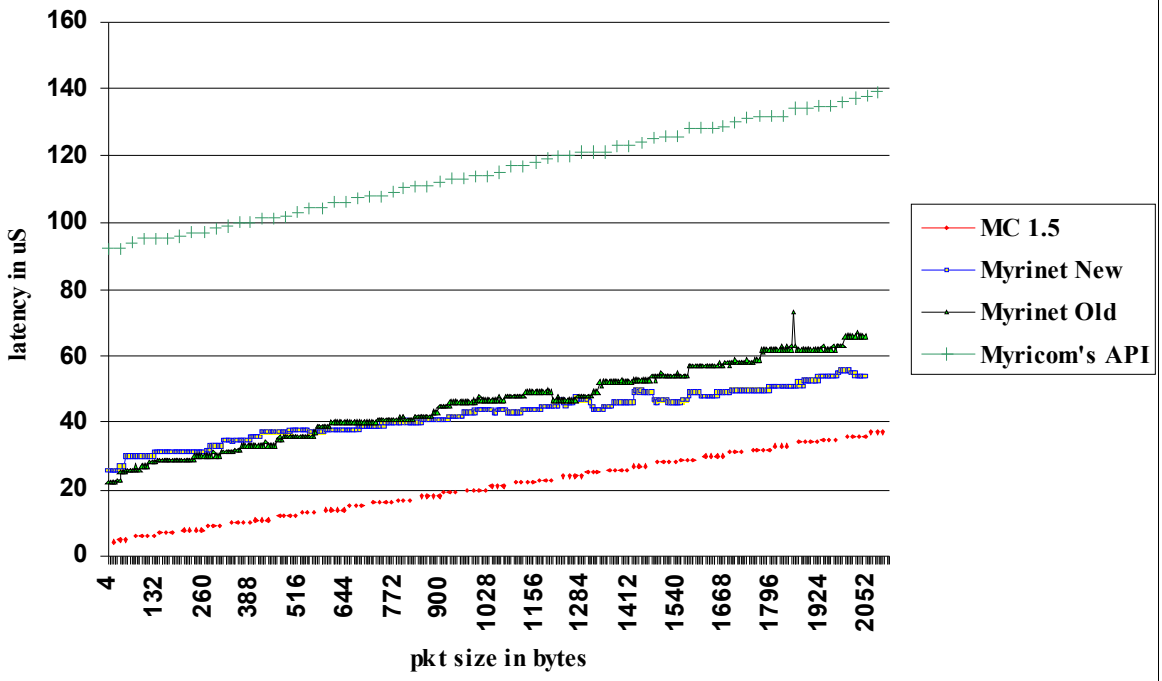
Single Packets, up to 8K

These are the performance numbers for a ping-pong test sending a single packet back and forth between two nodes. The bandwidth is calculated by dividing the total number of bytes sent by two and then by the total time it takes for the run. The bandwidth here is essentially the inverse of the latency for a single packet. We see quite clearly the benefit of overlapped DMA on receives.



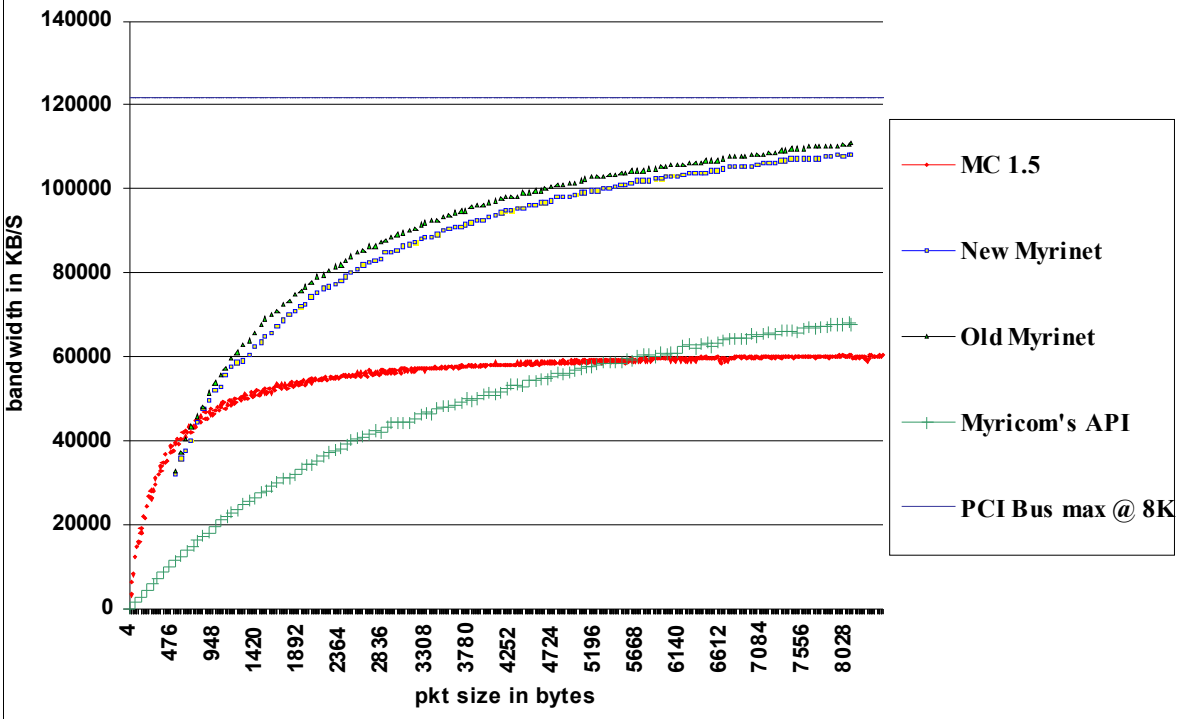
Small Packet Latencies

This result shows latencies for small packets. We see that MC is best by far. Here also we see that the move to overlapped receives has cost us some latency at smaller packet sizes. However, once we pass 800 bytes or so the new system begins to perform better.



Multiple 8K Packets (pipelining)

This result shows flat out bandwidth. The Myrinet systems are fully utilizing the pipelining in the system. The test sends 200 packets from the source to the destination. After 200 packets are received, the destination node sends an ack. We divide the total number of byte sent by the time until we receive the ack. We use a one way packet stream so that we don't experience contention for the host DMA engine. For Myrinet, we send up to 8 packets at a time (the send channel max). This allows the send pipelining to become active. MC, due to its architecture, does not really allow for pipelining, so the BW here is the same as for a single packet. It is still a mystery as to why the new myrinet system is slower even when pipelining is active.



CODE
& Date

References

[1] Duke Trapeze Project: LANAI - Flotsam & Jetsam
http://www.cs.duke.edu/ari/manic/tpz_www/notes.html

Other, general references:

The Duke Trapeze Project

Myricom's source code for their API and MCP

Myricom's The Myrinet API

Myricom's LANai4.x.doc

CLF source code and documentation, R.S. Nikhil

Appendix A – API Functions

```
/*
************************************************************************
*
* Initialize the API for use by a client program
*
* A counting semaphore is used to help determine if we are the first user
* of the board.  If we are, then full initialization is done.  Otherwise
* we assume the board is initialized already and we just make references
* to the board.
* Full initialization include:
* o Possibly loading MCP code; see mcp_code param.
* o Handshaking with the board
* o Creating/Initializing the shmem seg where send channel info is
*   shared.
* o Initializing the send channel
* o Initializing the route table
* o adding the zero route
*
* Args:
*   unit_num - which board we are interested in initializing
*   in_id - board id... not really used.
*   force_init - force init of the board even if the counting semaphore
*     of board users does not indicate that we are the first user of
*     the board.  USE WITH CAUTION
*   mcp_code - pointer to string holding name of mcp code file.
*     NULL or empty means do not load code.  Only used if we are the
*     first current caller of init or force_init is specified.
*
* Returns:
*   =0 Ok.
*   <0 Error. See myri_api0_rc.h
*
************************************************************************
*/
int
myri_api0_init(
    int         unit_num,
    int         in_id,
    int         force_init,
    char*   mcp_code)
```

```
/*
************************************************************************
*
* initialize a recv channel.
* Recv chans are not intended to be shared.
*
* Args:
*   unit_num - the board to use
*   chan_num - the recv channel on unit_num to use
*
* Returns:
*   =0 Ok.
*   <0 Error. See myri_api0_rc.h
*
************************************************************************
*/
int
myri_api0_init_recv_chan(
    int     unit_num,
    int     chan_num)
```

```
/*
************************************************************************
*
* get_send_buffer[_l]
* Get a send buffer from the board's dma space to use for sending a
* packet.
* Args:
*   unit_num - which board to get the buffer from.
*   buf - where to stick the buffer pointer.
*
* Returns:
*   =0 Ok.
*   <0 Error. See myri_api0_rc.h
*
************************************************************************
*/
int
myri_api0_get_send_buffer_l(
    int          unit_num,
    void**  buf)
```

```
/*
************************************************************************
*
* myri_api0_free[_l]
* Free a send buffer after it has been sent to the board.
* A freed buffer is not guaranteed to have been sent by the board,
* however, once freed the buffer will not be returned to the user via
* a call to myri_api0_get_send_buffer[_l] until the buffer is sent
* by the board.
* Args:
*   unit_num - board to free buffer to.  This *MUST* be the board we
*     got the buffer from.  No check on this is made.
*   buf - buffer to free.
*
* Returns:
*   =0 Ok.
*   <0 Error. See myri_api0_rc.h
*
************************************************************************
*/
int
myri_api0_free_send_l
(
    int         unit_num,
    char*   buf)
```

```
/*
************************************************************************
*
* myri_api0_send_route[_l]
*
* The buffer is queued to the single send channel on the board unit_num.
* The route corresponding to route_id is used to route the message to the
* destination.
* the _l version does locking around important internal data
* structures,
* providing exclusion between multiple calls to this function and
* calls to myri_api0_free_send[_l] and calls to
* myri_api0_get_send_buffer().
*
* If you are certain that only one call is going to be made to any of
* the above mentioned functions, then it is safe to use the non-locking
* version.
*
* A buffer should not be used after it has been queued to the board
* unless a call to myri_api0_buffer_busy_p() indicates that the buffer
* is no longer busy.  A sent buffer, however,  can be freed any time
* after it is sent.  See myri_api0_free_send_[_l].
*
* Args:
*   unit_num - which board to use
*   chan_num - which channel to send to.  Message will show up in
*     recv_chan[chan_num] on receiver.
*   buf - message buffer
*   buf_len - len of message buffer
*   route_id - id of a previously registered route via
myri_api0_add_route()
*
* Returns:
*   =0 Ok.
*   <0 Error. See myri_api0_rc.h
*
************************************************************************
*/
int
myri_api0_send_route_l
(
    int         unit_num,
    int         chan_num,
    void*   buf,
    int         buf_len,
    int         route_id)
```

```
/*
**************************************************************************
*
* see if there is a buffer ready from the myrinet.  if so, return a
* pointer to it.  Otherwise return an error indication.
* There is no locking around access to recv channels.  They are
* intended
* to be used to allow two end-points to communicate.  Any shared use of
* a recv channel must be implemented outside of the api.
*
* Args:
*   unit_num - board to use
*   chan_num - channel to use
*   buf - will hold pointer to newly received buffer
*   buf_len - len of newly received buffer
*
* Returns:
*   =0 Ok.
*   <0 Error. See myri_api0_rc.h
*
**************************************************************************
*/
int
myri_api0_recv(
    int          unit_num,
    int          chan_num,
    void**  buf,
    int*    buf_len)
```

```
/*
************************************************************************
*
* Allow user to check for buffers without consuming any.
* Args:
*   unit_num - board to use
*   chan_num - channel to use
*
* Returns:
*   =0 No recv buffers
*   >0 Num recv buffers ready
*   <0 Error.  See myri_api0_rc.h
*
************************************************************************
*/
int
myri_api0_recv_poll(
    int         unit_num,
    int         chan_num)
```

```
/*
****************************************************************************
*
* free a recv buffer back to the MCP so it can rx into it.
* Args:
*   unit_num - board to use
*   chan_num - channel to use
*   buf - buffer to be freed.  This must be a buffer gotten from this
*     channel via myri_api0_recv()
*
* Returns:
*   =0 Ok.
*   <0 Error. See myri_api0_rc.h
*
****************************************************************************
*/
int
myri_api0_free_recv(
    int          unit_num,
    int          chan_num,
    void*   buf)
```

```
/*
****************************************************************************
*
* Add a route to the board.
*
* It is the responsibility of the user of the API to
* allocate routes in a cooperative fashion.  A reinit
* of someone else's route is not be detected and will
* screw them up.  All future sends using that route will
* use the new one.
* Using MYRI_API0_NEXT_ROUTE as the route ID helps to prevent this.
* Note: it is possible to get a system assigned route id and still
* overwrite it by using the same route id.  The system assigned route
* ids are not special in any way.
*
* Args:
*   unit_num - board to use
*   route_id - number to use.  if route_id == MYRI_API0_NEXT_ROUTE,
*   then use the next free route slot.
*
* Returns:
*   >0 The route_id
*   <0 Error. See myri_api0_rc.h
*   =0 Should never happen
*
****************************************************************************
*/
int
myri_api0_add_route(
    int         unit_num,
    int         route_id,
    char*   route,
    int         len)
```

```
/*
****************************************************************************
*
* Returns, as well as can be expected since the mcp runs concurrently,
* the number of pending IOs on the board.
* Args:
*   unit_num - board to use
*
* Returns:
*   >= 0 Number of pending send IOs on the board
*   <0 Error. See myri_api0_rc.h
*
****************************************************************************
*/
int
myri_api0_num_pending(
    int      unit_num)
```

```
/*
************************************************************************
*
* Grab a lock on the send channel
* Args:
*   unit_num - the board to use
*
* Returns:
*   =0 Ok.
*   <0 Error. See myri_api0_rc.h
*
************************************************************************
*/
int
myri_api0_lock(
    int     unit_num)
```

```
/*
************************************************************************
*
* Release a lock on the send channel
* Args:
*   unit_num - the board to use
*
*
* Returns:
*   =0 Ok.
*   <0 Error. See myri_api0_rc.h
*
************************************************************************
*/
int
myri_api0_unlock(
    int      unit_num)
```

```
/*
****************************************************************************
*
* handshake w/MCP
*
* Args:
*   unit_num - the board to use
*
* Returns:
*   =0 Ok.
*   <0 Error. See myri_api0_rc.h
*
****************************************************************************
*/
int
myri_api0_handshake(
    int      unit_num)
```

```
/*
************************************************************************
*
* Return true if the buffer is busy.
*
* Args:
*   unit_num - the board to use
*   the buffer to check.
*
* Returns:
*   =0 Not busy
*   >0 Busy
*   <0 Error. See myri_api0_rc.h
*
************************************************************************
*/
int
myri_api0_buffer_busy_p(
    int          unit_num,
    void*   buf)
```

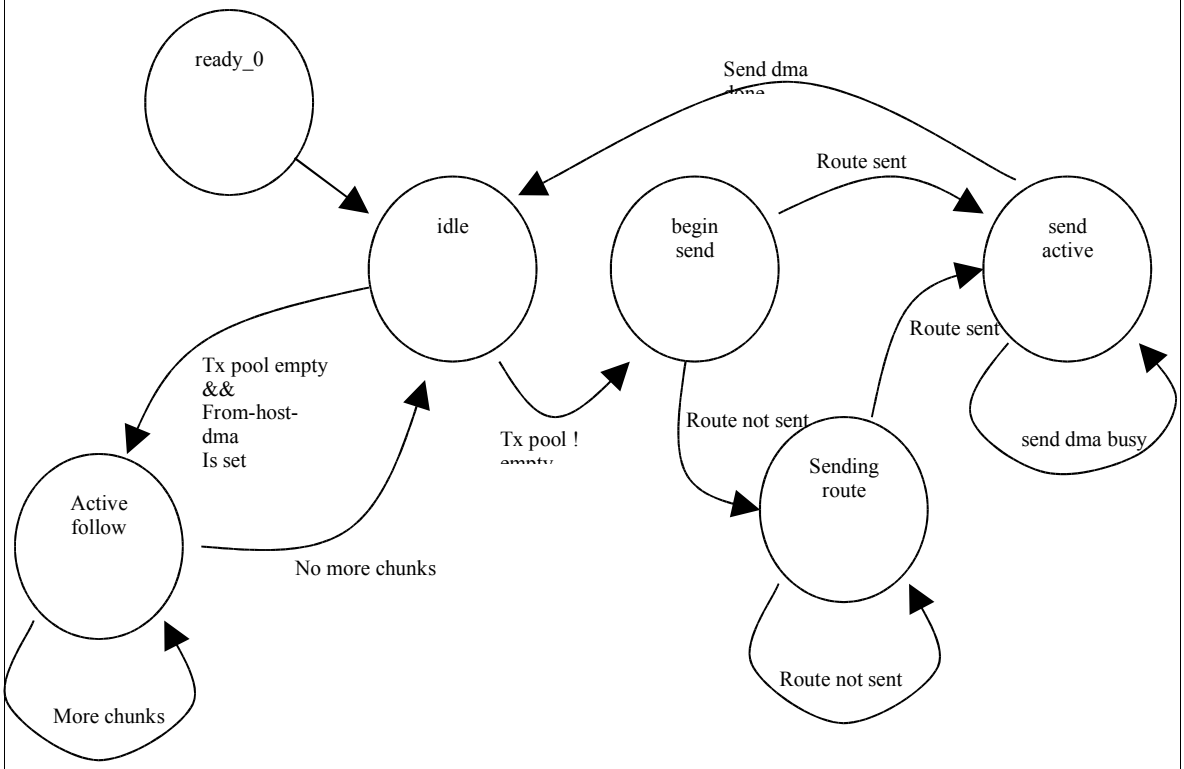
```
/*
****************************************************************************
*
* Asks the send channel to make a sweep thru its list of sent buffers
* to see if any of them have been sent.  This is the only way to
* check for completed buffer.  The myri_api0_get_send_buffer[_l]
* functions make calls to an internal version of this function to get
* the same results.
*
* Args:
*   unit_num - which board to use
*
* Returns:
*   =0 Ok.
*   <0 Error. See myri_api0_rc.h
*
****************************************************************************
*/
int
myri_api0_reap_sent_l(
    int          unit_num)
```

```
/*
**********************************************************************
*
* Get a channel on the indicated unit.  You can try to get a specific
* channel or try for any available channel.
* Args:
*   unit_num - which board we are interested in initializing
*   chan_num - A specific channel number:
*     (0 <= chan_num < MYRI_API0_NUM_CHANNELS) or MYRI_API0_ANY_CHAN
*     to as for any free channel.
*
* Returns:
*   >0 Ok, Channel number.
*   <0 Error.
*     MYRI_API0_CHAN_IN_USE if specific chan is requested and it is
*         not available.
*     MYRI_API0_NO_CHANS if ANY chan is requested and none are
*     available.
*
**********************************************************************
*/
int
myri_api0_get_chan(
    int unit_num,
    int     chan_num)
```

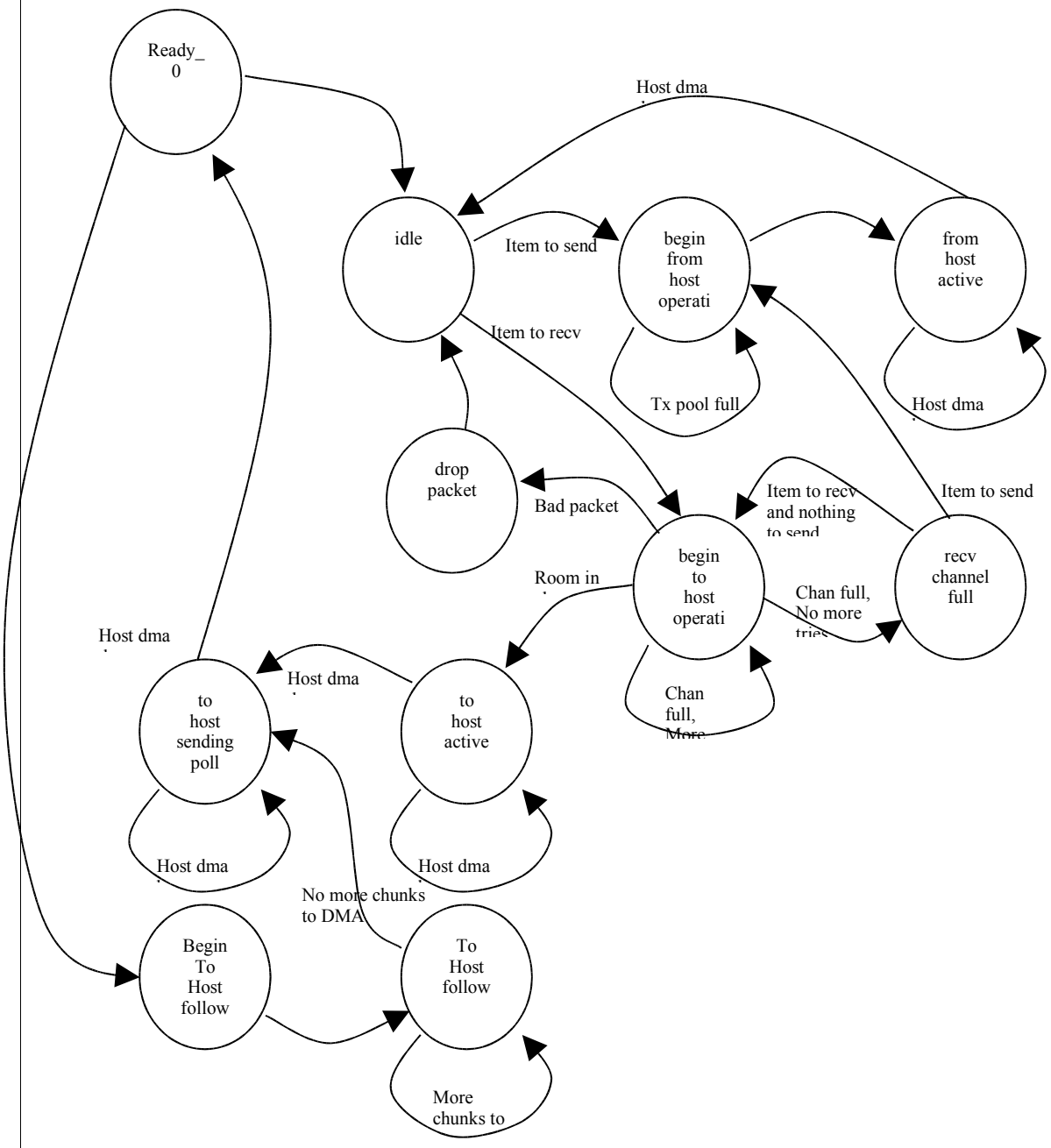
```
/*
**********************************************************************
*
* Free the channel on the indicated unit.
*
* Args:
*   unit_num - which board we are interested in initializing
*   chan_num - A specific channel number to free.
*
* Returns:
*   =0 Ok.
*   <0 Error.
*     MYRI_API0_CHAN_NOT_IN_USE if you haven't gotten this channel.
*
**********************************************************************
*/
int
myri_api0_rele_chan(
    int unit_num,
    int     chan_num)
```

Appendix B – State Machine Diagrams

Send DMA State Machine



Host DMA State Machine



Receive DMA State Machine