The Isotach Messaging Layer: Ironman Design

Michael N. Lack, Perry N. M. Myers
Paul F. Reynolds, Craig C. Williams

Technical Report CS-2000-17
Department of Computer Science
University of Virginia
Charlottesville, VA 22903
E-mail: [mnl3j, pnm2h, pfr, ccw]@cs.Virginia.EDU

May 9, 2000

Contents

Contents ii

1. Introduction 1
    1.1. System Overview 1
    1.2. Design Goals 2
    1.3. Document Overview 2

2. Isotach Messaging Layer Overview 4

3. Protocol Overviews 8
    3.1. Non-Isotach Overview 8
    3.2. Non-Isotach Group Communication........ 9
    3.3. Isotach Message Based Model Overview 10
    3.4. Isotach Shared Memory Model Overview 14
    3.5. Isotach Group Communication........ 18
    3.6. Flow Control 20
    3.7. Network Interface Unit Send/Receive Path 23

4. Messaging Layer Shared Data Structure 26
    4.1. Host mLayer Shared Data 26
    4.2. Host/NIU mLayer Interface Shared Data........ 30

5. Host Manager (hostman) Layer 32
    5.1. Host Manager (hostman) Module........ 32
    5.2. Isotach Program Initialization........ 35
    5.3. Network Status Table 36
    5.4. Allocation of Pinned Memory........ 36
    5.5. Synchronization 37
    5.6. Putting It All Together........ 38
    5.7. Isotach Program Termination........ 39

6. Application Programming Interface (API) Layer 40
    6.1. Non-Isotach Send (send) Module........ 40
    6.2. Isotach Send (iso_send) Module 41
    6.3. Non-Isotach Barrier (barrier) Module........ 45
    6.4. Isotach Signal (iso_signal) Module........ 47
    6.5. Isotach Barrier (iso_barrier) Module 50
    6.6. Isotach Retrieve (iso_retrieve) Module........ 54
    6.7. Non-Isotach Delivery (deliver) Module 54
    6.8. Isotach Delivery (iso_deliver) Module........ 56

7. Processing Layer 59
    7.1. Non-Isotach Flow Control (flow) module........ 59
    7.2. Isotach Flow Control (iso_flow) Module........ 62
    7.3. Isotach Ordering Module (IOM) 66

7.4. Isotach Shared Memory Manager (shmem) 69

8. Network Interface Unit (NIU) Layer 72
8.1. Non-Isotach Shipping (shipping) Module 72
8.2. Isotach Shipping (iso_shipping) Module.................................... 73
8.3. Non-Isotach Receiving (receive) Module 73
8.4. Isotach Receiving (iso_receive) Module 76

9. Network Manager (netman) Layer 78
9.1. Internal Functions 78
9.2. Exported Data Structures.. 78
9.3. Internal Data Structures 79
9.4. Tasks 79

10. Performance Results and Analysis 83
10.1. Overview of tests 83
10.2. Testbed.. 83
10.3. Results... 84
10.4. Bottleneck Analysis.. 85

11. Conclusions and Future Work 89
11.1. State of the system.. 89
11.2. Performance.. 89
11.3. Future Work 89

References 91

Glossary 92

Appendix A. Isotach API 94
A.1. Initialization/Shut-down 94
A.2. System housekeeping and status functions................................ 95
A.3. Non-Isotach Message Based Model... 96
A.4. Non-Isotach Group Communication.. 97
A.5. Isotach Message Based Model 98
A.6. Isotach Shared Memory Model 99
A.7. Message functions common to the Isotach SMM and MBM.... 100
A.8. Isotach Group Communication (GC)....................................... 100

Appendix B. Packet Formats 104
B.1. Network Packet Formats.. 104
B.2. Internal Packet Data Structures 107

Appendix C. Isotach Implementation Constants 109

Appendix D. Performance Testing Data 112

Appendix E. Configuration Files 113

Appendix F. Development Environment 115
F.1. Directory Structure.. 115
F.2. Module Components... 115

F.3. Makefiles 116
F.4. Scripts 118
F.5. Queue Macro Functions 119
F.6. LANai Debugging Tools 119
F.7. API Design Issues 120

Appendix G. Programming Conventions 122
G.1. Programming Style 122
G.2. Commenting Style 123
G.3. Version Control 124

Appendix H. Isotach Computers and Myrinet Drivers 126
H.1. Installation of the Isotach File Server 126
H.2. Installation of the Isotach Client Computers 128
H.3. If the Myrinet Driver does not load 129
H.4. Modification of bigphysarea 129
H.5. Modification of the Myrinet Drivers 130
H.6. Recompilation of the Myrinet Driver 130
H.7. Manual Reloading of the Myrinet Driver 131
H.8. The `CRT.O` Module for the LANai Compiler 131

Appendix I. Installation and Configuration Scripts 132

Appendix J. Isotach Source Code 141

1. Introduction

This document specifies the Ironman design of the Isotach prototype messaging layer. Ironman builds on both the Tinman [Wil99] and Strawman [BSV99] designs previously proposed, and represents the current development state of the Isotach messaging layer. Everything described in this document has been implemented, debugged and tested with the exception of the Isotach Shared Memory Model.

1.1. System Overview

The following information is provided only as a brief overview. It is assumed that the reader has a working knowledge of both Isotach systems and Myrinet networks. More information can be found in [Bar99], [Reg99], [RWW97], [Sza99], [Wil93], [Myr01], [Myr02], [Myr03].

1.1.1. Isotach Systems

In traditional parallel and/or distributed systems, synchronization among nodes requires relatively expensive mechanisms such as locks and barriers. An Isotach network inexpensively provides strong guarantees regarding message delivery order, thus allowing for sequential consistency and atomicity at a much lower cost.

Isotach ordering is an extension of Lamport's [Lam78] logical time algorithm. Every host in the network is aware of a logical time that is advanced through a token passing mechanism. The *Isotach invariant* states that a message issued by host A intended for host B at logical time T will be delivered at logical time T+D, where D is the logical distance between A and B. Thus, messages travel through the network at the rate of one unit of logical distance per unit of logical time. Assuming fixed distances between hosts, a sender can predetermine the exact logical delivery time of an issued message. This is the basis for the Isotach concurrency control mechanisms.

1.1.2. Myrinet Networks

Ironman has been implemented on a Myrinet network. A Myrinet network is switched Gb/s low latency network that is practically error free. Additionally, the network provides point-to-point FIFO packet delivery.

The Myrinet interface card contains a general-purpose microprocessor called the LANai. The LANai executes its own control program that can be written in a high-level language (such as C), and compiled into executable format. Additionally, the interface card contains 1MB of high performance SRAM and three different DMA engines: one to transfer data to the network from SRAM, one to transfer data from the network to SRAM, and one to transfer data between SRAM and the physical memory on the host. Additionally, the host can access SRAM through programmed I/O.

1.1.3. Custom Hardware

Although the initial Isotach prototype was developed entirely using software techniques, it has been determined that a substantial increase in performance can be obtained by moving some of the functionality to custom hardware devices. Two such devices – a token manager (TM) and a switch interface unit (SIU) – have been designed and debugged. The TM is connected directly to a switch, and is responsible for advancing logical time. The SIU is situated between the network interface on a host and a switch, and is primarily responsible for communicating with the TM, computing delivery times for messages, and determining the delivery order of incoming messages. Further performance testing will need to be done in order to determine whether the custom hardware devices significantly improve the performance of Isotach networks.

Ironman has been designed and implemented to work seamlessly with both the TM and SIU, and relies on their functionality. There is currently no support within the messaging layer for an all software Isotach implementation.

1.2. Design Goals

Ironman has been designed as a modularized system that is optimized to perform well with the hardware token manager and switch interface units. The ultimate goal is a fast, efficient messaging layer that provides Isotach functionality, as well as a baseline Non-Isotach functionality. Our goal is to design and implement a Non-Isotach messaging layer that is comparable in performance to other available messaging layers. Along with this goal, we wish to show that that Isotach guarantees should not generate an overhead of more than twice the latency and one half the throughput of a comparable protocol. Ironman is an attempt to demonstrate the correctness of these hypotheses. Preliminary results indicate that we have achieved most of these goals. Further information on performance testing is given later in this document. (See Chapter 10)

1.3. Document Overview

The main purpose of this document is to supply a complete description of the current Isotach prototype. Section 2 provides a brief overview of the messaging layer. Section 3 is a high level overview of the different protocols and send/receive paths within the messaging layer. Section 4 is a summary of data structures that are shared among different modules in the messaging layer. Sections 5-9 give a detailed description of the design of each layer and module. Section 10 describes our performance tests and results, along with an analysis of potential bottlenecks in the system. Finally, we conclude with some preliminary performance results and a description of future work.

Additional information about the system can be found at the end of the document in the glossary and appendices. Appendix A specifies the API for all supported protocols. Appendix B describes the format of packets both internally and out on the network. Appendix C provides a list of tunable system parameters. Appendix D contains our performance testing data. Appendix E describes the format of the configuration files that the messaging layer utilizes. Appendices F, G and H outline the development environment and coding conventions used to implement Ironman.

Finally, appendices I and J contain the source to various configuration scripts and the messaging layer source itself. This document has been developed as a hyper linked document so that following a link will lead the reader to additional information about a particular topic.

2. Isotach Messaging Layer Overview

The mLayer supports the following application level interfaces (see Appendix A for details): noniso MBM; noniso GC (group communication), iso MBM; iso SMM; and iso GC. To implement these interfaces, the mLayer can send several types of packets. There are two basic packet types: Isotach and noniso. Isotach packets are ordered by the SIU's. Noniso packets are not. Each packet type is further divided into subtypes as summarized in Figure 1. Noniso MBM interface messages are sent by the mLayer using noniso packets of subtype `noniso_mbm`. Each Isotach MBM message is sent by the mLayer as a pair of packets: a noniso packet of subtype `ordered` that contains the message payload and an Isotach packet of subtype `iso_pointer` that contains a pointer to the ordered packet and controls the order in which ordered packets are delivered at the receiving node. Isotach SMM memory accesses are sent using Isotach packets of subtype `iso_read`, `iso_write`, `iso_assign`, or `iso_sched`. Responses to reads are noniso packets of subtype `read_response`. The mLayer services GC calls from the application by sending Isotach packets of subtype `bs_marker` (for Isotach signals and barriers) or subtype `barrier` (for noniso barriers). The former packets are barrier-signal markers that go to the neighboring SIU. The credit packets are used in flow control and the sync packets are used during initialization. See Appendix B for a full description of internal and external packet formats.

mLayer Packet Subtypes		
Non-Isotach Packets	noniso_mbm	Ordinary Non-Isotach packets
	ordered	The payload of Isotach messages are sent as Non-Isotach packets
	read_response	Responses to Isotach reads are sent as Non-Isotach packets
	credit	Used in Non-Isotach flow control
	iso_credit	Used in Isotach flow control
	barrier	Non-Isotach barrier packets.
	credit_request	Used in Non-Isotach flow control
	sync	Used in synchronization
	sync_ack	Used in synchronization
	sync_done	Used in synchronization
	iso_pointer	A pointer to an ordered packet, used for Isotach Message Based Model (MBM) messages
Isotach Packets	iso_read iso_write iso_assign iso_sched	Isotach Shared Memory Model (SMM) sRefs
	iso_marker	Isochron marker sent back by the SIU
	eop_marker	End of pulse marker sent back by the SIU
	bs_marker	Barrier/Signal information sent to the SIU

Figure 1 - Packet Types and Subtypes

In structure, the mLayer consists of several layers representing the different stages of a packet's journey through the mLayer. Each layer is further divided into modules that perform a task specific to that layer. In addition, each module exports functions and data structures to the other modules. The relationship between modules is shown in Figure 2. In outline, the mLayer structure is as follows:

Host mLayer

- Hostman (host mLayer manager): initializes and shuts down host mLayer and loads the NIU mLayer. Exports the `poll()`, `open_net()`, `try_close_net()`, and system status API functions.

API layer

Supports functions visible to the Isotach API (see Appendix A).

- Send (noniso send): performs initial send side processing of noniso messages.
- Iso_send (Isotach send): performs initial send side processing of Isotach messages and memory accesses.
- Iso_barrier: handles (registers, clears, and participates in) Isotach barriers.
- Iso_signal: handles (registers, clears, and sends) Isotach signals.
- Iso_retrieve: returns to the application the result of previously executed read accesses to Isotach shared memory variables.
- Delivery: delivers noniso messages to the application.
- Iso_delivery: delivers Isotach messages and barrier/signal-notices to the application.

Processing layer

Manages flow control, group communication, and Isotach ordering.

- Flow (noniso flow control): implements flow control for noniso protocol packets.
- Iso_flow (Isotach flow control for Isotach packets): implements Isotach-based flow control.
- Barrier (noniso): implements noniso barrier.
- IOM (Isotach ordering module): sorts Isotach packets and bs-notices into Isotach logical time, separate packets that should go to the shmem from those that go to the application.
- Shmem (shared memory module): executes locally and remotely issued accesses to local portion of Isotach shared memory; generates read responses to remotely issued reads.

NIU interface layer

Sends packets to the NIU and receives packets from the NIU.

- Shipping: looks up and appends routes to packets and writes noniso packets into the noniso send buffer on the NIU.
- Iso_shipping: looks up and appends routes to packets and writes Isotach packets into the Isotach send buffer on the NIU.
- Receiving: processes received noniso packets (including ordered packets), i.e., check the CRC, garner the credit information, and enqueue a pointer to the message in the noniso delivery queue.
- Iso_receiving: performs initial receive side processing on Isotach packets (checks the CRC and passes the packets to the IOM.)

NIU mLayer

- Netman (network interface unit manager): initializes the NIU mLayer, puts packets onto the wire, receives packets from the wire, and transfers received packets from on-board memory into pinned memory on the host.

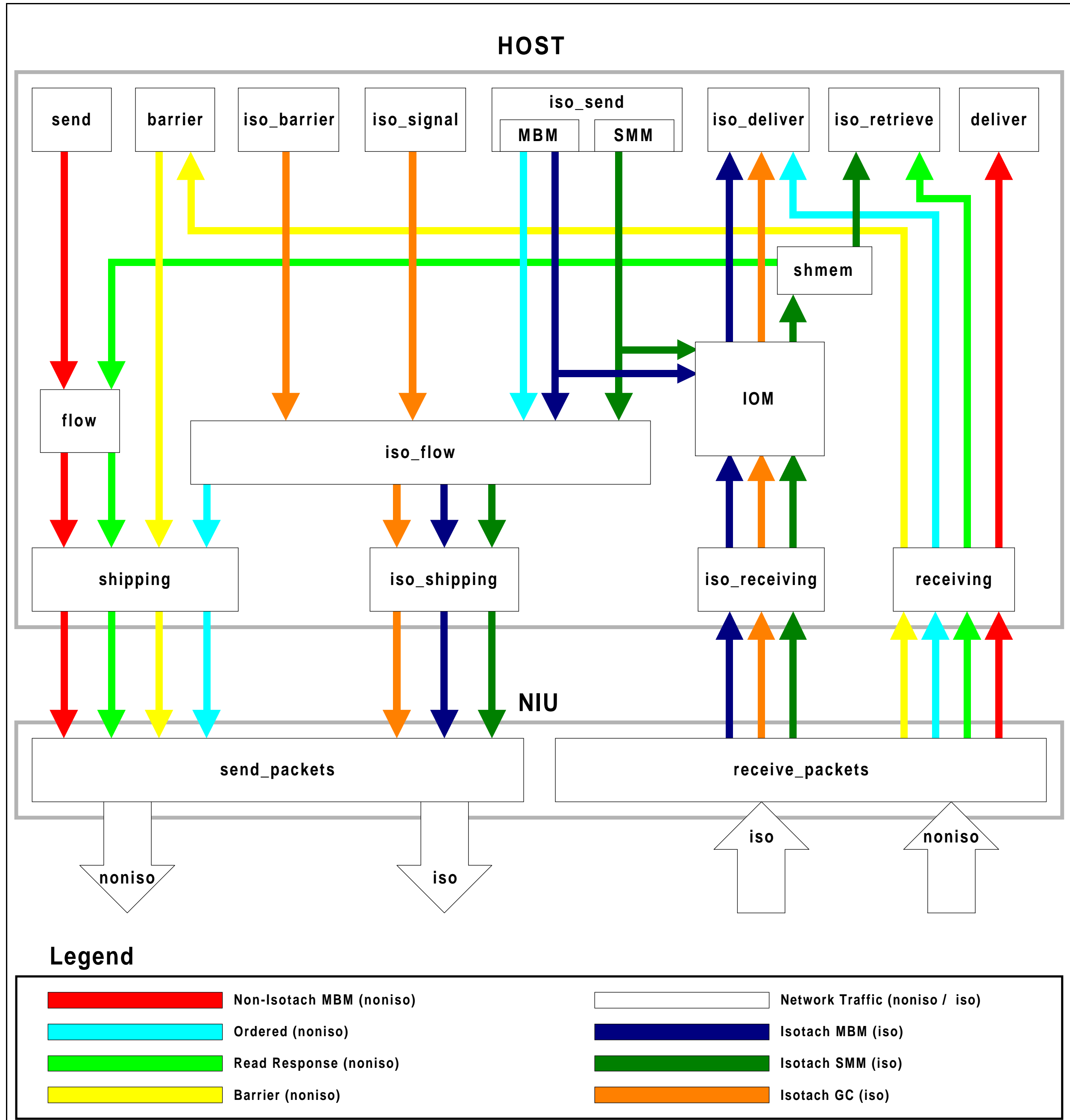


Figure 2 - MLayer Module Block Diagram

Flow of control

The modules are designed as independently executing processes communicating through buffers. This design separates the flow of control from the flow of information (packets).

For good performance, the application must frequently hand the mLayer control. It does so by calling the `poll()` function or by sending/receiving a message/access. The latter way (sending/receiving a message/access) is an indirect version of the former (poll) since each API function for sending/receiving a message/access calls `poll()` after performing the specified request.

When `poll()` is called, the hostman module, which exports `poll()`, calls each of the poll functions in the other modules in the host mLayer. Each module, in general, does as much work as it can before relinquishing control.

Programming note

The use of the term "module" is not intended to imply use of an object-oriented programming language. The implementation language is C. A measure of modularity can be achieved by using separate files and static declarations of functions and external variables. See Appendices E and F for more information.

3. Protocol Overviews

This section gives an overview of the different protocols that the messaging layer utilizes. It also describes flow control for each of these protocols and the functioning of the Netman module. The five different protocols discussed are:

1. Non-Isotach Message Based Model (noniso-MBM)
2. Non-Isotach Group Communication (noniso-GC)
3. Isotach Message Based Model (iso-MBM)
4. Isotach Shared Memory Model (iso-SMM)
5. Isotach Group Communication (iso-GC)

For further information on these application programming interfaces, please see Appendix A.

3.1. Non-Isotach Overview

3.1.1. Send Path

The application initiates a send by executing a `send()` function call giving a pointer to the message to be sent, the receiver id, and the message length.

To execute the `send()` function, the send module first determines whether there is room in the host's noniso send buffer, `send_buf`. If not, the call to `send()` fails. Otherwise, the send module copies the message into `send_buf` leaving room for headers, fills in several of the header fields, and passes the packet to the flow module by incrementing `send_t`.

The flow module implements specific sender-based flow control. Information on flow control for Non-Isotach packets and the data structures referenced here are available later in this chapter. The flow module approves packets for shipping in FIFO order. If the packet at the head of the flow control segment of `send_buf` can be sent, flow assigns the packet to the tail slot of the remote buffer (by writing `remote_t[j]` into packet's DMA base, where `j` is the destination node, and incrementing `remote_t[]`). Flow piggybacks send credit information onto the packet and passes the packet to the shipping module by incrementing `send_flow_ptr`.

Shipping transfers packets from `send_buf` to the NIU. For each packet, it looks up the route for the packet, writes the route into the packet's route field and then transfers the packet into the tail slot of `niu_send_buf`, the NIU's send buffer, incrementing `send_t` and `send_h` to reflect the transfer.

The NIU DMA's packets in `niu_send_buf` onto the network in FIFO order. After sending a packet, the NIU releases the space held by the packet by incrementing the send buffer's head pointer.

3.1.2. Receive Path

The NIU continually checks for packets incoming from the network. It DMA's a received packet into the tail slot of `niu_receive_buf`, the NIU's receive buffer, and increments `niu_receive_t`.

The NIU then DMA's noniso packets in `niu_receive_buf` in FIFO order into the location starting at the physical address in pinned memory indicated by the packet's DMA base field. The length of the packet to be DMA'd is determined by the packet interface hardware as the packet is read in. The NIU records this length in the packet. After it DMA's the packet, the NIU enqueues a pointer to the packet's physical address (the DMA base) in the tail element of `niu_delivery_q` and increments `niu_delivery_t`.

Receiving acknowledges new incoming packets by incrementing `niu_delivery_h`. It converts the physical address it read off `niu_delivery_q` to a virtual address, checks the CRC of the packet, extracts credit information and passes the information to flow via a call to `update_credit()`. In the case of a packet of subtype `noniso_mbm`, receiving writes a pointer to the packet into the tail slot of `delivery_q` and increments `delivery_t`. In the case of a packet of subtype `ordered`, receiving registers the receipt of the packet by incrementing `ord_receive_t[s]`.

Delivery module exports `receive()` to the application. In response to a `receive()` call, delivery removes the pointer at the head of `delivery_q` (thus incrementing `delivery_h`) and passes the application the id of the packet's sender, a pointer to the packet's application payload, and the application packet length. The next time it gets control, delivery deletes the packet it delivered on the last `receive()` call via a call to `delete_packet()`. Delivery delays deleting the packet to ensure that the application has a chance to see the packet before it is overwritten.

The `delete_packet()` function is exported by flow. On a call to `delete_packet()`, flow increments `receive_buf_h[s]`, where `s` is the sender of the deleted packet. This new value will be written in the credit info field of the next packet sent to `s`. When necessary (see flow module), flow constructs an explicit credit packet that it passes to shipping through a call to `ship_packet()`.

3.2. Non-Isotach Group Communication

Non-Isotach barriers (noniso barriers) are currently the only implementation of group communication for the noniso protocol. This barrier construct was created to allow the synchronized termination of Non-Isotach applications (i.e. those applications that do not use any feature of the Isotach Protocols). The barrier is implemented using a simple `nxn` communication between all hosts. Each host must participate in the barrier, and receive n-1 barrier messages (one from every other host), in order for the barrier to complete. Currently, the only use for noniso barrier packets is during system shutdown, where a barrier is initiated upon a call to `try_close_net()`. This function is called repeatedly. When the barrier completes, `try_close_net()` returns SUCCESS and the application can safely terminate.

3.2.1. Send Path

A noniso barrier packet is created through a call to the function `initiate_barrier()` (which is also implicitly called through the first call to `try_close_net()`). The barrier packet bypasses flow control by being placed directly on the LANai's send buffer.

3.2.2. Receive Path

A noniso barrier packet is received by the LANai and placed in a special control slot in the sending host's area in pinned memory. A pointer to this location is placed in the NIU delivery queue. The receiving module on the host reads the pointer from the NIU delivery queue, and processes the barrier packet by calling a function in the barrier module. The application receives notification of the completion of a barrier by calling `barrier_completed()`, which will return a success code when a barrier has finished.

3.3. Isotach Message Based Model Overview

This section gives an overview of the send and receive paths for Isotach MBM messages.

3.3.1. Send path

The application initiates a send by executing an `iso_send()` function call giving a pointer to the message to be sent, the receiver id, and the message length.

To send an Isotach MBM message (other than a self-message, discussed later), the iso_send module will produce two packets:

- *ordered packet* – a noniso packet of subtype `ordered` which contains the message as the application payload.
- *iso-pointer* – a small Isotach packet of subtype `iso_pointer` that contains a pointer to the receive buffer slot to which the corresponding ordered packet is assigned.

In response to an `iso_send()` call, iso_send first determines whether there is room in both `ord_send_buf`, the send buffer for ordered packets, and `iso_send_buf`, the send buffer for Isotach packets. If either buffer is full, the call to `iso_send()` fails.

Otherwise, iso_send constructs the ordered packet in the tail slot of `ord_send_buf`, copying the message into the application payload and filling in several header fields, and then passing the packet to iso_flow by incrementing `ord_send_t`.

Iso_send constructs the iso-pointer in the tail slot of `iso_send_buf`. Since the Isotach prefix differs depending on whether the packet is a start of Isochron (SOI), mid-Isochron, or end of Isochron (EOI) packet, iso_send tracks Isochron boundaries, updating the shared variable `mid_net_isochron`. Before it can advance `iso_send_t` to make the new packet visible to iso_flow, iso_send must determine whether the packet is an EOI packet. If the application calls `iso_send()` with the argument `last_in_isochron` set to `TRUE`, iso_send can advance `iso_send_t` immediately after constructing the iso-pointer. Otherwise, iso_send advances the

pointer only after the EOI status of the packet is determined. (For a more complete discussion, see the note on EOI status in the iso_send section.)

Iso_flow approves Isotach and ordered packets for shipping. If the Isotach packet at the head of the flow control segment of `iso_send_buf` can be sent and the packet is an iso-pointer, the corresponding ordered packet will be at the head of the flow control segment of `ord_send_buf`. Iso_flow performs the following tasks:

1.) assigns the DMA base to the ordered packet (see discussion of ordered flow control later in this chapter)
2.) writes the same (physical) address into the pointer field of the iso-pointer
3.) increments `iso_flow_ptr`
4.) increments `ord_flow_ptr` (if the packet on `iso_send_buf` was an `iso_pointer`) to approve both the iso-pointer and the corresponding ordered packet for shipping.

If the packet at the head of the flow control segment of `iso_send_buf` can be sent but is not an iso-pointer, iso_flow performs only step 3. When necessary, iso_flow constructs an explicit Isotach credit packet, which it passes to shipping through a call to `ship_packet()`.

Iso_shipping transfers packets from `iso_send_buf` to the NIU's `iso_niu_send_buf` in the same way shipping transfers packets from `send_buf` and `ord_send_buf`. It looks up the route for the packet, writes the route into the packet's route field and then transfers the packet into the tail slot of `iso_niu_send_buf`, incrementing `iso_send_h` and `iso_niu_send_t` to reflect the transfer.

The NIU DMA's packets in `iso_niu_send_buf` onto the network in FIFO order. After sending a packet, the NIU releases the space held by the packet by incrementing the send buffer's head pointer. (Analogous to send path for noniso packets)

3.3.2. Receive path

The receive path for ordered packets is the same as for other noniso packets except that instead of enqueuing a pointer to the packet on a delivery queue, receiving increments `ord_receive_t[s]`, where `s` is the id of the packet's sender. Incrementing the tail pointer into `s`'s ordered packet receive buffer allows iso_delivery to determine whether the ordered packet corresponding to the iso-pointer at the head of `iso_delivery_q` has been received. The remainder of this section discusses the receive path for iso-pointers.

The NIU continuously checks for packets incoming from the net. The NIU reads in the first word of an incoming packet to determine the packet's type. If it is an Isotach packet, the rest of the packet is then read directly into the `iso_niu_receive_buf`. The tail pointer to this buffer is incremented when the packet has completely arrived. The NIU also detects Isochron markers (by their size) and labels them for the host (for details see the Netman module description).

When the NIU copies an EOP marker into `iso_niu_receive_buf` or when the queue is full, it DMA's the contents of the buffer into `iso_receive_buf` in pinned memory, updates `iso_receive_t` to reflect the transfer and resets `iso_niu_receive_buf`.

Iso_receiving (on the host) acknowledges receiving newly incoming Isotach packets by incrementing `iso_receive_h`. Iso_receiving also checks the CRC of each packet and passes a pointer to the packet to IOM. Since iso_receiving deletes the packet after the call to `bucketize()` returns, IOM must copy all the information from the packet that it needs. For each packet it is passed other than an EOP marker, the IOM creates a shortened version (a `packet-core-3` structure, see Appendix B). A copy is performed in this case because the size of the copied data is very small and, without the copy, freeing space in `iso_receive_buf` would be complex since deletions occur in Isotach order, which is not the same as the insertion order.

The IOM maintains an array of buckets, one for each timestamp. If the pointer passed in a call to `bucketize()` refers to an iso-pointer or Isochron marker, the IOM enqueues a `packet-core-3` structure containing the subtype and the source fields and pointer field (iso-pointer) or iso_id field (Isochron marker) in the bucket specified by the packet's TS field. If the pointer refers to an EOP marker, the IOM uses the sort vector to move the packet-core structures to `iso_delivery_q`.

Iso_delivery delivers packets to the application in Isotach order. On an `iso_receive()` call, iso_delivery determines whether the ordered packet corresponding to the first iso-pointer on the `iso_delivery_q` has arrived. If the item on the head of the `iso_delivery_q` is not an `iso_pointer`, then it is a `bs_notice` or a self-message. See the section on Isotach GC and the following section on self messages for more information. for more information. If it has, iso_delivery hands the application a pointer to the application payload of the ordered packet, the length of the packet and the id of the sender. The next time it gets control, delivery deletes (via a call to `iso_delete_packet()`) the packet it delivered in the last `iso_receive()` call. The delay in deleting the packet is to ensure that the application has a chance to see the packet before it is overwritten.

3.3.3. Self-Messages and Isochron Markers

A self-message/packet is an Isotach message/packet sent by the issuing node to itself. A network message/packet is an Isotach message/packet sent to a remote node. An Isochron that contains *only* self-packets is a self-Isochron. An Isochron containing one or more net-packets is a net-Isochron (even if it also contains self-packets).

Self-messages are useful in the Isotach MBM protocol as a way to order local actions in relation to remote actions. A self-message that is part of an Isochron containing net-messages will be executed locally at the same logical time that the net-messages are executed remotely. A self-message issued after an Isochron will be executed at a logical time no earlier than the Isochron. In an SMM computation, local hits to Isotach shared variables are an important special case of self-packets called self-refs.

One way to implement the ordering required for self-messages is to send the messages into the network with a loop back route (and the appropriate logical distance). The SIU would then assign logical times to self-messages and include references to self-messages in the sort vector. The mLayer implements an alternative approach of holding self-messages within the host. The host enforces the same ordering guarantees for self-messages as for Isotach net-messages:

- *Isochronicity*: self-message *m* is executed at the same logical time as other messages in the same Isochron (whether they are self-messages or net-messages).
- *Sequential consistency*: self-message *m* is executed at a logical time no earlier than the logical time of any Isotach message issued by the same host before *m*.

Since the SIU, not the host, assigns logical times to messages, enforcing these ordering constraints requires communication between the SIU and host. For every net-Isochron that the host sends, the SIU returns an Isochron marker informing the host of the logical receive time it assigned to the Isochron. When this logical time arrives (signaled by the arrival of the EOP marker for that pulse from the SIU), the host executes/delivers all self-messages issued in the same Isochron as well as all self-messages issued by the host after the Isochron but before the next net-Isochron, that contains any net-messages. (We say these two classes of self-messages are executed *with* the Isochron marker.) The first class of self-messages are executed with the Isochron marker to enforce isochronicity; the second, to enforce sequential consistency.

The host thus requires a data structure that allows it find the self-packets that are to be executed with a given Isochron marker. This data structure is called the `hit_q` and is internal to the IOM. It has an entry called an `isochron-slot` for each locally issued net-isochron. The isochron-slot is a placeholder for the isochron marker that the SIU will return. Isochron markers and isochron-slots "match up", i.e., the order in which isochron-slot appear on `hit_q` is the same as the order in which the corresponding isochron markers become executable. (This property holds only if the SIU is enforcing sequential consistency over all Isotach messages). Each isochron slot contains a self-count field that gives the number of self-packets that are to be executed with the corresponding isochron marker. Since self-packets are stored in FIFO issue order (in another data structure: the `hit_buf` for MBM self-packets) and should be executed in FIFO order, the IOM does not need to store self-packets or pointers to self-packets in the `hit_q`. A simple count of the number of self-packets that should be executed with each net-Isochron's isochron marker is sufficient. The self-packets that should be executed are at the head of `hit_buf`.

When the application issues a self-message (i.e., when it calls `iso_send()` with target `my_id`), iso_send constructs the packet directly into the buffer from which it will be read, `hit_buf`.

At the end of each isochron, iso_send calls `post_isochron()` to tell the IOM:

1.) whether the isochron is a net-isochron
2.) the number of self-packets in this isochron
3.) if the isochron is a net-isochron, the isochron id (for use in a removable assertion claiming that isochron-slots and isochron markers match up).

If the isochron is a net-isochron, the IOM enqueues an isochron-slot for it on `hit_q` with the `self_count` field set to the self-count passed in as the second argument. Otherwise, if `hit_q` is empty, the self-packets in the isochron are executable. The IOM enqueues an isochron-slot with the specified self-count at the tail of `iso_delivery_q`. Finally, if the isochron is a self-isochron and `hit_q` is not empty, the self-packets should be executed with the isochron marker corresponding to the isochron-slot at the tail of `hit_q`. The IOM adds the count passed in as the second argument to the `self_count` field of the existing isochron-slot. As a result of this procedure, there is an isochron-slot on `hit_q` for each outstanding net-isochron and each isochron-slot has the count of the number of self-packets that should be executed with the corresponding isochron marker.

When iso_receiving gets an isochron marker, it passes it to the IOM, which adds an item (a `packet-core-3` structure of subtype `isochron_marker`) in the bucket for the marker's timestamp.

When the EOP marker for that bucket is received, the IOM handles the items in the bucket in order by sender. To handle an isochron marker, the IOM moves the isochron-slot at the head of `hit_q` to `iso_delivery_q`.

Each isochron-slot in `iso_delivery_q` represents `self_count` packets in `hit_buf` available for delivery to the application. When the item at the head of `iso_delivery_q` is an isochron-slot, iso-delivery delivers the first packet in `hit_buf` and decrements `self_count`. When `self_count` goes to zero, the isochron-slot is deleted from `iso_delivery_q`.

3.4. Isotach Shared Memory Model Overview

The Isotach SMM interface supports four Isotach SMM access functions (`iso_read()`, `iso_write()`, `iso_sched()` and `iso_assign()`). These functions access shared memory. The interface also supports `iso_retrieve()`, a function that does not itself access shared memory, but returns the value returned by a previous `iso_read()`.

NOTE: At the time of writing this document, Isotach SMM had not been fully implemented yet. See future work in the conclusion chapter for more details.

Each Isotach SMM access function call generates one or more *shared references* (sRefs). An `iso_read()` call generates a single sRef on the closest copy of the accessed variable `V`. A *change* (`iso_write()`, `iso_assign()` or `iso_sched()`) generates one sRef for each copy of `V`. There are four basic sRefs (split operations): read, write, sched, and assign. (Actually there are only three. The write is an optimization for the case where a process can issue an assign together with the corresponding sched.)

For a given sched, the *corresponding assign* is the assign that supplies the value that is to be written in the slot in `V`'s history reserved by the sched. Similarly, for a given assign the *corresponding sched* is the sched whose value the assign supplies. Operationally, an assign corresponds to a sched if it presents a version identifier (`vId`) that matches the sched's `vId`. We

use the `pId` for the process that issued the sched as the `vId` for the sched. We assume that the corresponding assign for each sched is issued by the same process that issued the sched and that each process has at most one unsubstantiated sched per variable. A sched is *unsubstantiated* if the corresponding assign has not been executed. An assign is always issued, received, and executed after the corresponding sched.

A *post* is a sched or a write. For each read, the *corresponding post* is the post whose logical receive time most closely precedes that of the read. The read returns the *value associated with the corresponding post*. (In the case of a write, the associated value is the value written. In the case of a sched, this value is the value supplied by the corresponding assign.) A variable `V` is *unsubstantiated* if the last post to `V` is an unsubstantiated sched. If `V` is substantiated, the SIU executes a read in the normal way, by returning the value of `V`. If `V` is unsubstantiated, the SIU executes the read by recording the read in association with the `vId` of the corresponding sched. When the assign with that `vId` is received, the SIU completes the execution of the read by returning the value assigned.

<u>Simplifying Observations:</u>

- A sched to `V` may be discarded if is not the last post to `V` and no reads to `V` were executed while the sched was the last post to `V`.
- The order among sched's on `V` preceding the last post to `V` does not need to be retained. Any subsequent read will return the value corresponding to the (current or future) last post.
- An assign that does not correspond to the last post or to a pending read corresponds to a sched that was discarded and can itself be discarded. (This observation requires the assumption that an assign is received after the corresponding sched and is the reason that assigns are sent as Isotach packets.)

<u>Requirements for implementing sRefs</u>

- For each variable, a way to record the status of each variable (i.e., whether it is substantiated). Each shared variable `V` is a structure containing two fields: `val`, `V`'s value; and `unsub`, set to 1 to indicate that `V` is unsubstantiated
- For each unsubstantiated variable, a way to determine the `vId` of the last sched. If a variable is unsubstantiated, the value in the `val` field is the `vId`.
- A way to find the reads pending on a given assign.

The shmem maintains a read pending table. Given `V` and the `vId` of a sched/assign, the read pending table finds the corresponding reads (i.e. the reads that are waiting for version `vId`) of `V` to be substantiated. This discussion does not specify the implementation of the read pending table. It is probably best implemented as a hash table (hashing on `V` or `V` concatenated with `vId`). In the v.1 mLayer, the read pending table is scanned linearly. In the worst case, there can be `n * read_cap entries`. Another implementation would use r tables, one for each of r partitions of shared memory.

Pseudo-code for each access

```
read(V, lvar, pId) ::
   ! the shmem executes read misses from remote nodes as well as from the local node
   if (V.unsub)                                    ! if V is unsubstantiated
      e = allocate entry in PENDING READ;          ! create an entry in PENDING READ cache
      e.variable = V's address;
      e.vId = V.val;!                              ! since V is unsub, V's value is a vId
      e.lvar = lvar;
      e.reader = pId;                              ! unique identifier for reader
   else
      if pId is my_id call handle_read_response()
      else send a read response packet to node pId;

sched(V, pId) ::
   set V.unsub;
   V.val = pId;

assign(V,vId,val)::
   if (V.unsub AND V.val = vId)        ! corresponding sched was last post to V so val is now V's value and
      V.val = val;                     ! V is now substantiated
      reset V.unsub;
   for each perfect match (both variable name & vId) in PENDING READ
      if reader is remote
         send a read response packet;
      else
         call handle_read_response();
   deallocate the entry in PENDING READ;

write(V,val) ::
   reset V.unsub;
   V.val = val;
```

Retrieving values returned by reads

When the application calls `iso_read()`, it specifies a local variable into which the mLayer should write the value returned by the read, but it cannot read that local variable immediately since the call to `iso_read()` returns immediately without waiting for the response to be returned. To retrieve the value returned, the application calls `iso_retrieve()` with the local variable as the argument. The call to `iso_retrieve()` is blocking, that is, it returns only after the value for the corresponding `iso_read()` has returned. After an `iso_retrieve()` call, the application can read the specified local variable. The value of the local variable will remain valid until the application reuses it, either by writing to it or by using it as the local variable argument to another `iso_read()` call.

3.4.1. Send path

When the application issues an Isotach SMM access, iso_send looks up the variable accessed in `shmem_map` to determine the number of sRefs it will generate in executing the access and the location of the copies of the variable. A read (`iso_read()`) results in a single sRef on the nearest copy. A *change* (`iso_write()`, `iso_assign()`, or `iso_sched()`) results in one sRef per copy (i.e. `copy_count` sRefs). SRefs on the local copy are self-refs. SRefs on remote copies are net-refs.

Iso_send builds each sRef required by the call. It builds self-refs in `self_ref_buf` and net-refs in `iso_send_buf` (the format is shown in Appendix B). If `iso_send_buf` or `self_ref_buf` do not have enough room for all the sRef that will be generated by the call, iso_send returns a failure code.

Iso_send otherwise treats SMM accesses in the same way as MBM packets. In particular, for each isochron, it maintains a count the number of `self_refs` in the current isochron and, at the end of the isochron, calls `post_isochron()`, to allow the IOM to track the number of self-refs that are to be executed with the isochron marker.

Iso_flow approves sRefs for shipping using the same rule as iso-pointers, but does not take any of the special actions specific to iso-pointers (it does not approve an ordered packet for shipping since there is no corresponding ordered packet in the case of an sRef.)

SRefs follow an identical to that followed by iso-pointers through iso_shipping and the NIU.

3.4.2. Receive path

The receive path for sRefs is the same as for iso-pointers from the NIU through iso_receiving.

The IOM maintains an array of buckets, one for each timestamp. If the pointer passed in a call to `bucketize()` refers to an sRef, the IOM enqueues in the bucket specified by the packet's timestamp field a `packet-core-3` structure containing the subtype, the source field, and shadder and operand fields. When emptying a bucket (upon the arrival of the EOP marker for the bucket), the IOM enqueues sRefs and isochron-slots on `shmem_buf`. Each isochron-slot on `shmem_buf` represents `self_count` self-refs that the shmem finds at the head of the `self_ref_buf`.

The shmem executes sRefs in the order in which they appear in `shmem_buf`. The pseudo-code for each sRef was shown above.

After executing a net-ref, shmem deletes it by calling `delete_iso_packet()` exported by iso_flow. After executing a self-ref, shmem deletes it by incrementing `self_ref_h`.

3.4.3. Read responses

When the shmem executes a read sRef to a substantiated, variable it sends a read response packet to the read's sender, if the sender is remote. When it executes an assign for which there were reads pending, it sends a read response packet for each remotely issued read. A read response packet is a noniso packet of subtype `read_response`. The response is sent as a noniso packet because it does not need the guarantees Isotach provides. (The value to be returned is determined by the time the timestamp of the read and is not changed by the order in which read responses are returned.) Each read response identifies the local (at the host that issued the read) variable to which the value returned should be written.

The receiving module handles a read response by calling `handle_read_response()`, a function exported by the iso_retrieve module. Iso_retrieve writes the returned value in the local variable specified in the receive response packet and sets the state of the local variable to valid. When the application calls `iso_retrieve()` with local variable `lvar` as the argument, the call returns immediately if `lvar` is valid. Otherwise, iso_retrieve stays in a polling loop until it becomes valid.

3.5. Isotach Group Communication

Isotach Group Communication consists of Isotach barriers and signals. These barriers and signals are subject to the ordering guarantees that Isotach enforces, just as any other Isotach message. This section will give a description of how the Isotach barrier mechanism and signal mechanism work. For more details about the actual iso_barrier and iso_signal modules, please see the sections later in this document.

The mLayer will grant the application permission to use a GC resource only if the mLayer has not been configured to use the resource itself. The application can register and release (clear) GC resources dynamically.

3.5.1. Isotach Barriers

The GC interface supports two types of barriers: `STRONG` and `WEAK`. An execution of a weak barrier can complete only after all processes that registered the barrier have participated in the execution of the barrier. A strong barrier makes the additional guarantee that all Isotach messages issued by a participating process prior to participation are received at a given host before the barrier completes at that host. In other words, after completion of a strong barrier at host `h`, host `h` will not receive any Isotach messages issued before the sender participated in the barrier.

An Isotach barrier is initiated through a call to iso_barrier(). However, before an iso-barrier can be sent, the barrier must be registered to the application and in the proper state. The following notes describe the mechanism used for determining state information about Isotach barriers.

A barrier can be in any of five states and can be registered to the application (+) or unclaimed (-). If the barrier is unclaimed (a - barrier), iso_barrier participates in a - barrier to make the barrier usable by other hosts. Thus, when a - barrier completes, iso_barrier promptly returns a bs-marker. This activity complicates barrier registration because iso_barrier will typically not be holding a barrier credit when it executes the `iso_register_barrier()` call.

Initially, iso_barrier sets the barrier state to `SEND` to prompt sending the initial bs-marker. Subsequent state transitions are shown in Figure 3. A "+" before the state name indicates that the state can only occur when the application has registered the barrier. A "+/-" indicates ownership may either be + (application) or - (unclaimed). The notes to the right of state diagram give the condition enabling the transition and [in brackets] any action taken in making the transition.

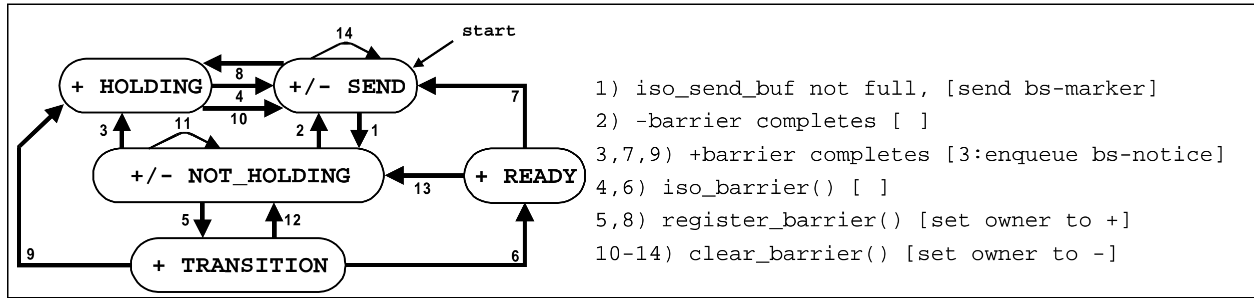


Figure 3 - Isotach Barrier States and Transitions

The five barrier states are:

> `NOT-HOLDING`: iso_barrier does not hold a barrier credit. It has sent a bs-marker but the barrier has not completed. Ownership may be either + or - (application or unclaimed).
>
> `SEND`: iso_barrier is ready to send a bs-marker. It holds a barrier credit but has not sent out the next bs-marker. Ownership may be either + or -.
>
> `HOLDING`: iso_barrier holds a barrier credit but cannot send a bs-marker until the application makes an `iso_barrier()` call. Ownership is +.
>
> `TRANSITION`: A transitional state in registering the barrier. Iso_barrier_man does not hold a barrier credit and the application has registered the barrier recently (since the last time iso_barrier sent a bs-marker). In this state, iso_barrier will accept an `iso_barrier()` call. Normally, to enforce the rule that the application can start a barrier only if the last instance of the barrier is complete, it accepts an `iso_barrier()` call only in `HOLDING` state. When it first registers for a barrier, the host does not know and is not informed of the completion of the previous barrier (a -barrier, i.e., a barrier started when ownership was -).
>
> `READY`: A second transitional state in registering a barrier. The application has registered a barrier and issued an `iso_barrier()` call while iso_barrier is awaiting completion of the previous - barrier.

Barriers can also be owned by the mLayer. A barrier claimed by the mLayer can be in one of two states: `NOT_HOLDING` or `SEND`. Only transitions 1 and 2 apply. The `complete_barrier()` call occurs after some triggering event that depends on the use the mLayer makes of the barrier: e.g., for Isotach flow control, the call occurs when the mLayer has delivered (or removed from the receive buffer) all Isotach packets received before the barrier and for tolerating lost messages, when the mLayer has ACK'd all previous messages. For now, the mLayer treats its barrier channel in the same way as an unclaimed barrier (except it does not allow the application to register it). NOTE: The implementation of mLayer iso-barriers is incomplete, which means that the mLayer cannot initiate its own barriers. Only the application can initiate barriers.

3.5.2. Isotach Signals

Sending a signal propagates a single bit of information to all application processes that have registered that signal. All processes receive the signal at the same logical time. Furthermore, all Isotach messages a process sends before sending a signal will be received before that signal is received. If multiple processes send a signal on the same channel concurrently (so that the signal arrives in the same epoch), only one signal will be received.

3.5.3. Send Path

Isotach barriers and signals traverse the same send path. Upon a call to either `iso_barrier()` or `iso_signal()` a barrier or signal (respectively) is created in the `iso_send_buf` as a packet with subtype `bs_marker`. Flow control automatically clears any `bs_marker` packets. These packets are not subject to flow control themselves, but they need to be FIFO with respect to all other Isotach traffic. Therefore, an Isotach MBM or SMM packet blocked on send credits in front of a `bs_marker` will hold up the `bs_marker` from being sent out.

Once iso_flow has cleared the `bs_marker`, iso_shipping copies the packet to the NIU's `iso_send_buf` and the NIU proceeds to send out the packet.

3.5.4. Receive Path

Isotach barriers and signals traverse the same receive path. A barrier/signal is received by the LANai in an EOP Marker. The EOP Marker has a `bits` field that contains the barrier/signal bits that have been set. The EOP Marker is transferred up to the host and placed onto the `iso_receive_buf` for iso_receiving to process. Iso_receiving passes the EOP Marker to the IOM module, which calls functions exported by the iso_barrier and iso_signal modules to inform them that barrier/signal bits have arrived. Afterwards, IOM passes the barrier/signal bits to the application by enqueueing a `bs_notice` (Note: `bs_notice` and `bs_marker` share the same subtype in this implementation) on the `iso_delivery_q`.

3.6. Flow Control

Flow control is a major task in the mLayer. Each major packet class (noniso, ordered, and Isotach) has its own receive buffers. The noniso class uses specific sender-based flow control, in which each node manages its receive buffers at each remote node. When it sends a packet, the node specifies the address of the slot into which the receiver should write the packet. The Isotach protocol uses non-specific flow control. Each node tracks the number of send credits it has at each remote node, but it does not specify the slot into which its messages are to be written when received. Flow control for ordered packets is based on Isotach flow control in a way that will be explained below.

3.6.1. Non-Isotach Flow Control

The noniso protocol uses specific sender-based flow control. Each node manages its remote receive buffers as though they were local (except that it gets delayed reports of deletions).

Initially, each node allocates to each other node a receive buffer for noniso packets and it gives the node the beginning address and slot count of its receive buffer in an initial sync packet.

Each node tracks the head and tail pointers into its remote receive buffers (in `remote_h[]` and `remote_t[]`, respectively). When node `i` sends a message to node `j`, `i`'s flow module holds the packet (by not incrementing `send_flow_ptr`) if the remote buffer is full. Otherwise, flow assigns the packet to the tail slot of the remote buffer (by writing `remote_t[j]` into the packet's DMA base field and then incrementing `remote_t[j]`) and approves the packet for shipping (by incrementing `send_flow_ptr`).

Slots become available as node `j` delivers node `i`'s messages to the application, deleting packets from the receive buffer allocated to `i` at `j`. After it delivers a packet from `i`, `j`'s delivery module informs the flow module (through a call to `delete_packet()`) and flow increments `receive_h[i]`. When `j` sends `i` a packet, `j`'s flow module writes `receive_h[i]` into the packet's credit info field. To cover the case in which traffic is sparse in one direction, `j`'s flow module can send node `i` an explicit *send credit packet* to update `i`'s view of `remote_h[j]`.

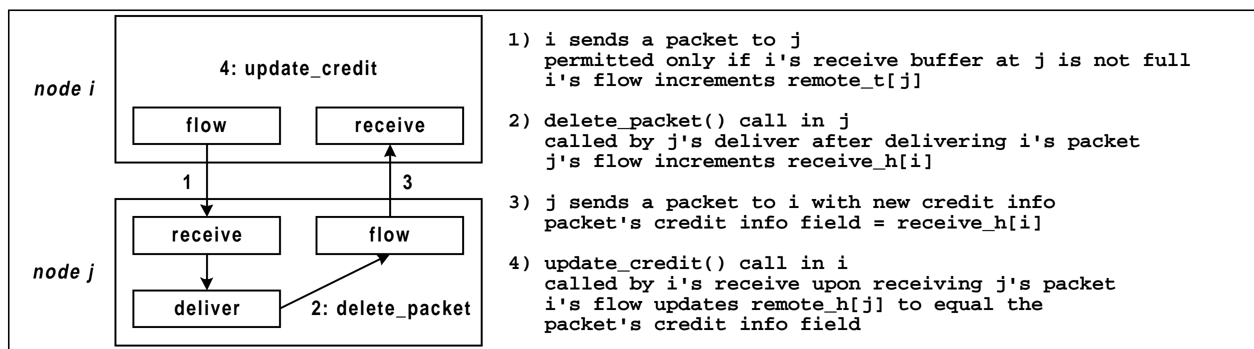


Figure 4 - Flow Control for Non-Isotach Packets

3.6.2. Ordered Flow Control

Flow control for ordered packets is based on Isotach flow control. For each Isotach packet that node `i` can send to node `j`, `j` allocates to `i` a slot in `j`'s receive buffer for ordered packets.[1] Thus if node `i` can send an iso-pointer (see overview of Isotach protocol) to node `j`, it can send the corresponding ordered packet.

Each node tracks the tail pointer for each of its remote ordered packet receive buffers, but not the head pointer (since it relies on Isotach flow control to ensure that the assigned slot will be available when the packet is sent). When node `i` sends an ordered packet to node `j`, `i`'s iso_flow module assigns the packet to the tail slot of its ordered receive buffer at `j` (by writing `ord_remote_t[j]` into the packet's DMA base field and incrementing `ord_remote_t[j]`).

[1] In the current implementation of the Isotach mLayer, the function that allocates pinned memory to each host does not differentiate between Isotach SMM and MBM interfaces. Therefore, even if Isotach SMM were selected (which requires no ordered packets), space will still be allocated for ordered packets.

When i’s iso_flow module approves the iso-pointer for shipping it also approves the corresponding ordered packet.

Piggybacking ordered flow control on Isotach flow control in this way is possible only because of the point-to-point FIFO assumption for messages. Although ordered packets may be consumed in an order that differs from the order in which they are received, ordered packets from a *given* sender are consumed in the order in which they are received, allowing ordered receive buffers to be treated as queues.

3.6.3. Isotach Flow Control

Flow control over Isotach packets uses a non-specific sender-based flow control approach. The sender tracks credits, not specific slots in the receive buffer. However, the non-specific approach can be made to look very much like the specific approach to take advantage of the existing mechanisms for specific sender-based flow control.

Isotach flow control is illustrated in Figure 5. Instead of maintaining an array of pointers into remote receive queues, each node tracks the number of its Isotach credits used at each remote node (in `my_credits_used[]`) and freed at each remote node (in `my_credits_freed[]`). Each counter has a range determined by the number of slots for Isotach packets the node has been assigned at the remote node. The `my_credits_used[]` and `my_credits_freed[]` counters are analogous to the `remote_t[]` and `remote_h[]` pointers, respectively. Thus a sender is out of credits at node `j` if `my_credits_used[j]+1 = my_credits_freed[j] mod` the counter range. Node `i` regains credits at `j` when `j`’s iso_delivery module calls `iso_delete_packet()`, after delivering an Isotach packet from `i`. In executing the call, `j`’s iso_flow module increments `your_credits_freed[i]`, and the new value of this counter gets sent to `i` in the credit info field of the next packet `j` sends to `i`. Node `i`’s iso_receiving calls `iso_update_credit()` when it receives the packet, resulting in the updating of `my_credits_freed[]`.

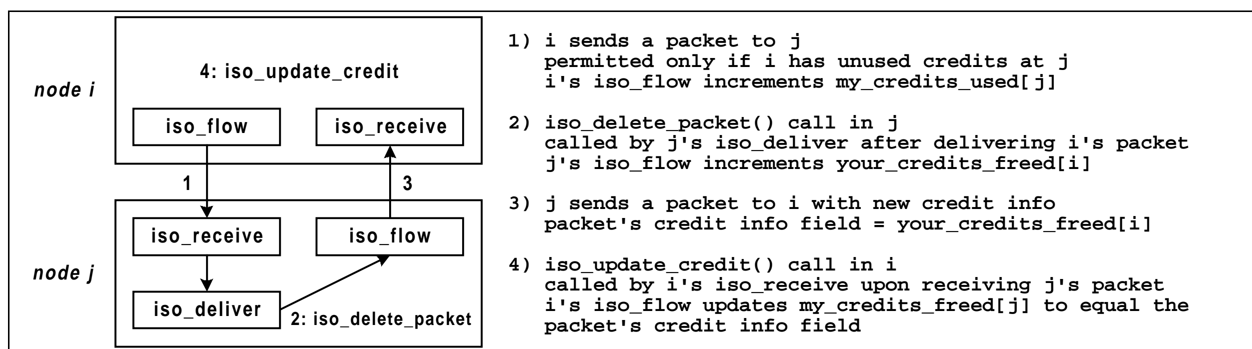


Figure 5 - Flow Control for Isotach Packets

Flow control for Isotach packets protects `iso_receive_buf` from overflow. Unless each bucket in the IOM is sized to hold all the Isotach packets in every node’s allocation, these buckets are protected only probabilistically. An alternative form of flow control based on host level time

(see ~ccw/group/flowcontrol.fm) would protect the buckets themselves and not require sending credit information.

Markers (isochron and EOP) complicate flow control because they are sent by the SIU outside of the constraints of flow control. Currently, a slack in the buffer is introduced to allow for these packets. The amount of slack allocated is a tunable system parameter.

3.7. Network Interface Unit Send/Receive Path

The send/receive model is designed to prevent Myrinet's hardware flow control from being triggered and to keep the DMA engines on the LANai as busy as possible. Hardware flow control causes incoming packets to be truncated and corrupted quickly. Thus, the design is heavily biased towards receiving, to prevent packets from sitting on the wire. To ensure that there is space to receive these packets, the toHost DMA engine on the LANai is continually kept busy transferring packets up to the host.

3.7.1. Sending

The LANai is capable of placing one packet onto the network at a time. When control is transferred to the send routine, if there are packets to send, it will send one noniso packet and one Isotach packet and then return control to the main loop in the LCP. Furthermore, while the packet is being transferred to the wire, the LANai attempts to keep receiving, thus helping to prevent hardware flow control from being triggered. Due to limitations of the LANai, and the fact that routes may not begin on a word boundary, the routing flits need to be sent out independently of the rest of the packet. Unless the number of routing flits can be fixed at compile time (i.e. every node is always x hops away), it is impossible to put the route and the packet on the wire in a single transfer.

3.7.2. Non-Isotach Receiving

As noniso packets are read in from the network, they are enqueued in a receive buffer on the LANai. The LANai is aware that the packet being received is noniso by reading in the first word of the packet and looking at the packet type field. Then, if the toHost DMA engine is free, the head packet will be DMA'd up to the host. If not, it will be sent the next time the DMA engine is free. With this system, noniso packets can continually be received from the network without having to wait for the previous packet to be sent to the host. If the receive buffer on the LANai is full, a toHost DMA will be forced, thus freeing up a slot for the incoming packet. With a large enough receive buffer, this situation is rare, and the corresponding delay will not result in dropped packets.

3.7.3. Isotach Receiving

Isotach packets are read into a special receive buffer on the LANai. They are accumulated in this buffer until an EOP marker is received, or until the buffer is full. At this point, the receive buffer is DMA'd up to the Isotach receive buffer on the host. To ensure that Isotach packets can still be received during this transfer, there are actually two separate Isotach receive buffers on the LANai. This allows a pipelining effect, in that while one buffer is being transferred to the host, Isotach

packets can be received from the network and stored in the second buffer. Then when the second buffer is being transferred, incoming packets can be stored in the first.

When transferring an Isotach receive buffer up to the host, there is the problem of queue wraparound. This occurs when the entire buffer on the LANai cannot fit in the remaining space at the end of the Isotach buffer on the host. Thus, before performing transfer n, the LANai must calculate if the next transfer ($n + 1$) will fit in the space remaining after transfer n. It knows if there is enough space for transfer ($n + 1$) if a full Isotach receive buffer would fit in the host's Isotach buffer. *If* transfer $n + 1$ would not fit, then the LANai appends a special "stop" packet to the end of transfer n, begins the transfer, and then sets the tail pointer into the host's receive buffer back to the beginning. The "stop" packet informs the host that the next packet will appear at the beginning of the buffer so that it can reset its head pointer.

3.7.4. Managing the DMA engine

In order to satisfy the constraints of continually receiving packets from the network and keeping the toHost DMA engine as busy as possible, it is necessary for these operations to proceed in parallel. Thus, only in extreme circumstances will the LANai stop receiving and wait for a toHost DMA transfer to finish. This is accomplished by keeping track of the state of the DMA engine, and periodically checking its status.

The DMA engine initially begins in an idle state. Once a transfer is initiated, its state is set to reflect what type of transfer is occurring (noniso or Isotach). Then, during the periodic checks of the DMA engine, if a transfer has finished, pointers into buffers can be updated properly depending on the state which is then set back to idle.

3.7.5. Summary of Receiving

Whereas packets are only sent out one at a time, the LANai keeps receiving packets while they are available from the network. In order to prevent receiving from completely starving sending, there is a preset limit of how many packets the LANai can receive in a single call to receive.

When a packet arrives, the LANai must first read the first word to determine what type of packet it is. Then, based on that determination, the first word is copied to the appropriate receive queue and the remainder of the packet is then read into the correct queue. One alternative is to read the entire packet off the network, examine its type, and then copy the entire packet to the appropriate queue. It is believed that this double copy is more costly than reading in the packet in two steps.

Once the packet type has been determined, the correct receive path is followed. The receive algorithm can be summarized as follows:

```
While there is something to receive from the network:
   Check the status of the from-net DMA engine;
   Read the first 2 bytes;

      If it is a noniso packet:
         If the noniso receive buffer is full:
            Force a noniso DMA to free up a slot;
         Read the rest of the packet into the noniso receive buffer;
         If the to-host DMA engine is idle:
            Begin transferring a noniso packet to the host;

      Else it is an Isotach packet
         Read the rest of the packet into the current Isotach receive
         buffer;
         If the packet was an EOP marker or the buffer is full:
            Wait for the to-host DMA engine to be idle;
            Begin transferring the current buffer to the host;
            Switch to the other buffer for receiving off the network;

   Check the status of the from-net DMA engine;
      If there are packets in the noniso receive buffer:
         If the to-host DMA engine is idle:
            Begin transferring a noniso packet to the host;
End
```

4. Messaging Layer Shared Data Structure

This section describes shared data in the mLayer. Shared data is data that is accessed by more than one module. The description of shared data is divided into two sections: shared data in the host mLayer, and shared data that makes up the host/NIU interface.

Most of the shared data structures are packet queues and buffers. The shared queues and buffers support the strategy of passing pointers to packets in place of copying packets. For each queue the following information is given: location; the modules that access the queue, count (number of elements), element type, element size; names of pointers into the queue, and for each pointer the name of the module that updates the pointer and the interpretation of the pointer (e.g., address, slot number, offset from a base location).

4.1. Host mLayer Shared Data

Packet queues in the host mLayer are shared to facilitate the strategy of passing pointers to packets instead of copying the packets themselves.

`send_buf`

The send buffer on the host. The host mLayer copies messages into the `send_buf` before transferring them to the NIU.

location - host, ordinary (nonpinned) memory

exporting module - shipping

accessing module. - send, flow, shipping, and the shmem

count - 1024. On overflow, send calls by the application fail.

element type - Each element is a slot for a packet. Each packet is of type noniso (see Appendix B).

element size - size of a noniso packet

associated pointers - (See Figure 6)

`send_t`: updated by send and the shmem after writing a new packet into the buffer; points to the next slot into which send should copy or construct a packet.

`send_flow_ptr`: updated by flow after assigning the current packet a slot in the receive buffer; points to the next packet that flow should consider

`send_h`: updated by shipping after transferring the current packet to the NIU; points to the next packet in `send_buf` that shipping should transfer to the NIU; read by send and shmem.

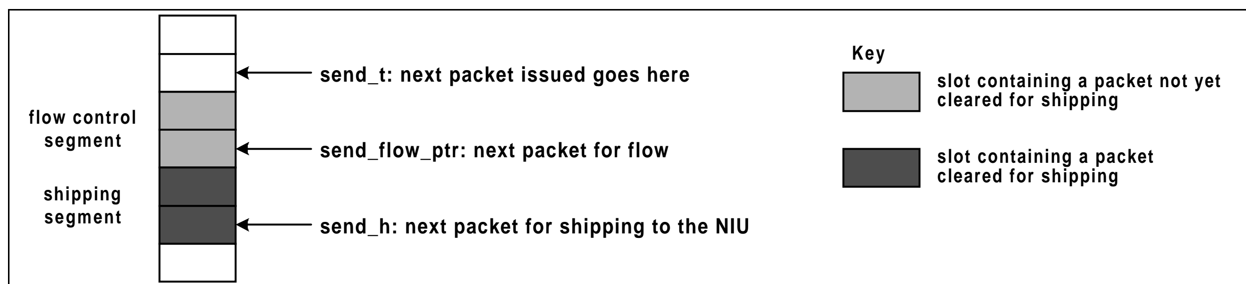


Figure 6 - send_buf Pointers

ord_send_buf

A send buffer for ordered messages used by the Isotach protocol. This buffer is the same as `send_buf` except:

- It contains messages of subtype `ordered` instead of `noniso_mbm`.
- It is accessed by iso_send and iso_flow instead of send and flow.

associated pointers

`ord_send_t`: updated by `iso_send` after writing a new packet into the buffer; points to the next slot into which `iso_send` should copy or construct an ordered packet.

`ord_flow_ptr`: updated by iso_flow after assigning the current packet a slot in the receive buffer; points to the next ordered packet that iso_flow should consider

`ord_send_h`: updated by shipping after transferring the current packet to the NIU; points to the next packet in `ord_send_buf` that shipping should transfer to the NIU

iso_send_buf

A send buffer for Isotach packets used by the Isotach protocol.

location - host, ordinary (nonpinned) memory

exporting module - iso_shipping

accessing modules - iso_send, iso_barrier, signal and iso_shipping

count - 1024 On overflow, send calls by the application fail. The application must retry the send.

element type - `iso_sendframe`

element size - size of an `iso_sendframe`

associated pointers

`iso_send_t`: updated by iso_send after writing a new packet into the buffer; points to the next slot into which iso_send should copy or construct an Isotach packet.

`iso_flow_ptr`: updated by iso_flow after approving an Isotach packet for shipping; points to the next Isotach packet that iso_flow should consider

`iso_send_h`: updated by iso_shipping after transferring the current packet to the NIU; points to the next packet in `iso_send_buf` that iso_shipping should transfer to the NIU

`receive_q`

An array of *n* receive buffers, one for each sender. Although the LANai DMA's packets into these buffers, this space is visible as a buffer only to local and remote flow modules. It is listed as a shared data structure because modules pass packets through it even though they do not see it as a buffer.

location - host's pinned memory.
exporting module - flow
accessing modules - Receiving, delivery and flow. Flow is the only local module to which the buffer is visible as an array of buffers. (The flow module at each remote node also sees its part of flow as a buffer.)
count - determined based on packet size and number of hosts. Cannot overflow due to sender-based flow control. Size of each buffer determines the number of packets each sender can have outstanding to each receiver.
element type - Each element is a packet of type noniso (see Appendix B).
element size - size of a noniso packet
associated pointers

`receive_t[]:` doesn't exist locally. The tail pointer into a receive buffer is maintained at the node to which that receive buffer has been allocated.

`receive_h[]:` local to flow.

`ord_receive_buf`

The receive buffer for ordered noniso messages sent by the Isotach protocol. Similar to `receive_buf`. An array of n receive buffers, one for each sender.
associated pointers

`ord_receive_t[n]:` incremented by receiving to register an ordered packet's arrival and read by iso_delivery. The pointers are physical address pointers.

`ord_receive_h[]:` doesn't exist.

`hit_buf`

Buffer for ordered self-packets (see Glossary).
location - host's ordinary (nonpinned) memory. The buffer's pointers are virtual memory addresses.
exporting module - iso_send
accessing modules - iso_send, iso_delivery
count - 512.
element type - Each element is a packet of subtype ordered (see App. B).
element size - size of a noniso packet
associated pointers

`hit_t:` incremented by iso_send after it enqueues a self-packet.

`hit_h:` incremented by iso_delivery to delete a delivered self-packet.

self_ref_buf (SMM)

Buffer for self-refs (see Glossary).

location - host's ordinary (nonpinned) memory. Might be an overlay for hit_buf since only one (hit_buf or self_ref_buf) will be used.

exporting module - iso_send

accessing modules - iso_send, shmem

count - start with 256.

element type - packet-core-3 structure

element size. 3 words

associated pointers

self_ref_t: incremented by iso_send after it enqueues a self-ref.

self_ref_h: incremented by shmem after it executes a self-ref.

delivery_q

A queue containing pointers to received packets.

location - host's ordinary (nonpinned) memory.

exporting module- delivery

accessing modules- receiving, delivery.

count - On overflow, the program crashes. Since the consequences of overflow are bad, the element size is small, and the type of memory required (nonpinned memory) is cheap, the array is big enough to hold everything that can be sent.

element type - pointer to a received message.

element size - word

associated pointers

delivery_t: incremented by receiving after enqueuing a new pointer in the queue

delivery_h: incremented by delivery after passing a packet to the application.

iso_delivery_q

A queue of packet-core-2 structures (see Appendix B) in the order in which the corresponding Isotach messages or bs-notices should be delivered (Isotach logical time order).

location - host's ordinary (nonpinned) memory.

exporting module - iso_delivery.

accessing modules - IOM

count - 16384. On overflow, the program crashes. Since the consequences of overflow are bad, the element size is small and the type of memory required (host's nonpinned memory) is cheap, the queue is quite large.

element type - Each element is a packet-core-2 structure of subtype iso_pointer or bs_notice.

element size - 2 words

associated pointers

iso_delivery_t: incremented by iso_receiving after enqueuing a new descriptor in the queue

iso_delivery_h: incremented by iso_delivery after passing a packet to the application.

4.2. Host/NIU mLayer Interface Shared Data

The host mLayer and NIU mLayer must communicate in order to transfer outgoing messages from the host to the on-board memory in the NIU (the SRAM) and incoming messages from the SRAM to pinned memory in the host. This section describes the data structures supporting that communication. Most of the data structures are located in a mapped portion of the SRAM. Memory on the SRAM can be mapped to the mLayer address space so that a read (write) by the host to a mapped address reads (writes) the associated location on the NIU.

4.2.1. Send-side shared queues

On the send-side, the shipping module in the host mLayer must communicate with the LANai to transfer messages from the host to the SRAM on the NIU. The send-side interface is the `niu_send_buf` and associated pointers (all located on the NIU in mapped SRAM):

`niu_send_buf`

NIU's send buffer for noniso packets. The host mLayer copies packets from send_buf into `niu_send_buf` using PIO.
location - NIU's SRAM memory, mapped into application's address space
accessing modules - shipping
count - Start with 64. On overflow, `send_buf` in the host would start to back up because the host would not be able to transfer packets to the NIU.
element type - Each element is a packet of type noniso (see Appendix B).
element size - size of a noniso packet
associated pointers

`niu_send_t`: Points to the next slot into which the host can write a packet. Written by host; read by NIU and host. The host (shipping) increments `niu_send_t` after it PIO's a packet into the `niu_send_buf`.

`niu_send_h`: Points to the next packet to be DMA'd out onto the network. The host reads `niu_send_h` (and `niu_send_t`) to test for overflow. The LANai increments `niu_send_h` when it finishes DMA'ing a packet onto the network.

`iso_niu_send_buf`

The NIU's send buffer for Isotach packets. It is the same as `niu_send_buf` except that each element is a slot for an Isotach send frame.

4.2.2. Receive-side shared queues

The LANai must communicate with the receiving and iso_receiving modules when it DMA's messages into pinned memory. The Isotach and non-Isotach receive side paths are different.

niu_delivery_q

A queue of pointers to newly arrived noniso packets in pinned memory that have not yet been acknowledged by the host mLayer. When the LANai DMA's a packet into pinned memory it writes into `niu_delivery_q` the pointer to the physical address in pinned memory to which it DMA'd the packet. The receiving module reads `niu_delivery_q` to locate incoming packets.
location - NIU's SRAM memory, mapped into application's address space
accessing modules - receiving
count - 8192. The element size is small and the consequences of overflow are bad. On overflow, the NIU would not be able to transfer messages to the host, thus niu_receive_buf would begin to back up. If `niu_receive_buf` overflows, the network would back up. Thus, to ensure that this queue is somewhat protected by flow control, set it to the maximum number of packets that can be received by a host (determined by packet size and the length of the noniso and ordered buffers).
element type - Each element is a pointer (physical address) to a packet in pinned memory.
element size - word (address size)
associated pointers

`niu_delivery_t`: Points to the element into which the LANai will write the next pointer. Incremented by the LANai after it DMA's a new noniso packet into pinned memory. Read by receiving to test for and locate new incoming packets.

`niu_delivery_h`: Points to the next packet descriptor that receiving will process. Incremented by receiving to acknowledge packets.

iso_receive_buf

A buffer of Isotach packets DMA'd by the NIU to the host. When the LANai DMA's packets into this buffer it adjusts `iso_receive_t` accordingly. The host processes packets in this buffer in FIFO order. As it finishes processing a packet, it deletes it from the buffer by incrementing `iso_receive_h`.
location - Host's pinned memory
accessing modules - iso_receiving
count - Calculated based on how many hosts are on the network and message size.
element type - word
element size - 4 bytes
associated pointers

`iso_receive_t`: updated by the LANai after writing a batch of Isotach packets into the buffer. Points to the word into which the LANai will write the first word of the next batch. The LANai increments this pointer by the number of words in the batch after it DMA's a batch into the buffer. The host reads this pointer to determine if there are new incoming Isotach packets. (this pointer is located on the LANai).

`iso_receive_h`: updated by iso_receiving to delete packets from the buffer as they are processed. Points to the first word in the next Isotach packet that iso_receiving will process. Receiving increments this pointer by the number of words in the packet when it has finished processing the packet (it may increment the pointer once for several packets). The packet may then be overwritten by the LANai. (this pointer is located on the host)

5. Host Manager (hostman) Layer

This section describes the hostman module, as well as some issues pertaining to this module.

5.1. Host Manager (hostman) Module

Hostman performs initialization, shutdown, and housekeeping functions for the host mLayer and loads the NIU mLayer.

5.1.1. Functions Exported

`get_my_node_number()`
: API function that returns the node number of the current host.

`get_number_of_hosts()`
: API function that returns the total number of hosts in the system.

`get_max_payload()`
: API function that returns the maximum payload size of a packet for this configuration of the Isotach system.

`get_SIU_state()`
: API function that returns `TRUE` if hardware SIU support is enabled, `FALSE` otherwise.

`get_MBM_ver()`
: API function that returns the current version number of the IOM module.

`get_node_number()`
: API function that returns the node number of the host given as the argument.
: Arguments: `char* hostname` (a valid short form hostname of a machine on the Myrinet network)

`open_net()`
: API function that initializes the mLayer and performs system wide reset. Must be called before any other API functions. For more information, see the section on Program Initialization.
: Arguments: int mode (the Application programmer actually only calls this function with NO parameters. The function is changed by the preprocessor during Application compilation to give the proper parameter based upon the header files that are included. For more information, see the section on Preprocessor Tricks in Appendix F)

`try_close_net()`
: API function that attempts to shutdown the mLayer. Must be called repeatedly, until it returns `SUCCESS`. When this happens, the mLayer is completely shutdown and the application may exit safely.

`iso_poll()`
: Corresponds to the API function `poll()`. This function is called implicitly when `poll()` is called in an application that is configured to use any Isotach functionality. This should never be called directly by the application.

`noniso_poll()`
Corresponds to the API function `poll()`. This function is called implicitly when `poll()` is called in an application that contains no Isotach functionality. This should never be called directly by the application.

`get_lanai_sym()`
This function is a Myrinet function, which maps a variable name (specified as the first argument) to an address memory mapped to the LANai SRAM.

`callback()`
This is a dummy function necessary for the lanai_load_and_reset() function.

`print_out_packet()`
This function takes a packet as an argument, and prints out the relevant fields. Useful for debugging.

`print_out_iso_packet()`
Same as above function but takes an iso_packet as a parameter.

`print_out_iso_marker()`
Same as above function but takes an iso_marker as a parameter.

`subtracttime()`
Takes 2 time values and produces the difference. Useful for performance testing and timing.

`shutdown()`
This function terminates the mLayer and calls the de-initialization function for hostman. It can either be called by a module in the mLayer (during an unrecoverable error) or during normal shutdown by the signal interrupt handler. For more information, see the section on Program Termination.

5.1.2. Functions Called

module_`init()`
Each module's initialization function. Called in `isotach_init()` during `open_net()` to initialize all data structures.

module_`deinit()`
Each module's de-initialization function. Called in `isotach_deinit()` during `shutdown()` to terminate the mLayer.

module_`poll()`
Each module's `poll()` function (if it has one). Called in either `iso_poll()` or `noniso_poll()` to perform work for the module, if necessary.

`iso_send_mLayer_signal(RESET_SIGNAL)`
Called in `isotach_init()` to send out the initial reset signal, when an Isotach system is being used along with hardware SIU functionality.

`initiate_barrier()`
Called in `try_close_net()` to create a barrier so that all hosts can synchronize when shutting down. Used when Isotach functionality is not requested by the application.

`barrier_completed()`
Called in `try_close_net()` to inform the mLayer that the shutdown barrier has completed, and the application may exit. Used when Isotach functionality is not requested by the application.

iso_mLayer_barrier(SHUTDOWN_BARRIER, STRONG)

Called in try_close_net() to initiate an Isotach barrier so that all hosts can synchronize when shutting down. Used when Isotach functionality is requested by the application.

5.1.3. Internal Functions

allocate_pinned_memory()

This function takes the size of pinned memory (as reported by the Myrinet driver) and partitions pinned memory appropriately. For more information, see the section on Allocation of Pinned Memory.

Arguments: int max_length

initialize_configuration_table()

This function reads in the file network.cfg from disk, and creates the Network Status Table (NST) from it. Information about host names, routes and node numbers are read in from this file. For more information, see the section on Network Status Table and Appendix E.

synchronize()

This function synchronizes with all of the running hosts, and exchanges pinned memory information with each host. After synchronization, each host should have a completed NST. For more information, see the section on Synchronization.

isotach_init()

This function is called during open_net() and initializes each module by calling their respective init functions. If the system requests Isotach functionality, an Isotach barrier is initiated by host 0 and completed here by all hosts. For more information, see the section on Program Initialization.

isotach_deinit()

This function is called during shutdown() and de-initializes each module by calling their respective deinit functions. The LANai is also reloaded with the LANai Control Program (LCP) to prevent garbage data from being erroneously sent out on the network in the case of abnormal program termination. For more information, see the section on Program Termination.

print_network_configuration()

This prints out the network configuration for the currently running system. Called from open_net().

print_config()

This prints out the pinned memory layout. Called from open_net().

5.1.4. Exported Data Structures

SYS_TYPE

Can have the values: ISOTACH, NONISOTACH or BOTH. Describes the current state of the system.

number_of_hosts

An integer that stores the total number of hosts in the running system.

host_node_id

An integer that stores the node number of the current host.

init_stage

A variable used for during LANai/host synchronization and initialization.

ISO_PINNED_SIZE
  Stores the size of the Isotach region of pinned memory (in bytes).
ISO_RECV_SIZE
  Stores the size of the Isotach receive buffer (in bytes).
ISO_CREDITS
  Stores the total number of Isotach credits assigned to this node.
NONISO_CREDITS
  Stores the total number of Non-Isotach credits assigned to this node.
offset
  Stores the offset value to convert physical memory addresses into virtual memory addresses.

5.1.5. Internal Data Structures

LCPFILE
  The location of the LANai Control Program (LCP)
CONFIG_FILE
  The location of the network configuration file (`network.cfg`).
nst
  The Network Status Table. Stores the following information about each host: node id, host name, route, alive and pinned memory information for both remote and local buffers.
iso_base_ptr
  The virtual address of where the Isotach region begins in pinned memory.
SIU_SLEEP_TIMER
  The SIU cannot have messages IMMEDIATELY sent to it following a system reset. A brief pause (approx. 2 seconds) must elapse between finishing a hardware reset and beginning message sending.
netman_hostbase
  The physical address used by the LANai that points to the base of host's pinned memory.
hostman_hostbase
  The virtual address used by the host that points to the base of pinned memory.
maxlen
  Stores the size of pinned memory as returned by the Myrinet driver.
NIU_SYS_TYPE
  A copy of `SYS_TYPE` that exists in LANai SRAM.
netman_maxlen
  A copy of the `maxlen` variable. This variable exists on the LANai SRAM.
niu_iso_recv_base
  The base address in pinned memory of the `iso_recv_buf`.

5.2. Isotach Program Initialization

Prior to running any Isotach applications, it is necessary to ensure that the network is free of any unwanted packets. When a Myrinet driver is initialized on a Linux system, the card's SRAM contains random data, which causes the LANai to send out garbage packets on the network. These packets can disrupt the functioning of the hardware SIU's and any other connected hosts. Specifically, hosts will not synchronize properly if one of the other hosts connected to the network is sending out garbage packets. To ensure that there is no garbage on the network, the LANai SRAM must be cleared immediately after its driver is loaded. This is done by loading a 'dummy LCP' (LANai Control Program) that simply loops infinitely.

Initialization of the messaging layer begins when the application calls the API function `open_net()`. This host loads the LCP onto the LANai, data structures are initialized and the hosts synchronize. After synchronization is completed, the `isotach_init()` function is called. This function calls each of the module's initialization functions, and if the system is using either Isotach MBM or SMM performs the Isotach reset procedure. If the current host is node 0 on the network, it sends out a bs-marker with the reset bit set and both barrier bits set (to prime the Token Manager with barrier credits). All other hosts wait until the reset bs-marker is received indicating that the Isotach reset is complete.

5.3. Network Status Table

Both the network configuration and the allocation of pinned memory are stored in a data structure called the Network Status Table. This table is passed by Hostman to many of the modules' initialization routines. The table is implemented as an array of structures. Each structure contains the following information about a particular host:

- `node_id` – node number of the host as specified in the configuration file
- `host_name` – short name of the host (i.e. `bugs`)
- `route` – word containing route from yourself to that particular host
- `alive` – flag indicating that the particular host is alive
- `local_noniso` – structure containing pinned memory information for that host's Non-Isotach receive buffers on the local host
- `local_iso` – structure containing pinned memory information for that host's Ordered receive buffers on the local host
- `remote_noniso` – structure containing pinned memory information for the local host's Non-Isotach receive buffers on that particular host
- `remote_iso` – structure containing pinned memory information for the local host's Ordered receive buffers on that particular host

Note that the first three fields can be determined from the network configuration file. The alive flag is set to false for all hosts except the local host. The "local" pinned memory structures can be set to the values determined during the allocation of pinned memory. The "remote" fields require information from the other hosts in the system[2]. This information is provided during synchronization.

5.4. Allocation of Pinned Memory

Currently, the Myrinet drivers are configured to use 4MB of physical memory on the host. These pages are "pinned" so that the operating system cannot swap them out. This provides fixed physical addresses for the DMA engine on the LANai to copy data.

---

[2] Note that since pinned memory on every host in our system is configured in the same manner, these fields could be determined prior to synchronization time. However, to provide greater flexibility, each host will wait to be told where its receive buffers are located on the other hosts. On different systems, the bigphysarea pinned memory region may reside at different addresses.

As the Ironman design calls for sender specific flow control, this region of pinned memory must be divided among all of the hosts, and among Isotach and Non-Isotach traffic. Furthermore, the Isotach section must be divided into Isotach and Ordered traffic. In the design, many of the parameters are configurable at compile time. These include the size of pinned memory, the ratio specifying how it is divided between Isotach and Non-Isotach traffic, and a percentage specifying how much space to allocate for Isotach markers (Isochron and EOP), which are not protected by software flow control.

These parameters, coupled with the packet size and the number of hosts allows the system to compute the maximum size of a Non-Isotach receive buffer and an Isotach receive buffer. Please see the source code for a description of the set of linear equations that are solved. Once the sizes of the receive buffers are computed, it is a simple manner to compute the location of each hosts receive buffers in pinned memory.

Pinned memory is split at a point determined by the Iso/NonIso ratio. The first section of pinned memory is used to store Non-Isotach traffic, and is divided into a series of regions – one for each host on the network. Within that region, the first few packet slots are designated as control slots, where specific mLayer messages will be received. These include credit packets, requests for credit packets, barrier packets, etc. The remainder of the region is the Non-Isotach receive buffer for a particular host.

The second section of pinned memory is used to store Isotach and Ordered traffic. Based on the size of the region, the size of ordered packets, and the percentage of slack allocated for markers, the system can compute how much memory is allocated to Ordered traffic and how much is given to Isotach traffic. Please see the source code for a more detailed description of this calculation. The ordered region is partitioned similarly to the Non-Isotach region, except that no slots are reserved for mLayer traffic. The Isotach region is not partitioned by host, and is left as a “chunk” of memory starting and ending at particular physical addresses.

Note that a receive buffer is allocated for the local host; so that host N will actually contain receive buffers for host N. The allocation algorithm could be modified fairly simply to remedy this situation, however for debugging and sanity reasons, it has been simpler to have every host divide its pinned memory in the same manner.

5.5. Synchronization

Before hosts can begin exchanging messages, each host must know that every other host on the network is alive, and it must receive information telling it where its receive buffers are located on the other hosts. To accomplish this, Ironman provides a synchronization routine:

The local host loops, and sends out a synchronization packet to every other host on the network. This synchronization packet contains information about each hosts receive buffers on the local host. The local host waits until it has received acknowledgement packets from all of the hosts it has sent synchronization packets to. It also waits to receive synchronization packets from every other host, and sends out acknowledgement packets to any host that has sent it a synchronization packet.

Once this host has received all synchronization packets (and responded to them) from every other host, and has sent out its own synchronization packets (and gotten responses back from other hosts) it is assured that every other host is alive, and that every other host knows that it is alive.

However, in a system with more than two hosts, a subset of the hosts can finish synchronizing before the rest of the hosts. Thus, a simple barrier is initiated at the end of synchronization to ensure that every host on the network has finished synchronizing.

The packets used in synchronization are Non-Isotach packets, however they do not follow the send/receive path described in the design. Instead, outgoing packets are copied directly into a region of the LANai's SRAM. During synchronization, the LANai continually scans this region looking for packets to send out. Packets that are received by the LANai are DMA'd into pinned memory in successive regions. Thus, during synchronization, all of pinned memory is treated as a simple queue.

5.6. Putting It All Together

The following table illustrates the full initialization and synchronization process on the host and LANai:

<table>
<tr><th>Host</th><th>LANai</th></tr>
<tr><td>Read the symbol table from the LANai Control Program and map pointers on the host to variables in the LANai's SRAM.</td><td rowspan="2">Not running yet</td></tr>
<tr><td>Seed the beginning of pinned memory with a set value for a test of the DMA engine</td></tr>
<tr><td>Load the LANai with the control program</td><td rowspan="2">Spinning</td></tr>
<tr><td>Write the value of the beginning of pinned memory into the LANai's SRAM</td></tr>
<tr><td rowspan="2">Spinning</td><td>Initialize a small region of SRAM with a set value</td></tr>
<tr><td>DMA from SRAM to pinned memory</td></tr>
<tr><td>Check pinned memory to ensure that the DMA engine is working properly</td><td>Spinning</td></tr>
<tr><td>Spinning</td><td>Initialize all necessary variables in SRAM</td></tr>
<tr><td>Read in network configuration file</td><td rowspan="3">Spinning</td></tr>
<tr><td>Allocate pinned memory</td></tr>
<tr><td>Initialize network status table</td></tr>
<tr><td>Enter host synchronization routine</td><td>Enter LANai synchronization routine</td></tr>
<tr><td>Initialize all modules of the mLayer
If Isotach System, send out initial Reset Marker.</td><td>Spinning</td></tr>
<tr><td>Return to application</td><td>Enter main event loop</td></tr>
</table>

Table 1 - Initialization and Synchronization Steps

5.7. Isotach Program Termination

Normal termination of an Isotach program is done with a non-Isotach or Isotach Barrier. This is initiated through the `try_close_net()` function provided by the hostman module. Each host waits on the barrier. When the barrier is complete, all hosts call the `isotach_deinit()` function. This function calls each module's de-initialization function to free up memory, and close down the messaging layer. Finally, the LANai is reloaded with the LCP to prevent it from erroneously sending out packets on the network after application termination.

If the application needs to be terminated abnormally the application should be stopped with the CONTROL-C sequence. The messaging layer has a signal handler set, waiting to intercept this keystroke and allows the system to be exited in a slightly better state than if the process was just aborted. The signal handler calls the mLayer function `shutdown(0)`. This function simply calls the `isotach_deinit()` function. This is particularly important because abnormal application termination may cause the LCP to enter a state in which it sends out packets on the network after host level application termination. Using CONTROL-C, which implicitly calls the `shutdown()` and `isotach_deinit()` functions reloads the LCP and prevents this behavior. However, there still could be problems in an Isotach network that uses hardware SIU's. If an application is terminated abnormally, the LANai will be reloaded, but it may have been reloaded in the middle of sending a packet. This means that the host's SIU will be waiting indefinitely for the end of the packet (which will never come). In addition, if the host was in the middle of an Isochron, the SIU will be waiting indefinitely for an End of Isochron packet. In either of these cases, it is necessary to manually reset each SIU and Token Manager in the Isotach network.

One possible solution to these problems would be to allow the LCP to send out everything in its buffers before allowing the host to reload the LANai. This would involve a locking mechanism between the host and LANai and should be simple to implement. A fake EOI packet would also need to be sent out, to ensure that the SIU was not waiting for an end of Isochron when the system stopped.

6. Application Programming Interface (API) Layer

This layer offers functions visible to the application.

6.1. Non-Isotach Send (send) Module

Send provides the interface to the application for sending noniso MBM messages and does the initial processing required to send such messages.

6.1.1. Functions exported

`send_init()`
  Initialization function used by hostman. (stub function – performs no actions)
`send_deinit()`
  De-initialization function used by hostman. (stub function – performs no actions)
`send()`
  API function that performs a Non-Isotach MBM send.
  Arguments: `int target`, `void *data`, `int size`

6.1.2. Tasks

On a call to `send()`

- If `send_buf` is already full, call `poll()` and then return a failure code to the application to signal failure
- Check arguments. If either argument is invalid, call `poll()` and then return a failure code
  - `target` should be a valid receiver id. In the noniso protocol, `host_node_id` is an invalid argument.
  - `length` should be no greater than `MAX_PAYLOAD_SIZE`.
- Construct the packet in the tail slot of `send_buf`.
  - Write the receiver id into the last two bytes of the route field.
    This is a temporary use of the route field. The next module down, flow, needs the receiver id.
  - Write `noniso_mbm` into the packet subtype field.
  - Write the sender id into the sender field.
  - Write the packet length (supplied as an argument to `send()`) into the payload length field
  - Copy the message into the application payload.
- Increment `send_t`
- Call `poll()`

6.2. Isotach Send (iso_send) Module

Iso_send provides the interface to the application for sending Isotach messages/accesses. It does the initial processing required for sending Isotach messages.

6.2.1. Functions Exported

`iso_send_init()`
Initialization function used by hostman.
`iso_send_deinit()`
De-initialization function used by hostman. (stub function – performs no actions)
`iso_send()`
API function that performs an Isotach MBM send.
`iso_read()`
API function that performs an Isotach SMM Read on a shared variable. NOT IMPLEMENTED
`iso_write()`
API function that performs an Isotach SMM Write on a shared variable. NOT IMPLEMENTED
`iso_sched()`
API function that performs an Isotach SMM Sched on a shared variable. NOT IMPLEMENTED
`iso_assign()`
API function that performs an Isotach SMM Assign on a shared variable. NOT IMPLEMENTED
`iso_end()`
API function that signifies that the end of Isochron (EOI) has occurred.

6.2.2. Functions Called

`post_isochron()`
Exported by the IOM. Called at the end of each isochron.
Arguments: `mid_net_isochron`, `self_count`, `net_isochrons_sent`

6.2.3. Internal Functions

`EOI_found()`
Called at the end of each isochron. Determines whether a packet should be an end of Isochron (EOI) packet and posts the Isochron if any net messages have been sent.
`start_iso_packet()`
This function performs some tasks common to enqueuing an Isotach packet in `iso_send_buf`. It assumes that it has already been determined that there is room for the packet in the buffer.

6.2.4. Exported Data Structures

`HIT_BUF_SIZE`
The maximum number of messages that can be stored in the hit buffer. Initially set to 512.

mid_net_isochron

Set to TRUE if we are currently in the middle of sending out a network Isochron, FALSE otherwise.

hit_buf[HIT_BUF_SIZE]

This buffer stores any locally executed messages. For each self-message sent, there is an entry in this buffer.

Pointers: hit_h, hit_t

6.2.5. Internal Data Structures

EOI_decided

A Boolean set to TRUE if the EOI status (see discussion under Notes) of the item in the tail slot of iso_send_buf has been determined or if iso_send_buf is empty and to FALSE otherwise. Initially, EOI_decided is TRUE. It is set to FALSE when a packet is constructed in iso_send_buf and is set to TRUE when the EOI status of the packet is determined.

seq_con_set

A Boolean initially set to 1 and flipped each time an Isotach packet is sent. To maintain sequential consistency, the bit must flip with each packet, so that the bit is set in alternating Isotach packets. See the Isotach specification for a description of its purpose.

net_isochrons_sent

A counter (range 0-255) used to generate and track isochronal id's. Initialized to 0. Incremented for each Isochron.

self_count

The number of self-packets in the current Isochron.

shmem_map

A table read in from a configuration file that gives the copy set for each page of Isotach shared memory. NOT CURRENTLY IMPLEMENTED.

6.2.6. Tasks (Common to Both Isotach MBM and SMM)

On a call to iso_send_init()

- Initialize local and exported variables (seq_con_bit = TRUE, net_isochrons_sent = 0, EOI_decided = TRUE, self_count = 0, mid_net_isochron = FALSE)
- Initialize the head and tail pointers into the hit buffer.
- Initialize the sender, subtype and type fields for all entries of the hit buffer.

On a call to EOI_found()

- If EOI_decided is FALSE
  - Set the EOI bit (bit 3, starting with 1) in the Isotach prefix of the packet in the tail slot of iso_send_buf.
  - Increment iso_send_t
  - Set EOI_decided to TRUE
- Call post_isochron() with mid_net_isochron and self_count as arguments. If mid_net_isochron is TRUE, indicating the current Isochron is a net-Isochron, pass the

value of net_isochrons_sent as the third argument, and otherwise pass in a -1 as the third argument.

- If mid_net_isochron is TRUE
  - Increment net_isochrons_sent (mod isochron_allowance)
  - Set mid_net_isochron to FALSE
- Set self_count to 0.

On a call to start_iso_packet()

This function assumes the buffer has room for the packet.

- If EOI_decided is FALSE, the packet in the tail slot is not an EOI packet (Event 2 in Notes). Increment iso_sent_t, passing the previously enqueued Isotach packet to flow control.
- Set EOI_decided to FALSE.
- Zero out the Isotach prefix.
- If mid_net_isochron is FALSE, this is an SOI (start of Isochron) message
  - Set mid_net_isochron to TRUE
  - Write 0000b110 into the first byte of the Isotach prefix, where b = seq_con_set and flip seq_con_set
  - Write net_isochrons_sent into the next byte in the Isotach prefix (the isochron_id field)
  - If send deltas are being used (currently, they are not), write the send delta into the prefix here.

On a call to iso_end()

- Call EOI_found()
- Call poll()

6.2.7. Tasks (Specific to Isotach MBM)

On a call to iso_send()

- Check arguments. If any argument is invalid, call poll() and then return a failure code
  - length should be no greater than MAX_PAYLOAD_SIZE.
  - target should be a valid receiver id. host_node_id is a valid target.
- If target = host_node_id (self-message case)
  - If hit_buf is full, call poll() and return a failure code to the application to signal failure.
  - Construct the packet in the tail slot of hit_buf and increment hit_t:
    Write the packet length (supplied as an argument) into the payload length field
    Copy the message into the application payload.
    Ignore route field. A hit_buf packet should look like an ord_receive_buf packet.
  - Increment self_count

- Else (net-message case)
  - If `EOI_decided` is `FALSE`
    If either the `ord_send_buf` is full or the `iso_send_buf` (after incrementing the `iso_send_t` by 1) is full
      Call `poll()` and return a failure code.
  - Else If `EOI_decided` is `TRUE`
    If either the `ord_send_buf` is full or the `iso_send_buf` is full
      Call `poll()` and return a failure code.
  - Call `start_iso_packet()` to begin construction of a packet in `iso_send_buf`.
  - Finish constructing the iso-pointer in `iso_send_buf`.
    Write the receiver id into the route field
    Write `iso_pointer` into the packet subtype field
  - Construct the packet in the tail slot of `ord_send_buf` and increment `ord_send_t`. Same as in send except:
    Use `ord_send_buf` and associated pointers instead of `send_buf`
    Ignore packet type, packet subtype and sender fields — they are already initialized.
- If `last_in_isochron` is `TRUE`, call `EOI_found()`.
- Call `poll()`

6.2.8. Tasks (Specific to Isotach SMM)

On a call to `iso_read()`

- Determine whether the node has a local copy of the variable.
  - Look up `shadder` in `shmem_map` to determine the location of nearest copy.
  - If shmem_map has no entry for `shadder`, call `poll()` and return a failure code.
- If the copy is local (self-ref case)
  - If `self_ref_buf` is full, call `poll()` and return a failure code.
  - Construct the packet in `self_ref_buf`
  - Increment `self_count`
- Else (net-ref case)
  - If `iso_send_buf` is full, call `poll()` and return a failure code.
  - Call `start_iso_packet()` to begin construction of a packet in `iso_send_buf`. If the call fails, call `poll()` and return a failure code to the application.
  - Finish constructing the sRef in `iso_send_buf`.
- If `last_in_isochron` is `TRUE`, call `EOI_found()`.
- Call `poll()`

On a call to `iso_write()`, `iso_assign()` or `iso_sched()`

- Check arguments.
- Return a failure code if `iso_send_buf` and `self_ref_buf` do not have room for the packets the call will generate.
  - Look up the variable in `shmem_map`

- Return a failure code if any copy is local and self-ref buffer is full or if `iso_send_buf` does not have room for `remote_copy_count` additional packets
- If the node has a local copy of the variable (self-ref case)
  - Construct the self-ref in `self_ref_buf`
  - Increment `self_count`
- For each remote copy
  - Call `start_iso_packet()` to begin construction of the net-ref in `iso_send_buf`.
  - Finish constructing the sRef in `iso_send_buf`.
- If `last_in_isochron` is `TRUE`, call `EOI_found()`.
- Call `poll()`

Notes

- `hit_buf` and `ord_send_buf` initialization - For every slot in each buffer, initialize the packet type to `NONISO`, the `packet_subtype` to `ordered`, and the sender to `host_node_id`.
- `iso_send_buf` initialization - For every slot in the buffer, initialize the packet type to `ISO` and the sender field to `host_node_id`.
- EOI status - A packet should be marked EOI if and only if it is the last net-packet sent in its isochron. Unfortunately, deciding whether a given message is the last net-packet may require looking at subsequent messages. Consider a given net-packet m. If m is sent with the argument `last_in_isochron` set to `TRUE` (the easy case), m is an EOI packet. Otherwise, m's EOI status is undecided until the first of the following events occurs: 1) the application calls `iso_end()`(m is an EOI packet); 2) the application calls `iso_send()` to send a net-message (m is not an EOI message); or 3) the application calls `iso_send()` to send a self-packet with `last_in_isochron` set to TRUE (m is an EOI message). While the application calls `iso_send()` with `target = host_node_id` and `last_in_isochron = FALSE`, the EOI status of m cannot be decided.
- The third argument (`net_isochrons_sent`) is passed out in the call to `post_isochron()` only to support an assertion in the IOM and can be dropped if and when the assertion is removed.

6.3. Non-Isotach Barrier (barrier) Module

Noniso barriers are implemented using *n*x*n* communication, i.e., every node informs every other node that it is at the barrier. The noniso barrier is used by the mLayer in `try_close_net()` can also be used by the application.

6.3.1. Functions Exported

`barrier_init()`
    Initialization function called by hostman.
`barrier_deinit()`
    De-initialization function called by hostman.
`initiate_barrier()`

API function (also called by `try_close_net()`). No arguments. Returns 0 if the barrier could be initiated. Returns 1 (failure) if the barrier could not be initiated. The call fails if another barrier is in progress.

`barrier_completed()`

API function (also called by `try_close_net()`). No arguments. Called to determine whether the current barrier has completed. Returns 0 if the barrier is complete, 1 if there is no current barrier or if the current barrier is not complete.

`process_barrier()`

Called by receiving on receipt of a barrier message. Argument: the id of the barrier message's sender.

6.3.2. Functions Called

`ship_packet()`

Called to ship a barrier message directly to the LANai's `send_buf`. Barrier packets are not subject to flow control.

6.3.3. Internal Data Structures

`barrier_recv[n]`

Boolean array indicating whether a barrier has been received from the `i`th host. Set to `FALSE` initially and when a barrier completes. Barrier sets `barrier_recv[s]` to `TRUE` when notified of receipt of a barrier from node `s` (via a `process_barrier()` call).

`barrier_packet_base[n]`

Array that stores for each remote node the physical address to which this node should address its barrier messages. The array is initialized to `remote_noniso_base[n] + 2`.

`in_barrier`

Boolean indicating whether last barrier initiated is incomplete. A call to `initiate_barrier()` fails if `in_barrier` is `TRUE`.

6.3.4. Tasks

On a call to `barrier_init()`

- Create space for the `barrier_recv[]` and `barrier_packet_base` by malloc'ing the right amount of space.
- Set `in_barrier` to be `FALSE`
- Set the `barrier_recv[i] = FALSE` for each host, and initialize the `barrier_packet_base` array.

On a call to `barrier_deinit()`

- Free the space for `barrier_recv[]` and `barrier_packet_base`.

On a call to `initiate_barrier()`

- If `in_barrier` is `TRUE`, return a failure code.
- Set `in_barrier` to TRUE
- Set `barrier_recv[host_node_id]` to `TRUE` to represent sending a barrier message to yourself.
- Construct the skeleton of a barrier packet.
- Send barrier packets. For each remote host target
  - Set the route field to target.
  - Set the address field to `barrier_packet_base[target]`.
  - Call `ship_packet()` with the barrier packet as the argument until the call is successful.
- Return `SUCCESS`

On a call to `barrier_completed()`

- Call `poll()`
- If `in_barrier` is `FALSE`, return a failure code.
- Loop through all elements of `barrier_recv`, returning if any element is `FALSE`.
- If all elements are `TRUE`, the barrier is complete
  - Set `in_barrier` to `FALSE`
  - Set each element in `barrier_recv` to `FALSE`
  - Return `SUCCESS`

On a call to process_barrier()

The argument `s` is the id of the sender of the barrier message.

- Assert that `barrier_recv[s]` is `FALSE`.
  NOTE: There is a condition in which the above value will not be FALSE. If one node completes a barrier before another node, the node that has completed can initiate a new barrier. However, because the other node had never finished the barrier the new barrier message will be lost by the node that hadn't finished the first barrier. This situation would be rare, but could happen.
- Set `barrier_recv[s]` to `TRUE`.

6.4. Isotach Signal (iso_signal) Module

Iso_signal exports functions for registering, clearing, and sending Isotach signals. (The application receives notice of incoming signals through iso_delivery)

6.4.1. Functions Exported

`iso_signal_init()`

Initialization function called by hostman.

`iso_signal_deinit()`

De-initialization function called by hostman. (stub function – performs no actions)

`iso_register_signal()`
API function that registers a signal for use with the application.
Arguments: `int channel`

`iso_clear_signal()`
API function that clears the registration for a signal channel.
Arguments: `int channel`

`iso_send_signal()`
API function that sends a signal on a previously registered signal channel.
Arguments: `int channel`

`iso_send_mLayer_signal()`
Function to allow the mLayer to use signals for SIU hardware reset.
Arguments: `int channel`

`iso_signal_poll()`
Housekeeping function for use by hostman which hostman calls as part of a poll.

`iso_signal_notify()`
Called when a signal is received.
Arguments: `UCHAR bits` (the bits from the EOP marker when a signal is received)

6.4.2. Internal Functions

`enqueue_signals()`
Enqueue a bs-marker in `iso_send_buf`.

6.4.3. Exported Data Structures

`RESET_SIGNAL`
The signal, which is registered to be the reset signal. Should be set to 5.

`reset_count`
The number of resets that have occurred since system startup.

6.4.4. Internal Data Structures

`NUM_SIGNALS`
The total number of signals usable by the mLayer and Application. Set to 6.

`INITIAL_RESET_SIGNAL`
The first reset signal must have the reset signal bit set, and have both barrier bits set in the same packet. Therefore, this was created to do both in the signal module, instead of having to use both the barrier module and signal module together. This is set to `0x83`.

`signals[NUM_SIGNALS]`
Array giving ownership (`APPLICATION`, `MLAYER`, `UNCLAIMED`) of each signal. Initialize to [`MLAYER`, `UNCLAIMED`, `UNCLAIMED`, `UNCLAIMED`, `UNCLAIMED`, `MLAYER`], meaning the reset signal and signal 5 are claimed by the mLayer.

`signal_accumulator`
Bit vector of length 8 representing a signal/barrier field. Remembers signals to be enqueued for sending the next time `iso_send_buf` has room. The first 6 bits are signal bits, the last 2 bits are barrier bits and are always 0. Initialize to zeros.

6.4.5. Tasks

On a call to `iso_signal_init()`

- Set the `signal_accumulator` and `reset_count` to 0.

On a call to `enqueue_signals()`

- If `iso_send_buf` is full, return a failure code.
- Create a bs-marker in the tail slot of `iso_send_buf`:
  - Write `010xxxxx` into the first byte to identify the packet as a bs-marker.
  - Ignore the second byte (so will the SIU, unless a barrier bit is set).
  - Write `signal_accumulator` into the signal/barrier field.
  - Zero out `signal_accumulator`
  - Increment `iso_send_t`.

On a call to `iso_register_signal()`

The argument `c` names the signal the application is requesting.

- If channel is out of the range `[0-5]` or if `signal[c]` does not equal `UNCLAIMED`, return a failure code.
- Set `signal[c]` to `APPLICATION`.

On a call to `iso_clear_signal()`

The argument `c` names the signal the application is releasing.

- If `c` is out of the range `[0-5]` return a failure code.
- If `signal[c] = APPLICATION`, set `signal[c]` to `UNCLAIMED`.

On a call to `iso_send_signal()`

The argument `c` names the signal the application is sending.

- If `signal[c]` does not equal `APPLICATION` or if `mid_net_isochron` is `TRUE`, return a failure signal. No signal should be sent if iso_send has sent the SOI, but not the EOI, packet in an Isochron because the SIU must not receive a bs-marker while it is receiving an Isochron.
- Set bit `c` in `signal_accumulator`.
- Call `enqueue_signals()`.

On a call to `iso_send_mLayer_signal()`

- Same as `iso_send_signal()` except check that `signal[c]` equals `MLAYER`
- Also, if this is the first reset signal to go out, place `INITIAL_RESET_SIGNAL` onto the `signal_accumulator`.

On a call to iso_signal_notify()

For each signal c that is contained in the argument bits

- If signal[c] = MLAYER
  - If c = RESET
    If reset_count = 0, reset_count = 1 {hostman will be checking this variable}
    Else (this is a real reset) exit
- Else if signal[c] = APPLICATION enqueue a bs-notice for the signal in iso_delivery_q

On a call to iso_signal_poll()

- If any bit in signal_accumulator is set, call enqueue_signals().

6.5. Isotach Barrier (iso_barrier) Module

Iso_barrier exports functions for registering, clearing, and participating in Isotach barriers. (The application receives notice of barrier completion through iso_delivery).

6.5.1. Functions Exported

iso_barrier_init()
    Initialization function called by hostman.
iso_barrier_deinit()
    De-initialization function called by hostman. (stub function – performs no actions)
iso_register_barrier()
    API function that registers a barrier for use by the Application.
    Arguments: int channel, UCHAR bmode
iso_clear_barrier()
    API function that clears a previously registered barrier that the Application has used.
    Arguments: int channel
iso_barrier()
    API function that sends out a barrier on the specified barrier channel.
    Arguments: int channel, UCHAR bmode
iso_mLayer_barrier()
    This function is the same as the iso_barrer() function above, except it is used by the mLayer, and not the Application.
    Arguments: int channel, UCHAR bmode
iso_barrier_notify()
    Called by IOM when an EOP marker is received that has one or more barrier bits set. The argument is an 8 bit value where the last two bits represent barrier channels 0 and 1.
    Arguments: UCHAR bits
iso_barrier_poll()
    Housekeeping function for use by hostman which hostman calls as part of a poll.

6.5.2. Internal Functions

`enqueue_barrier()`

Enqueue a bs-marker for the specified barrier channel in `iso_send_buf`.
Argument: `int channel`

6.5.3. Exported Data Structures

`SHUTDOWN_BARRIER`

This is the default barrier to be used by the `try_close_net()` function in hostman. Set to a value of 0.

`barrier_mode`

An enumerated type which creates the constants: `WEAK, STRONG, TICK` and `MAX`.

6.5.4. Internal Data Structures

`NUM_BARRIERS`

Defines the number of barriers available to the system. Set to 2.

`NUM_BARRIER_MODES`

Defines the number of different barrier modes available to the system. Set to 4.

`MIN_BARRIER_COUNT`

Defines the minimum number of barrier ticks that must elapse for a barrier to complete.

`MAX_BARRIER_COUNT`

Defines the maximum number of barrier ticks that can elapse in a barrier.

`barrier_state`

An enumerated type which creates the constants: `HOLDING, NOT_HOLDING, SEND, TRANSITION and READY.`

`barriers[NUM_BARRIERS]`

Array of barrier resource structures. Each one contains a structure with the following fields:
`owner` (`APPLICATION, MLAYER, UNCLAIMED`)
mode (`WEAK, STRONG, TICK, MAX`)
state (`HOLDING, NOT_HOLDING, SEND, TRANSITION, READY`)

`barrier_count[NUM_BARRIER_MODES]`

Constant array of barrier counts indexed by mode internal to `enqueue_barrier()`. The array has 4 elements giving the barrier count for `WEAK`, `STRONG`, `TICK` (host level logical time), and `MAX` mode barriers. Constants:
`barrier_count[WEAK] = network_diameter+1;`
`barrier_count[STRONG] = barrier_count[WEAK]` (because the send delta is zero);
`barrier_count[TICK] = 2;`
`barrier_count[MAX] = 37.`

`network_diameter`

The maximum number of hops between any two nodes.

6.5.5. Tasks

On a call to `iso_barrier_init()`

- Check the route between each host to determine what the maximum number of hops between each host is. This is the `network_diameter`.
- Initialize the `barriers` array to the following:

```
barriers[0].owner = MLAYER;
barriers[1].owner = UNCLAIMED;
barriers[i].state = SEND;
barriers[i].mode  = MAX;
barriers[SHUTDOWN_BARRIER].state = HOLDING;
barriers[SHUTDOWN_BARRIER].mode  = STRONG;
```

- Initialize each of the `barrier_count` entries:

```
barrier_count[WEAK]   = network_diameter + 1;
barrier_count[STRONG] = barrier_count[WEAK];
barrier_count[TICK]   = MIN_BARRIER_COUNT;
barrier_count[MAX]    = MAX_BARRIER_COUNT;
```

On a call to `enqueue_barrier()`

The argument `c` names the barrier channel the bs-marker should set.

- Assert (removable) that `barrier[c].state` is `SEND`.
- If `iso_send_buf` is full, return a failure code.
- Create a bs-marker with barrier bit `c` set in the tail slot of `iso_send_buf`:
  - Write `010xxxxx` into the first byte to identify the packet as a bs-marker.
  - Set the `barrier_count` field to `barrier_count[barrier[c].mode]`.
  - Clear the third byte (the signal/barrier field).
  - Set barrier bit `c` in the barrier field.
  - Increment `iso_send_t`.
- Set `barrier[c].state` to `NOT_HOLDING`.

On a call to `iso_register_barrier()`

The argument `c` names the barrier channel the application is requesting. The argument `bmode` indicates whether the barrier is `STRONG` or `WEAK`.

- Check arguments. If `c` is out of range `[0-1]`, `barrier[c].owner` does not equal `UNCLAIMED`, or `bmode` does not equal `STRONG` or `WEAK`, return a failure code.
- Set `owner` field of `barrier[c]` to `APPLICATION` and `mode` field to `bmode`.
- Assert (removable) that state is `NOT_HOLDING` or `SEND`. If `state = NOT_HOLDING`, set `state` to `TRANSITION`. Otherwise, set state to `HOLDING`.

On a call to `iso_clear_barrier()`

The argument `c` names the barrier the application is releasing.

- If `c` is out of the range `[0-1]`, return a failure code.
- If `barrier[c].owner` is not `APPLICATION`, return normally (clearing an unclaimed barrier is a NOP).
- Set `owner` field of `barrier[c]` to `UNCLAIMED` and `mode` field to `MAX`.
- If `barrier[c].state` is `TRANSITION` or `READY`, set state to `NOT_HOLDING`. If state is `HOLDING`, set state to `SEND`.
- Else (`state NOT_HOLDING` or `SEND`), `state` does not change.
- If `barrier[c].state` is `SEND`, call `enqueue_barrier(c)`.

On a call to `iso_barrier()`

The argument `c` is the name and argument `bmode` is the type of the barrier in which the application intends to participate.

- If `c` is out of the range `[0-1]` or `bmode` does not agree with the registered mode, i.e., if it does not equal `barrier[c].bmode`, return an error code.
- If `barrier[c].owner` is not `APPLICATION`, return a failure code.
- If `mid_net_isochron` is `TRUE`, return a failure code.
- If `barrier[c].state = HOLDING`
  - Set `state` to `SEND`
  - Call `enqueue_barrier(c)`
- Else if `barrier[c].state = TRANSITION`, set `state` to `READY`.
- Else return a failure code.

On a call to `iso_barrier_notify()`

The argument `bs_field` is an 8-bit value in which the last two bits (6 and 7) represent the barrier channels. The return value is `bs_field` with any non-application barriers masked out. Note that a barrier bit should be masked out when it represents the completion of a barrier that was initiated as an `UNCLAIMED` barrier completes even if it has since been registered to the application.

For each barrier bit `c` that is set in `bs_field`, a barrier on channel `c` has completed.

- Assert (removable) that `barrier[c].state` is neither `SEND` nor `HOLDING`.
- If `barrier[c].state` is `TRANSITION` or (`barrier[c].state` is `NOT_HOLDING` and `barrier[c].owner` is `APPLICATION`)
  - If `barrier[c].state` is `NOT_HOLDING`, call `enqueue_barrier(c)`
  - Set `state` to `HOLDING`.
- Else if `barrier[c].state` is `READY` or (`barrier[c].state` is `NOT_HOLDING` and `barrier[c].owner` is not `APPLICATION`)
  - Set `state` to `SEND`
  - Call `enqueue_barrier(c)`
- If `barrier[c].owner` is not `APPLICATION` or if `barrier[c].state` is `READY` or `TRANSITION`
  - The application should not be notified of this barrier completion. Set `c` to 0 to mask it out.
- Return `bs_field`

On a call to `iso_barrier_poll()`

- If `barrier[0].state` is `SEND`, call `enqueue_barrier(0)`.
- If `barrier[1].state` is `SEND`, call `enqueue_barrier(1)`.

6.6. Isotach Retrieve (iso_retrieve) Module

The iso_retrieve module exports a function to the application that allows a process to receive the result of its previously executed read access to an Isotach shared memory variable.
NOTE: This module is not implemented yet.

6.6.1. Functions Exported

`iso_retrieve_init()`
: Initialization function called by hostman.

`iso_retrieve_deinit()`
: De-initialization function called by hostman.

`iso_retrieve()`
: API function. The application calls `iso_retrieve()` to receive the value returned by a previously issued `iso_read()`

`handle_read_response()`
: Called by the receiving module upon receiving a read response packet and by the `shmem` upon substantiating a locally issued read. The function has two arguments: `val`, the value returned, and `lvar`, a pointer to the local variable to which the value should be written.

6.6.2. Tasks

On a call to `iso_retrieve()`

- If `lvar` is valid, return the value of `lvar`.
- Otherwise call `poll()` until `lvar` is valid.

On a call to handle_read_response()

- Assert that the state of `lvar` is invalid.
- Assign the value returned by the read response to `lvar`.
- Set the state of `lvar` to valid.

6.7. Non-Isotach Delivery (deliver) Module

Deliver delivers noniso MBM messages to the application.

6.7.1. Functions Exported

`deliver_init()`
: Initialization function called by hostman

`deliver_deinit()`
: De-initialization function called by hostman

`receive()`
: API function, which returns a message to the application if there is one available.

6.7.2. Functions Called

`delete_packet()`

Exported by flow. The call will increment the head pointer for the receive buffer containing the packet, resulting in the packet's deletion.

6.7.3. Exported Data Structures

`DELIVERY_SIZE`

The size of the `delivery_q`

`delivery_q`

Messages are placed on this queue for delivery to the application. The application reads messages off this queue through calls to `receive()`.

Pointers: `delivery_h`, `delivery_t`

6.7.4. Internal Data Structures

`last_packet`

pointer to the last packet delivered.

`packet_sender`

sender id of the last or current ordered net-packet delivered.

6.7.5. Tasks

On a call to `deliver_init()`

- Determine the value of `DELIVERY_SIZE` by multiplying the `number_of_hosts` by the size of the noniso region in pinned memory.
- Allocate space for the `delivery_q`
- Initialize the head and tail pointers into the `delivery_q`
- Set `last_packet = NULL`

On a call to `deliver_deinit()`

- Free the space occupied by the `delivery_q`

On a call to `receive()`

- If the last packet delivered has not yet been deleted (`last_packet != NULL`), delete it.
  - Call `delete_packet()` with `last_packet` and `packet_sender` as the arguments
  - Set `last_packet` to NULL.
- Call `poll()`
- If `delivery_q` is empty, return a failure code.
- Assign the value of pointer at the head of `delivery_q` to last_packet (virtual address).
- Dequeue the head packet from `delivery_q`
- Return the success code and return the following to the application: (in a `noniso_mbm` structure)

- pointer to the start of the application payload
- the payload length in bytes, found by de-referencing `last_packet`
- `packet_sender`

6.8. Isotach Delivery (iso_deliver) Module

Iso_deliver delivers Isotach messages and bs-notices to the application.

6.8.1. Functions Exported

`iso_receive_init()`
  Initialization function called by hostman
`iso_receive_deinit()`
  De-initialization function called by hostman (stub function – performs no actions)
`iso_receive()`
  API function, which returns an Isotach message to the application if there is one available.

6.8.2. Functions Called

`iso_delete_packet()`
  Exported by iso_flow. Restores an Isotach credit.
  Argument: id of the packet's sender.

6.8.3. Exported Data Structures

`ISO_DELIVERY_SIZE`
  The size in bytes of the `iso_delivery_q`. Please see notes in Chapter 4 on this variable.
`iso_delivery_q`
  The Isotach delivery queue. Messages are placed onto this queue for delivery to the application. The application reads messages off this queue through calls to `iso_receive()`.
  Pointers: `iso_delivery_h`, `iso_delivery_t`
`ord_receive_t`
  An array of tail pointers into the ordered receive buffers

6.8.4. Internal Data Structures

`last_packet`
  pointer to the last ordered packet delivered.
`packet_sender`
  sender id of the last or current ordered packet delivered.

6.8.5. Tasks

On a call to `iso_receive_init()`

- Set `last_packet = NULL`
- Initialize the pointers into the `iso_delivery_q`

On a call to `iso_receive()`

- If the last packet delivered has not yet been deleted (`last_packet != NULL`), delete it:
  - If `packet_sender == host_node_id` the last packet was a self-packet
    Increment hit_h to delete the self-packet
  - Else call `iso_delete_packet()` with packet_sender as the argument. This call restores credit for the iso-pointer (and ordered packet).
  - In either case, set `last_packet` to `NULL`.
- Call `poll()`
- If `iso_delivery_q` is empty, return a failure code.
- Assert (removable) that the item at the head of `iso_delivery_q` is an iso-pointer, an isochron-slot with self-count > 0, or a bs-notice.
- On an `iso_receive()`call in which the item at the head of `iso_delivery_q` is an iso-pointer.
  - Assign `last_packet` the value from the pointer field, converted to a virtual address, and assign `packet_sender` the value from the sender field of the iso-pointer in the head slot.
  - Determine whether the ordered packet pointed to by `last_packet` has arrived and return a failure code if it has not:
    - If `ord_receive_t[s] = last_packet`, reset `last_packet` to `NULL` and return a failure code.
      Explanation: Since Isotach order, receive order, and send order are the same for packets from the same sender, `last_packet` will point to the packet at the head of `ord_receive_buf[s]`. This packet has been received unless the buffer is empty. Since `last_packet = ord_receive_h[s]`, we can test for emptiness by testing `ord_receive_t[s]` and `last_packet` for equality. (Alternatively, we can use valid/ invalid tags in the ord buffer slots.)
    - Dequeue the iso-pointer from `iso_delivery_q`
    - Create the iso_mbm message to be sent back to the application:
      `data->ISO_MBM`
      `data->info.msg.data = last_packet.data`
      `data->info.msg.length = ntohs(last_packet.payload_length)` (the ntohs is to switch the byte ordering)
      `data->info.msg.sender = packet_sender`
    - Remove the message from the `iso_delivery_q`

- On an `iso_receive()`call in which the item at the head of `iso_delivery_q` is an isochron- slot, return the item at the head of hit_buf.
  - Write `hit_h` into `last_packet`.
  - Write `host_node_id` into `packet_sender`.
  - Decrement self_count
    If the `self_count = 0`, dequeue the isochron-slot from `iso_delivery_q`
  - Create the iso_mbm message to be sent back to the application:
    ```
    data->ISO_MBM
    data->info.msg.data = last_packet.data
    data->info.msg.length = last_packet.payload_length
    data->info.msg.sender = packet_sender
    ```

- On an iso_receive()call in which the item at the head of iso_delivery_q is a bs-notice.
  - Dequeue the bs-notice from iso_delivery_q
  - Copy the signal and barrier bits from the bs_notice into the iso_mbm structure.
  - Return the success code and a iso_mbm (see Appendix B) with tag bs_notice.
  - Remove the message from the iso_delivery_q

7. Processing Layer

These modules are in the middle layer of the host mLayer and perform tasks related to flow control, group communication, and ordering.

7.1. Non-Isotach Flow Control (flow) module

Flow handles outgoing packets in FIFO (issue) order. A packet blocked for send credit will block packets issued after it. The noniso protocol is not required to block subsequently issued messages except packets sent to the same receiver as the blocked message, but FIFO handling is expected to benefit the performance/maintainability by simplifying the common case (in which packets are not blocked).

7.1.1. Functions Exported

`flow_poll()`
: Housekeeping function which hostman calls as part of a poll.

`flow_init()`
: Initialization function for use by hostman

`flow_deinit()`
: Shutdown function for use by hostman

`update_credit()`
: Called by receiving when it receives a packet and used by flow to update the head pointer to the receive buffer it manages at the specified remote node. Arguments: a node id `i` and a pointer to the head of the local node's remote receive buffer at node `i`.

`delete_packet()`
: Called by delivery after it delivers a packet to the application. Argument: a pointer to the packet to be deleted, and the id of the packet's sender. The call increments the head pointer into the specified remote receive buffer and may result in sending a credit packet.

`send_credit_packet()`
: Called by receiving when handling an explicit `credit_packet_request` and internally when an explicit credit packet needs to be sent to a node. The argument is the id of the node to which the credit packet should be sent.

7.1.2. Internal Data Structures

`remote_t[n]`
: Array that stores the tail pointers (physical address) for the noniso receive buffers allocated to the local node at remote nodes, i.e., the array stores for each remote node, the last slot allocated to a noniso MBM packet bound for that node.

`remote_h[n]`
: Array that stores the head pointers (physical address) into the noniso receive buffers allocated to the local node at remote nodes. These values determine whether a packet can be sent.

remote_q[n]
: Array that stores the pointers (physical address) to the beginning of the noniso receive buffers allocated to the local node at remote nodes. These values are used to handle resetting head and tail pointers for queue wrap around.

remote_buf[n] and remote_noniso_size
: Defines the physical address bounds for the local node's noniso receive buffer at each remote node.

receive_h[n]
: Array that stores the head pointers (physical address) into the local receive buffers of remote nodes. These values are piggybacked on packets to inform remote nodes of the state of their receive buffers at the local node. On a call to delete_packet(), flow increments the pointer for the packet's sender.

receive_q[n] and REMOTE_NONISO_SIZE
: Defines the bounds for each remote node's local noniso receive buffer. These are physical (not virtual) addresses. Needed for incrementing receive_h.

receive_last_h[n]
: Array that stores in element i the last value inserted by flow in the credit info field of a packet to remote node i. Initialize to the initial value of the head pointer for i. These values are used in determining whether to piggyback credit information and in determining when to send credit packets.

credit_packet_base[n]
: Array that stores in element i the address that should be written into the DMA base field of a send credit packet sent to node i.

credit_packet_threshold
: Threshold used in determining when to send a credit packet. When the head pointer into the local receive buffer of a remote node advances credit_packet_threshold slots beyond the point at which the location of the pointer was last communicated to the remote node, send a credit packet.

CREDIT_PACKET_THRESHOLD_PERCENTAGE
: Float (range 0 to 1). The credit_packet_threshold = number of slots in a receive buffer * CREDIT_PACKET_THRESHOLD_PERCENTAGE.

credit_packet_slot
: Template for a credit packet. All the fields that remain constant over all credit packets sent by this node should be initialized to the appropriate value: the packet type is 0x0620; the subtype field is credit; the sender is my_id; the length is 0.

send_attempts
: A counter with the range 0 to MAX_SEND_ATTEMPTS. The counter is initialized to 0, is incremented when a packet cannot be cleared for sending due to lack of send credits and it reset to 0 when: 1.) the packet is cleared for sending or 2.) an explicit credit request packet is sent. This variable is declared as static within flow_poll.

MAX_SEND_ATTEMPTS
: Number of times flow_poll should attempt to ship a packet before a credit request packet is sent. Set to 50 now.

7.1.3. Tasks

On a call to `flow_init()`

Malloc() space for all of the data structures. Use the information passed in by hostman to initialize all arrays of pointers to their correct values. Calculate `credit_packet_threshold`. Complete the pre-determined fields of `credit_packet_slot`.

On a call to `flow_deinit()`

Free all malloc'd data structures.

On a call to `send_credit_packet()`

Construct a packet in `credit_packet_slot`. Write *dest* into the last two bytes of the route field (flow will look up the route); `credit_packet_base[dest]` into the DMA base field; and `receive_h[dest]` into the credit info field. Call `ship_packet()` with a pointer to `credit_packet_slot` until the call succeeds.

On a call to `update_credit()`

Update `remote_h[]`. The arguments to `update_credit()` are a node id *node_id* and the head pointer *new_head* for the local node's remote receive buffer at node *node_id*. Write *new_head* into `remote_h[node_id]`. The assignment updates the local node's view of the space available to it in the receive buffer at *node_id*.

On a call to `delete_packet()`

The arguments to `delete_packet()` are the pointer *packet* to the packet to be deleted and *sender*, the sender id.

- Test point-to-point FIFO assertion
  Assert (removable) that *packet* points to the packet at the head of *sender*'s receive buffer. i.e., assert that `packet = receive_h[sender]` (after `receive_h[sender]` is converted to a virtual address). The assertion should hold because for any two noniso packets received from the same sender *sender*, packets should be received and thus deleted in the same order as they are sent (and thus assigned slots in `receive_buf[sender]`).
- Delete *packet* from the receive buffer allocated to sender.
  Increment `receive_h[sender]`.
- If the difference between `receive_h[sender]` and `receive_last_h[sender]` is greater than `send_credit_threshold`, call `send_credit_packet(sender)`.

On a call to `flow_poll()`

Attempt to process all packets in the flow control segment of `send_buf`, returning when the segment is empty (if `send_t = send_flow_ptr`) or when a packet cannot be sent due to lack of send credits. For each packet:

- Determine whether the packet can be sent.
  A packet can be sent to node `i` unless the receive buffer allocated to this node at `i` is full, i.e., a packet can be sent unless `remote_t[i] + 1 (mod remote_noniso_size) = remote_h[i]`.
- If the packet can be sent
  Reset `send_attempts` to 0

  Write `remote_t[i]` into the packet's DMA base field and increment `remote_t[i]`.

  Piggyback credit information if the credit information has changed. If `receive_h[i]= receive_last_h[i]`, write `null_credit` into the packet's credit info field. This special value indicates to the receiver that the head pointer to its receive queue at the sender has not changed since the last packet from this sender. Otherwise write `receive_h[i]` into both the packet's credit info field and into `receive_last_h[i]`.

  Increment `send_flow_ptr` as the final step. (shipping can now send the packet out.)
- Else (the packet cannot be sent)
  Increment `send_attempts`

  If `send_attempts` > `MAX_SEND_ATTEMPTS`

  call `send_credit_packet()` with the id of the packet's receiver as the argument

  Reset `send_attempts` to zero.

7.2. Isotach Flow Control (iso_flow) Module

Iso_flow is the flow control manager for Isotach packets. Iso_flow handles packets in `iso_send_buf` in FIFO order. A packet blocked for lack of send credits blocks subsequently issued packets.

7.2.1. Functions Exported

`iso_flow_poll()`
  Housekeeping function which hostman calls as part of a poll.

`iso_flow_init()`
  Initialization function for use by hostman

`iso_flow_deinit()`
  Shutdown function for use by hostman

`iso_update_credit()`
  Called by receiving when it receives a packet and used by flow to update the head pointer to the receive buffer it manages at the specified remote node. Arguments: a node id `node_id` and the number of Isotach send credits `credit_info` that the local node has at the remote receive `node_id` .

`iso_delete_packet()`
: Called by iso_delivery after it delivers a packet to the application and by shmem after executing a net-ref. Argument: `node_id`, the id of the packet's sender. The call frees an Isotach credit for `node_id`.

7.2.2. Internal Data Structures

`my_credits_used[n]`
: Array of counters for the Isotach credits used by the local node at remote nodes. (Analogous to `remote_t[]` in noniso.)

`my_credits_freed[n]`
: Array of counters for Isotach credits restored to the local node at remote nodes. (Analogous to `remote_h[]` in noniso.)

`my_credits[n]`
: For each remote node, the total number of Isotach credits allocated to the local node by that remote node. The range of the `my_credits_used[i]` and `my_credits_freed[i]` counters is determined by the value of `my_credits[i]`. (Analogous to `remote_q[n]` and `remote_buf_limit[n]` in noniso.)

`your_credits_freed[n]`
: Array of counters tracking the number of Isotach credits restored to remote nodes. These values are piggybacked on packets to inform remote nodes of the state of their receive buffers at the local node. On a call to `iso_delete_packet()`, iso_flow increments the counter for the packet's sender. (Analogous to `receive_h[n]` in noniso.)

`your_credits[n]`
: For each remote node, the number of Isotach credits allocated to the node by the local node. The range of `your_credits_freed[n]` is determined by the value of `your_credits[i]`. (Analogous to `receive_buf_base[n]` and `receive_buf_limit[n]` in noniso.)

`your_credits_freed_last[n]`
: Array of counters used to remember the last credit info sent to each remote node. Initialize to 0. Update when piggybacking credit info. These values are used in determining whether there is new credit info to piggyback on packets and in determining when to send credit packets. (Analogous to `receive_last_h[n]` in noniso.)

`iso_credit_packet_base[n]`
: Array that stores in element `i` the physical address that should be written into the DMA base field of any Isotach send credit packet sent to node `i`.

`iso_credit_packet_threshold`
: Threshold used in determining when to send an Isotach credit packet.

`ISO_CREDIT_PACKET_THRESHOLD_PERCENTAGE`
: Float (range 0 to 1). The `iso_credit_packet_threshold` = number of slots in a receive buffer * `ISO_CREDIT_PACKET_THRESHOLD_PERCENTAGE`.

`iso_credit_packet_slot`
: Template for an Isotach credit packet. All fields that remain constant over all Isotach credit packets sent by this node should be initialized to the appropriate value: the packet type is `0x0620`; the subtype field is `iso_credit`; the sender is `my_id`; the length is 0.

`ord_remote_q[n]` and `REMOTE_ORD_SIZE`
: Used to define the bounds for the local node's ordered packet receive buffer at each remote node. Needed for incrementing the pointers in `ord_remote_t[]`

`ord_remote_t[n]`
Array of tail pointers into the local node's ordered packet receive buffer at each remote node. Need for assigning slots to outgoing ordered packets.

7.2.3. Tasks

On a call to `iso_flow_init()`

`Malloc()` space for all of the data structures. Use the information passed in by hostman to initialize all arrays of pointers to their correct values. Calculate `iso_credit_packet_threshold`. Complete the pre-determined fields of `iso_credit_packet_slot`.

On a call to `iso_flow_deinit()`

Free all malloc'd data structures.

On a call to `iso_update_credit()`

The arguments to `iso_update_credit()` are a node id `node_id` and an integer `credit_info`, taken from the credit info field of incoming packet from node `node_id`.
Assert (removable) that `credit_info` is within the range 0 to `my_credits[credit_info]`.

`Write credit_info into my_credits_freed[node_id].`

- This assignment lets the local node send more packets to `node_id`.

On a call to `iso_delete_packet()`

The argument to `iso_delete_packet()` is a node id `node_id`, representing the sender of a delivered/executed Isotach packet.
Increment `your_credits_freed[node_id]`.

- Restores an Isotach send credit to node_id.

Send credit packets when appropriate. If the difference between `your_credits_freed[node_id]` and `your_credits_freed_last[node_id]` is greater than `iso_send_credit_threshold`, send an Isotach credit packet.

- Construct the packet in `iso_credit_packet_slot`. Write `node_id` into the last two bytes of the route field (flow will look up the route); `iso_credit_packet_base[node_id]` into the DMA base field; and `your_credits_freed[node_id]` into the credit info field. Call `ship_packet()` with a pointer to `iso_credit_packet_slot`.

On a call to `iso_flow_poll()`

Attempt to process all packets in the flow control segment of iso_send_buf, returning when out of packets (iso_send_t = iso_flow_ptr) or when a packet blocks on send credits. For each packet:

- If the packet is a bs-marker (i.e., if the packet's Isotach tag (first 3 bits) = 010), it is sent unconditionally.
  - Increment `iso_flow_ptr` to approve the packet for shipping.
- Otherwise
  - Determine whether there are send credits for the packet.
    A packet can be sent to node `i` if the local node has at least one credit at `i`. For example, unless `my_credits_used[i] + 1 (mod remote_iso_size) = my_credits_freed[i]`.
  - If the packet can be sent
    Increment `my_credits_used[i]`

    Piggyback credit information if the credit information has changed.

    If `your_credits_freed[i] = your_credits_freed_last[i]`, write `null_credit` into the packet's credit info field. This special value indicates to the receiver that its credits freed counter at the sender has not changed since the last packet from this sender. Otherwise write `your_credits_freed[i]`into both the packet's credit info field and into `your_credits_freed_last[i]`.

    If the packet's subtype is `iso_pointer`

    The corresponding ordered packet is at the head of the flow control segment of `ord_send_buf`.
    `Write ord_remote_t[i] into the DMA base field of this packet.`
    Write the same value into the iso-pointer.
    Increment `ord_remote_t[i]`
    Increment `ord_flow_ptr` to approve the corresponding ordered packet for shipping.

    In any case, increment `iso_flow_ptr` to approve the Isotach packet for shipping.

Notes.

- Although bs-markers are not subject to flow control requirements, a bs-marker should not leave the processor before previously issued Isotach messages. (Isotach messages can leapfrog over barriers/signals, but not vice versa.) Thus, markers and messages must follow the same send path.

7.3. Isotach Ordering Module (IOM)

The IOM orders messages/accesses so that they can be delivered in Isotach receive order. Iso_receiving hands the IOM a pointer to each Isotach packet received at the node in the order in which they are received. The IOM must be able to handle Isotach packets of several types: iso-pointers or sRefs (depending on mode), EOP markers, and isochron markers. The IOM stores isochron markers and iso-pointers/sRefs in buckets by TS. When the EOP marker for a bucket arrives, the packets in the bucket become executable. In MBM mode, the IOM hands executable Isotach messages to iso_delivery. In SMM mode, the IOM hands executable sRefs to the shmem. If any signal and/or barrier bits are set in the EOP marker, the IOM notifies signal and/or iso_barrier.

Currently, there are two versions of the IOM implemented: version 1 and version 2. Version 1 does not support Isotach ordering, and merely forwards messages on to the application as soon as they are received. This version does not support self-messages. Version 2 supports full Isotach ordering and self-messages. Version 2 still needs to be enhanced to support the Shared Memory Model.

7.3.1. Functions Exported

`iom_init()`

Initialization function for use by hostman

`iom_deinit()`

Shutdown function for use by hostman

`bucketize()`

Called by iso_receiving to hand the IOM an Isotach packet.

Purpose: pass a newly received Isotach packet to IOM.

Arguments: pointer *ptr*, pointing to an Isotach packet in `iso_receive_buf`, and `pkt_subtype`, a byte indicating what type of Isotach packet this is (EOP marker, Isochron Marker, Iso-pointer, or SMM).

Caveat: IOM must extract all information needed to order Isotach packets and find the corresponding ordered packets. After `bucketize()` returns, the pointer may no longer be valid.

`post_isochron()`

Called by iso_send at the end of each isochron.

Purpose: inform the IOM that the application has issued an isochron so that it can ensure that self-packets issued in the isochron or subsequent to the isochron are delivered/executed at a logical time no earlier than the TS that will be assigned to the isochron. The IOM inserts a placeholder for an isochron marker in `hit_q`.

Arguments: `net_isochron` is a Boolean indicating whether this isochron is a net-isochron; `self_count` gives the number of self-packets in the isochron; `net_isochrons_sent`, is a value in the range 0 through `isochron_allowance`-1. If this isochron is a net-isochron, this value equals the isochron id field of the SOI message for the isochron and thus will be the value in the isochron id field of the corresponding isochron marker sent by the SIU after it has assigned a TS to the isochron. The third argument is passed in only to support an assertion.

7.3.2. Functions Called

`iso_signal_notify()`
Called when any bit is set in the EOP's signal field.
`iso_barrier_notify()`
Called when any bit is set in the EOP's barrier field.

7.3.3. Internal Functions

`process_bucket_entry()`
Called as a bucket is being drained. This function examines the type of the entry and processes it accordingly.

7.3.4. Internal Data Structures

`bucket[]`
An array of `bucket_count` buckets; where each bucket is a queue of `packet-core-3` structures. The bucket array stores the core of iso-pointers, isochron markers, and sRefs pending the arrival of the EOP marker enabling their delivery/execution. At the time an EOP marker for the bucket *b* is processed by the IOM, bucket *b* will contain only one generation of packets, i.e., all the packets in the bucket belong to the same pulse and not merely to pulses that map to the same bucket due to wrap.
count - (number of entries per bucket). On overflow, print an informative error message and exit. If we find that buckets are highly variable, we may have to use link in arrays to extend buckets dynamically.
element type - Each element is a `packet-core-3`
element size - 3 words
`bucket_t[]`: an array of tail pointers, one for each bucket.
`hit_q`
Stores isochron-slots representing locally issued Isochrons.
Caveat: This organization of `hit_q` assumes that the mLayer is maintaining sequential consistency over all messages/accesses sent by the application. A more complex structure would be required to reflect partial orderings.
count - 8192. On overflow, print an informative error message and exit.
element type - Each element is a packet-core-2 of subtype `isoslot`.
element size - 2 words (size of a `packet-core-2` struct)

7.3.5. Tasks

On a call to `bucketize()`

(Version 1 of IOM -- NO ORDERING OR SELF-MESSAGES)

- If the packet is an EOP marker, check for any signal bits and pass them to `iso_signal_notify()`.
- If the packet is an Isochron marker, ignore it.
- Else assert (removable): the packet is of subtype `iso_pointer`. (Later the IOM will see packets of subtype smm as well.)

- Enqueue a `packet-core-2` structure containing the packet subtype (`iso_pointer`), sender, and pointer fields from the iso-pointer into the tail slot of `iso_delivery_q`.

(Version 2 of IOM -- ORDERING AND SELF-MESSAGES)
- If the packet is an isochron marker with timestamp b, assert that bucket[b] has not overflowed. Then copy all information out of the marker and into the tail slot of bucket[b].
- Else if the packet is an iso-pointer timestamp b, assert that bucket[b] has not overflowed. Then copy all information out of the iso-pointer and into the tail slot of bucket[b].
- Else handle the EOP marker.
    - Assert (removable) that the number of items in bucket[b], where b is the EOP marker's TS, equals the EOP marker's count field.
    - If the count is less than or equal to `sort_vector_count`, use the sort vector to process each bucket entry.
    - Else, disregard the sort vector and re-sort the bucket. For such a small number of items, it is faster to resort, rather than try to merge the two sections of the bucket. Additionally, since the number of senders is currently smaller than the number of bucket entries, the most efficient sort is to loop through the bucket for each sender s, processing every entry for s as it is encountered.
    - Once all of the entries in the bucked have been processed, examine the EOP marker for barrier and signal bits, and pass them to the appropriate modules via `iso_signal_notify()` and `iso_barrier_notify()`. Then, pass signals and barriers to the application by enqueueing a bs_notice for each at the end of `iso_delivery_q`.

On a call to internal function `process_bucket_entry()` (*Version 2*)

Examine the subtype of the bucket entry:
- If the entry is an iso-pointer, enqueue it onto the `iso_delivery_queue`.
- If the entry is an sRef, enqueue it onto `shmem_buf`.
- If the entry is an isochron marker,
    - Assert (removable) that the isochron id of the isochron-slot at the head of `hit_q` is the same as the isochron id of the isochron marker in the bucket.
    - Move the isochron-slot from the head of `hit_q` to the tail of the destination queue if there are self messages to be delivered.
    - Increment `net_isochrons_executed`.

On a call to `post_isochron()` (*Version 2*)
- If `net_isochron` is `TRUE` enqueue a `packet-core-3` structure of subtype `isochron_slot` in the tail slot of `hit_q`. Fill in the `self_count` and `isochron_id` fields from the second and third arguments passed into `post_isochron`.
- Else, if `hit_q` is empty the self-packets in the current isochron are executable
    - If the item at the tail of `iso_delivery_q` is an isochron-slot add `self_count` to the `self_count` field of the existing isochron-slot.

- Otherwise, enqueue a packet-core-2 structure of subtype `isochron_slot` at the tail of `iso_delivery_q`. Fill in `self_count` field from the argument passed into `post_isochron`. (Fields other than `subtype` and `self_count` can remain "as is")
- Else, add `self_count` to the `self_count` field of the isochron-slot at tail of `hit_q`.

Notes

- When the IOM moves an iso-pointer from a bucket onto the `iso_delivery_q`, it is moving a 3 word structure into a 2 word structure. This works because the last word of a packet-core-3 structure is pad except in the case of an sRef and the IOM never moves an sRef onto `iso_delivery_q`. (sRefs are passed to the shmem.)

7.4. Isotach Shared Memory Manager (shmem)

The shmem executes sRefs on the locally stored portion of shared memory. The sRefs are stored in Isotach logical time order on `shmem_buf`, a buffer of `packet-core-3` structures. The shmem handles the items on `shmem_buf` in order. Each item is a `packet-core-3` structure of subtype `iso_read`, `iso_assign`, `iso_write`, `iso_sched`, or `isochron_slot`. Each `isochron_slot` has a count field (`self_count`) giving the number of self-refs that the shmem should execute from the head of `self_ref_buf` to process the `isochron_slot`.

NOTE: This module is currently not implemented

7.4.1. Functions Exported

`shmem_poll()`
: Housekeeping function.

`shmem_init()`
: Initialization function for use by hostman

`shmem_deinit()`
: Shutdown function for use by hostman

7.4.2. Functions Called

`handle_read_response`
: Exported by iso_retrieve. Called by the shmem to return responses to locally issued reads on locally stored shared variables.

7.4.3. Internal Functions

The argument to each function except `send_read_response()` is `p`, a pointer to a `packet-core-3` structure of the appropriate subtype (e.g. `iso_read` for `read_remote()` or `read_local()`). The `read_remote()` and `assign()` functions return a failure code if they cannot send a required a read response (due to a full `send_buf`). The other functions always succeed.

`read()`
: Handle a read to the locally stored portion of Isotach shared memory. May fail in the case of a remotely issued read.

`write()`
: Handle a write to the locally stored portion of Isotach shared memory.

`assign()`
: Handle an assign to the locally stored portion of Isotach shared memory. May fail.

`sched()`
: Handle a sched to the locally stored portion of Isotach shared memory.

`send_read_response()`
: Enqueue a read response in `send_buf`. The arguments are the `pId` of the reader, the value to be returned, and a pointer to the local variable into which the value returned should be written.

7.4.4. Internal data structures.

- The locally stored portion of Isotach shared memory. (See current implementation)
- Pending reads. A table (see discussion in SMM overview) that allows retrieval of pending reads given the variable address and vId on which the reads are pending. Each entry identifies the variable and vId on which the read pends and gives the reader (pId of the read's sender) and the local (at the reader) variable in which the value read should be written once returned.

7.4.5. Tasks

On a call to `shmem_poll()`

Process each item in `shmem_buf` in order, stopping when the buffer is empty or if `read()` or `assign()` fails (indicating `send_buf` is full when a read response must be sent).

- If the item at the head of `shmem_buf` is of subtype `isochron_slot`
  - If `self_count` > 0 process the first access in `self_ref_buf`
    De-reference `self_ref_h` to find the type of access.
    Depending on the access type, call internal function `read()`, `write()`, `assign()`, or `sched()` with `self_ref_h` as the argument.
    If the call succeeds (it may not in the case of an assign)
      Decrement `self_count` field in the isochron_slot
      Increment `self_ref_h` to delete the self-ref just processed
  - Else (`self_count` is 0) increment `shmem_h` to delete the `isochron_slot`
- Else (the head item is an `iso_read`, `iso_write`, `iso_assign`, or `iso_sched`)
  - Call the appropriate internal function with `shmem_h` as the argument.
  - If the call succeeds (it may not in the case of `read()` or `assign()`) increment `shmem_h` to delete the sRef.

On a call to internal function `read()`

- If V (the variable accessed) is unsubstantiated, create an entry in `pending_reads`
  - Allocate the entry and record in the entry V's address, V's vId, the `lvar` field of the read, and the pId of the sender

- Else if `p = self_ref_h` (self-ref case), call `handle_read_response()` with V's value and the `lvar` field in the `iso_read` structure as arguments and return.
- Else (the source is remote) call the internal function `send_read_response()`with V's value, and the `source` and `lvar` fields of the `iso_read` structure as the arguments. Return the code that call returns (fail *iff* the call to `send_read_response()` fails).

On a call to internal function `sched()`
- Mark V (the variable accessed) unsubstantiated.
- Assign the source pId to V's value/vId field.

On a call to internal function `assign()`

- If V(the variable accessed) is unsubstantiated and V's value/vid equals the source field from the `iso_assign` structure pointed to by `p`, the assign corresponds to the last post to V. Assign the value transmitted to V and mark V as substantiated
- For each perfect match (both variable name and vId) in `pending_reads`
  - If the reader is remote
    Call the internal function `send_read_response()`
    If the call fails, return a failure code
  - Else (local reader), call `handle_read_response()`
  - De-allocate the entry in `pending_read`

On a call to internal function `write()`

- Mark V (the variable accessed) substantiated.
- Assign the value transmitted to V's value field.

On a call to internal function `send_read_response()`

The first argument is `val`, the value to be returned; the second is `reader` the pId of the process to which the read response is sent; the third is a `lvar`, a pointer to the reader's local variable in which the value returned should be written.
- If `send_buf` is full, return a failure code
- Otherwise, construct a `read_response` packet in the tail slot of `send_buf`.
  - Write `reader` into the last two bytes of the route field.
  - Write `read_response` into the packet subtype field.
  - Write `my_id` into the sender field.
  - Write 8 into the payload length field (the payload of a read response is 2 words)
  - Write the `val` into the `data` field (first word of payload) and `lvar` into the `lvar` field (second word of payload).
- Increment `send_t`

8. Network Interface Unit (NIU) Layer

The modules in the NIU interface layer send packets to the NIU and receive packets from the NIU.

8.1. Non-Isotach Shipping (shipping) Module

Shipping fills in the route field in outgoing noniso packets and transfers packets to the NIU. Shipping takes packets from `send_buf` and `ord_send_buf`.

8.1.1. Functions Exported

`shipping_init()`
  Initialization function for use by hostman
`shipping_deinit()`
  Shutdown function for use by hostman
`shipping_poll()`
  Housekeeping function which hostman calls as part of a poll.
`ship_packet()`
  Called by flow to send out credit packets and barrier and credit requests. The argument is a pointer to the packet to be shipped.

8.1.2. Internal Data Structures

`routes[n]`
  An array of words representing the route (in network byte order) from the local host to every other host on the network.

8.1.3. Tasks

On a call to `shipping_init()`

- Malloc space for routes[n] and copy the routes from the network status table.
- Map pointers for the send buffer on the LANai
- Initialize the host's send buffers
- Fill in all static information for every slot in the ordered send buffer

On a call to `shipping_deinit()`

- Free the space allocated for routes[n].

On a call to `ship_packet()`

- Return a failure code if `niu_send_buf` is full
- Look up packet's route in routes and write the route into the route field

- Move the packet into the tail slot of `niu_send_buf` (incrementing `niu_send_t`)

On a call to `shipping_poll()`

Transfer all packets in the shipping segments of `send_buf` and `ord_send_buf`, returning when both shipping segments are empty or when `niu_send_buf` is full. As two different queues are being emptied, it is important that one queue not starve the other. This is accomplished with three different loops. The first loop executes while there are packets remaining in both queues. The second loop executes while there are packets waiting in `send_buf`, and the third executes while there are packets waiting in `ord_send_buf`. Notice that only one of the final two loops will execute. Within each loop, for each packet:

- Determine whether current packet can be sent.
  - A packet can be sent unless the NIU's send buffer is full.
- If the packet can be sent
  - Look up the packet's route in routes and write the route into the route field
  - Move the packet into the tail slot of `niu_send_buf` (increment `niu_send_t` and the head pointer for the buffer from which the packet was taken).

Notes

- Shipping does not need to read the memory mapped pointer `niu_send_h` each time it checks for room in `niu_send_buf`. Shipping can read the value into a local variable and reread `niu_send_h` only when using the local yields a finding that the buffer is full.
- Similarly, shipping does not need to write the memory mapped pointer `niu_send_t` each time it sends a packet, but can instead write to a local. However, moving the pointer only once for several packets prevents the NIU from processing the packets until the pointer is moved. [Initially assume that shipping updates `niu_send_t` for each packet.]

8.2. Isotach Shipping (iso_shipping) Module

Iso_shipping performs the same functions as shipping, except that iso_shipping moves packets from `iso_send_buf` to `iso_niu_send_buf`. Each time it gets control, iso_shipping transfers the packets in the shipping segment of `iso_send_buf` unless the shipping segment is empty.

Additionally, routes are stored differently for Isotach packets. The hardware SIU expects routes to range from two to six bytes. As the noniso protocol supports up to four routing bytes, Isotach packets are currently limited to that as well. To implement this, Isotach packets contain two different route fields – one that is two bytes long, and one that is four bytes long. Depending on the length of the route (one to four bytes), these fields are populated in a specific manner. A more detailed explanation of this scheme is available in the source code.

8.3. Non-Isotach Receiving (receive) Module

Receiving performs initial receive-side processing on all types of noniso packets (`noniso_mbm`, `ordered`, `credit`, `credit_request`, `barrier`, `iso_credit`, `read_response`) received from the NIU. Receiving reads packet pointers from `niu_delivery_q`, checking the CRC and

extracting credit information from each new incoming packet. In the case of a `noniso_mbm` packet, receiving enqueues the pointer into delivery's `delivery_q`. In the case of an ordered packet, receiving increments the tail pointer to the sender's local receive buffer and drops the pointer (the pointer that is read off `niu_delivery_q`, not the tail pointer it just incremented).

8.3.1. Functions Exported

`receive_poll()`
: Housekeeping function which hostman calls as part of a poll.

`receive_init()`
: Initialization function for use by hostman

`receive_deinit()`
: Shutdown function for use by hostman

8.3.2. Functions Called

`update_credit()`
: Exported by flow. Gives flow information (the head pointer for the remove receive buffer) about this node's noniso receive buffer at the specified remote node.

`send_credit_packet()`
: Exported by flow. Called by receiving when it receives an explicit `credit_request` to send an explicit credit packet. The argument is the id of the `credit_request` packet's sender.

`process_barrier()`
: Exported by barrier. Called to when a barrier packet is received.

`handle_read_response()`
: Exported by iso_retrieve. Called to hand the information from a read_response packet to iso_retrieve.

8.3.3. Internal Data

`ord_receive_buf[n]` and `ord_receive_buf_limit[n]`
: The base and limit arrays for each remote node's local ordered receive buffer. These are physical (not virtual) addresses. Needed for incrementing `ord_receive_t[]`.

`this_packet`
: pointer (virtual address) to the current packet

8.3.4. Tasks

On a call to `receive_init()`

- Map pointers into the delivery queue on the LANai
- Allocate space for the arrays of pointers into the ordered receive buffers
- Initialize these pointers using the information contained in the network status table

On a call to `receive_deinit()`

- Free space used by the arrays of pointers into the ordered receive buffers

On a call to `receive_poll()`

Receiving processes all the packets that the NIU has DMA'd into pinned memory up to the time of the call (but not beyond). New packets are available unless `niu_delivery_h = niu_delivery_t`. Receiving reads `niu_delivery_t` when it gets control, but does not reread `niu_delivery_t` until the next time it gets control.

- For each new incoming packet
  - Obtain the virtual address of the next packet: `this_packet = niu_delivery_q [niu_delivery_h] + offset` (to calculate the virtual from the physical address).
  - Acknowledge the packet by incrementing `niu_delivery_h`. Note that since this is stored on the LANai in network byte order, it is necessary to make a copy of the head pointer, switch the byte ordering, increment it, switch the byte ordering back, and then store the value on the LANai.
  - Check `this_packet's` CRC. If it is bad, print an informative message to the user and gracefully terminate execution.
  - Extract credit information. If the credit info field has a value other than `null_credit`, pass the information to flow via a call `iso_update_credit()` for `iso_credit` packets and `update_credit()` otherwise. The arguments to the credit updating function are the packet's sender and the value of the credit info field.
  - For each `noniso_mbm` packet
    Write `this_packet` into the tail slot of `delivery_q` and increment `delivery_t`.
    If `delivery_q` is full, print an informative error message and exit.
  - For each ordered packet
    Assert (removable) that `this_packet = ord_receive_t[s]`. The assertion should hold because ordered packets issued by the same sender to the same receiver should be received in issue order.
    Increment `ord_receive_t[s]`, where `s` is the packet's sender, found by dereferencing `this_packet`.
  - For each credit or iso_credit packet
    Drop the packet. The useful information has already been extracted.
  - For each barrier packet
    call `process_barrier(s)`
  - For each credit request packet
    call `send_credit_packet(s)`
  - For each read response packet
    call `handle_read_response(this_packet)`

Notes

- Eventually receiving should acknowledge packets in batches (i.e., update `niu_delivery_h` once for a batch of instead of once for every packet) when `niu_delivery_q` is near empty.
- Upon review of the design, it appears that many potential bottlenecks occur in this module. Incrementing the head pointer is expensive (as noted above). Furthermore, as PIO reads are more expensive than writes, having to read a pointer in the LANai's SRAM for every noniso packet received appears to be costly. A potential solution is to use valid flags within the

receive buffers which the host can check at each invocation of `receive_poll()`. For a large number of potential senders, this may be expensive, however, for smaller number of hosts, it would most likely be more efficient than doing expensive reads over the PCI bus. The delivery queue would still be necessary for mLayer packets, however these packets are relatively rare. Further investigation into this matter is necessary.

8.4. Isotach Receiving (iso_receive) Module

Performs initial receive side handling of all incoming Isotach packets. Isotach packets include the following: iso-pointers, Isotach SMM packets, EOP markers, and isochron markers. Iso_receiving handles packets in `iso_receive_buf` in FIFO order, extracting credit information from each new incoming packet (except markers), and checking CRCs. Iso_receiving hands each packet to the IOM via a call to `bucketize()`.

8.4.1. Functions Exported

`iso_receiving_poll()`
Housekeeping function which hostman calls as part of a poll.
`iso_receive_init()`
Initialization function for use by hostman
`iso_receive_deinit()`
Shutdown function for use by hostman

8.4.2. Functions Called

`bucketize()`
Exported by the IOM. Call to pass an Isotach packet to the IOM.
`iso_update_credit()`
Exported by iso_flow. Gives iso_flow information (the head pointer for the remote receive buffer) about this node's Isotach receive buffer at the specified remote node.

8.4.3. Internal Data

`iso_recv_buf`
Pointer (virtual address) into the Isotach receive buffer in pinned memory.
`iso_recv_h`
Head pointer (virtual address) into the Isotach receive buffer
`iso_recv_t`
Tail pointer (physical address) into the Isotach receive buffer (stored on the LANai)
`iso_recv_start`
Pointer (physical address) into the Isotach receive buffer (stored on the LANai)
`iso_recv_size`
Size of the Isotach receive buffer (stored on the LANai)

8.4.4. Tasks

On a call to `iso_receive_init()`

- Map symbols into the LANai's SRAM
- Initialize head and tail pointers

On a call to `iso_receive_deinit()`
- Nothing to do.

On a call to `iso_receive_poll()`

- Process all the packets that the NIU has DMA'd into `iso_receive_buf` up to the time of the call (but not beyond). A packet is available unless `iso_receive_h = iso_receive_t`. Iso_receiving reads `iso_receive_t` when it gets control, but does not reread `iso_receive_t` until the next time it gets control.
- The LANai is in charge of handling queue wrap around. When it is about to reset the tail pointer, it enqueues a special "stop" packet into the Isotach receive buffer. When this stop packet is encountered, the head pointer is reset back to the beginning of the buffer.
- For each packet in `iso_receive_buf`
  - Check the CRC of the packet. If the packet is bad, print an informative error message and exit.
  - Extract returned credits. For each packet other than a marker, if the credit info field has a value other than `null_credit`, pass the information to iso_flow via a call to `iso_update_credit()`. The arguments to the credit updating function are the packet's sender and the value of the credit info field.
  - Call `bucketize()` with a pointer to the packet and the packet's subtype
  - When `bucketize()` returns, delete the packet from `iso_receive_buf` by incrementing `iso_receive_h` by the packet size.

9. Network Manager (netman) Layer

The netman layer consists of a single module that is responsible for communicating with both the host and the network. It receives packets from the host to put out on the wire, and also receives packets from the network and transfers them up to the host.

9.1. Internal Functions

`main()`
Initial function executed when the LANai Control Program starts.
`synchronize()`
Serves as an interface between the host and the network for synchronization packets.
`send_packets()`
Places packets onto the wire.
`check_dma()`
Checks to see if a toHost DMA transfer has completed. Updates all necessary pointers.
`noniso_dma()`
Attempts to initiate a DMA transfer of a noniso packet up to the host. If the toHost DMA engine is busy, simply returns.
`force_noniso_dma()`
Guarantees that a noniso packet will be transferred to the host before the function returns.
`iso_dma()`
Initiates a DMA transfer of the Isotach receive buffer on the LANai up to the host.
`receive_packets()`
Receives packets from the network.

9.2. Exported Data Structures

`niu_send_buf[]`
Buffer for outgoing noniso packets.
`niu_send_h`
Head pointer into the noniso send buffer.
`niu_send_t`
Tail pointer into the noniso send buffer.
`iso_niu_send_buf[]`
Buffer for outgoing Isotach packets.
`iso_niu_send_h`
Head pointer into the Isotach send buffer.
`iso_niu_send_t`
Tail pointer into the Isotach send buffer.
`niu_delivery_q[]`
Queue of pointers to noniso packets that have been received and transferred to the host.
`niu_delivery_h`
Head pointer into the noniso delivery queue.
`niu_delivery_t`
Tail pointer into the noniso delivery queue.

iso_recv_t

Tail pointer into the Isotach receive buffer on the host. Used by the host to determine when new Isotach packets are available.

9.3. Internal Data Structures

niu_receive_buf[]

Buffer for incoming noniso packets.

niu_receive_h

Head pointer into the noniso receive buffer.

niu_receive_t

Tail pointer into the noniso receive buffer.

niu_iso_recv_buf0[]

A buffer for incoming Isotach packets.

niu_iso_recv_buf1[]

A second buffer for incoming Isotach packets.

niu_iso_buffer[]

Contains the starting addresses of each of the Isotach receive buffers on the LANai.

niu_iso_thresh[]

Contains the high water mark for each of the Isotach receive buffers on the LANai. Used to indicate when to transfer the current Isotach receive buffer up to the host.

cur_niu_iso_buf

Index into the array of buffer addresses indicating which Isotach buffer is currently being used to receive packets.

niu_iso_tail_ptr

Tail pointer into the current Isotach receive buffer.

iso_recv_buf_thresh

High water mark for the Isotach receive buffer on the host. Used to indicate when queue wraparound is about to occur.

new_iso_recv_t

Location of the tail of the Isotach receive buffer on the host once the current Isotach DMA transfer has completed.

buffer

Used to store the first word of an incoming packet that is read in separately from the rest of the packet.

state

Indicates the state of the toHost DMA engine. Possible values are IDLE, DMA_NONISO, or DMA_ISO.

9.4. Tasks

On a call to main()

- Perform a simple test of the toHost DMA engine.
- Initialize all buffers and head/tail pointers. Calculate all threshold values.
- Call synchronize().
- Enter into the main event loop (loops forever):

- Check to see if a toHost DMA transfer has completed. If so, update all necessary pointers.
- Check to see if there are noniso packets waiting to be transferred to the host. If so, attempt to initiate a toHost DMA transfer.
- Call `send_packets()`.
- Call `receive_packets()`.

On a call to `synchronize()`

- Send out all sync packets generated by the host.
- Receive all incoming sync packets using a pipelined scheme. One slot stores the incoming packet while a second slot is being transferred up to the Host.
- Transfer received packets up to the host using all of pinned memory as a single buffer.

On a call to `send_packets()`

- If there is a noniso packet to send:
  - Send the route independently. Check to see how many bytes the route is by checking for 0x00 inside the word containing the route.
  - Transfer the remainder of the packet onto the wire.
  - While waiting for that transfer to finish, call receive_packets.
- If there is an Isotach packet to send:
  - If the packet is a Barrier/Signal marker, simply write the word onto the wire.
  - Otherwise, it is a regular Isotach packet. First, write the word containing the prefix directly onto the wire.
  - Calculate how much of the route needs to be sent out (see the code for a more detailed explanation), and write it directly onto the wire.
  - Transfer the remainder of the packet onto the wire
  - While waiting for that transfer to finish, call `receive_packets()`.

On a call to `check_dma()`

- If the current transfer has not completed, return the state of the DMA engine.
- If a noniso transfer has completed:
  - Enqueue a pointer to the packet onto the delivery queue.
  - Advance the tail pointer into the delivery queue.
  - Advance the head pointer into the noniso receive buffer
  - Set the state of the DMA engine to `IDLE`.
  - return `DMA_NONISO`.
- If an Isotach transfer has completed:
  - Set the tail pointer into the hosts Isotach receive buffer to the value calculated by iso_dma.
  - Set the state of the DMA engine to `IDLE`.
  - return `DMA_ISO`.

On a call to `noniso_dma()`

- If toHost DMA engine is busy, simply return.
- Otherwise, initiate a transfer to the host:
  - The address of the data on the LANai is simply the head of the noniso receive buffer
  - The destination address on the host is stored in the packet.
  - Check to ensure that the transfer is not outside the bounds of pinned memory
  - The size of the transfer is contained within the packet in the `pad2` field.
  - Set the state of the DMA engine to `DMA_NONISO`.

On a call to `force_noniso_dma()`

- Wait for the DMA engine to finish its current transfer.
- Call check_dma.
- If the transfer that just finished was a noniso, begin a new noniso transfer and return.
- Otherwise, the previous transfer was an Isotach transfer.
  - Initiate a noniso DMA transfer
  - Wait for the transfer to complete.
  - Begin a new noniso transfer and return.

On a call to `iso_dma()`

- If the DMA engine is busy, wait for it to finish, its current transfer and call `check_dma()`.
- Calculate where the tail pointer into the host's buffer will be after this transfer has completed.
- If the new tail pointer is greater than the previously calculated threshold
  - Write a stop packet at the tail of the LANai's Isotach receive buffer
  - Set the new value of the tail of the Host's receive buffer back to the start of that buffer
- Initiate a transfer to the host:
  - The address of the data on the LANai is the beginning of the current receive buffer
  - The destination address on the host is the tail pointer into the host's receive buffer
  - The size of the transfer is the size of the current receive buffer on the LANai
  - Set the state of the DMA engine to DMA_ISO
- Switch to the other Isotach receive buffer on the LANai to keep receiving packets while this transfer is occurring.

On a call to `receive_packets()`

While there are packets waiting on the wire and less than the preset limit of packets has been received:

- Call `check_dma()`.
- Read in the first word of the packet off the wire to determine what type it is.
- If the new packet is of type noniso:
  - If the noniso receive buffer is full, call force_noniso_dma.
  - Write the previously read in first word into the tail slot of the receive buffer.
  - DMA the remainder of the packet into the tail slot
  - Calculate the size of the packet and store it in the `pad2` field

- Increment the tail pointer into the noniso receive buffer
- Call `noniso_dma()`

- Otherwise, the new packet is of type Isotach:
  - Write the previously read in first word into the tail slot of the current Isotach receive buffer.
  - DMA the remainder of the packet into the tail slot of the buffer
  - Calculate the size of the packet
  - If the size of the packet indicates that it is an Isochron marker:
    - Logically "or" the Isochron CRC in with Myrinet's CRC
    - Write ISO_MARKER into the subtype field
    - Increment the tail pointer into the current Isotach receive buffer
  - Else if the packet is an EOP marker:
    - Find the CRC (it may be in the sort vector), and store it in the CRC field in the structure
    - Increment the tail pointer into the current Isotach receive buffer
  - Else, the packet must be either an iso-pointer or an sRef:
    - Increment the tail pointer into the current Isotach receive buffer
  - If the current Isotach receive buffer has grown past the threshold, or the received packet was an EOP marker, call `iso_dma()`
- If the toHost DMA engine is busy, call `check_dma()`
- If there are noniso packets waiting to be transferred to the host, call `noniso_dma()`

10. Performance Results and Analysis

As previously stated, our design goals were to design a baseline messaging layer that is comparable in performance to other available messaging layers, as well as implementing Isotach functionality with a minimal loss in performance. For the most part, we have realized these goals.

10.1. Overview of tests

We chose to measure the throughput and latency of our messaging layer for various configurations, and compare it using tests previously designed [Bar99] FastMessages and the initial Isotach prototype. The tests for Ironman were designed to match the previous tests as closely as possible. They were run for a variety of packet payload sizes – 64, 128, 256, 512, and 1024 bytes.

10.1.1. Latency

Our latency test sent 500 round-trip messages one at a time between 2 hosts. Once a message was returned by the server, it was copied into the applications address space. The clock was started immediately before the first message left the application running on the client, and was stopped once the last message was received from the server and copied by the client into its own address space. From these numbers, an average round-trip time was calculated.

10.1.2. Throughput

Our throughput test was designed to measure sending bandwidth for the client and receiving bandwidth for the server. The client sent 40 Mbytes of data to the server, which copied the received data into its own address space. At the client, the clock was started when the first message left the application, and stopped when the last message left the application. On the server, the clock was started upon receipt of the first message and stopped once the last message was copied into the application's address space. The test was repeated 5 times and an average was calculated.

10.2. Testbed

We ran the tests on dual Pentium III 450MHz machines with 256 MB RAM. For a more detailed description of the machines, see Appendix H. The hosts were configured in a variety of ways to capture all of the data we needed, as well as to perform some preliminary bottleneck analysis. The configurations were:

1. FastMessages 1.1 (FM)
2. Non-Isotach MBM without hardware SIU's in the link (NONISO)
3. Non-Isotach MBM with hardware SIU's in the link (NONISO_SIU)
4. Isotach MBM without hardware prefix and without host ordering (ISO_NOPRE)
5. Isotach MBM with hardware prefix and without host ordering (ISO_NOIOM)
6. Isotach MBM with hardware prefix and with host ordering (ISO)

A comparison of configurations 1 and 2 demonstrates how Non-Isotach MBM compares with FastMessages. To determine the overhead that Isotach guarantees impose on the system, configurations 3 and 6 can be compared.

Various other combinations of configurations can be compared to help determine where the bottlenecks in the system may be. Configurations 2 and 3 allow us to determine if the presence of the hardware in the link causes a drop in performance. In configurations 4-6, the hardware SIU is physically present in the link, however, sending Isotach packets without the hardware prefix causes them to pass through the SIU without any ordering being assigned. Configurations 4 and 5 indicate what effect the SIU has on Isotach packets. Finally, configurations 5 and 6 provide a rough indication of the costs of ordering on the host.

10.3. Results

The following graphs illustrate the performance of various components of the Ironman messaging layer. The actual data can be found in Appendix D.

10.3.1. Non-Isotach MBM vs. FastMessages 1.1

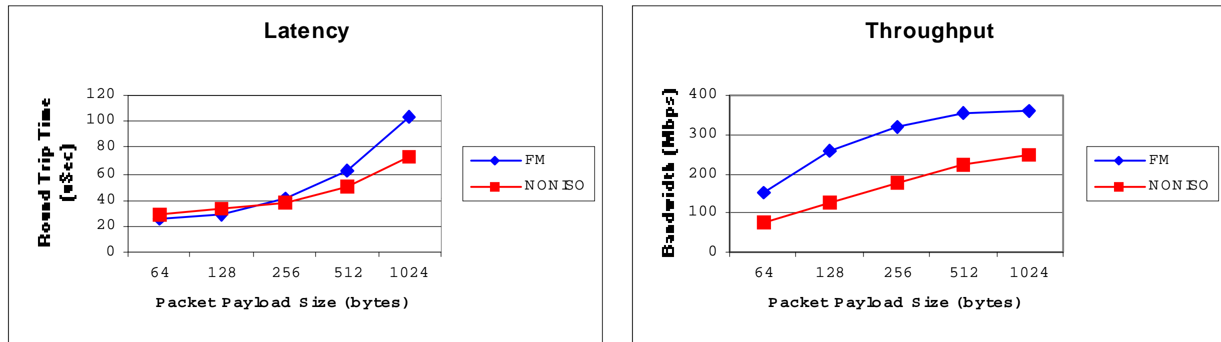


Figure 7

As can be seen in Figure 7, the Non-Isotach MBM protocol is roughly equal to or better than FastMessages in the latency tests. Unfortunately, the throughput results were more discouraging. Clearly, FastMessages has a much higher throughput than Non-Isotach MBM. See the next section for a discussion of a possible reason.

10.3.2. Isotach MBM vs. Non-Isotach MBM

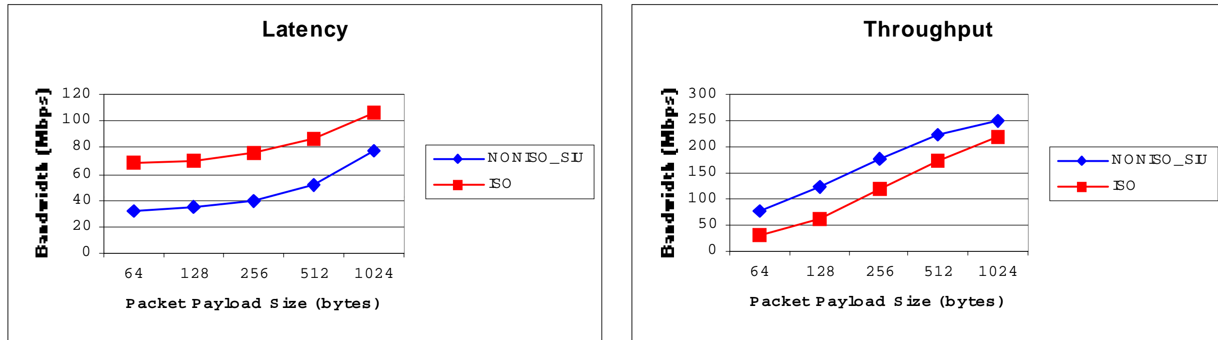


Figure 8

As shown in Figure 8, we have realized our performance goals in that Isotach guarantees do not cost more than double the latency and one-half the throughput of a comparable messaging protocol. However, we expected to observe much better performance for Isotach messages. To support Isotach guarantees, the Isotach MBM protocol introduces several additional factors into the system, including separate send/receive paths on the host and the hardware SIU. The following section attempts to determine which of these factors is affecting performance.

10.4. Bottleneck Analysis

10.4.1. Why is Non-Isotach MBM throughput lower than FastMessages 1.1?

As can be seen from Figure 7, our Non-Isotach MBM protocol outperformed FastMessages 1.1 in latency, but had a much lower throughput. One possible potential bottleneck in the Ironman design is the delivery queue on the LANai for Non-Isotach packets. For every packet that is received, the host must determine its location by performing a read across the PCI bus into the LANai's SRAM. These reads can be extremely costly and may be contributing to the lower throughput figures. (See the note at the end of the receive module for more information)

10.4.2. What is the effect of the hardware SIU in the link?

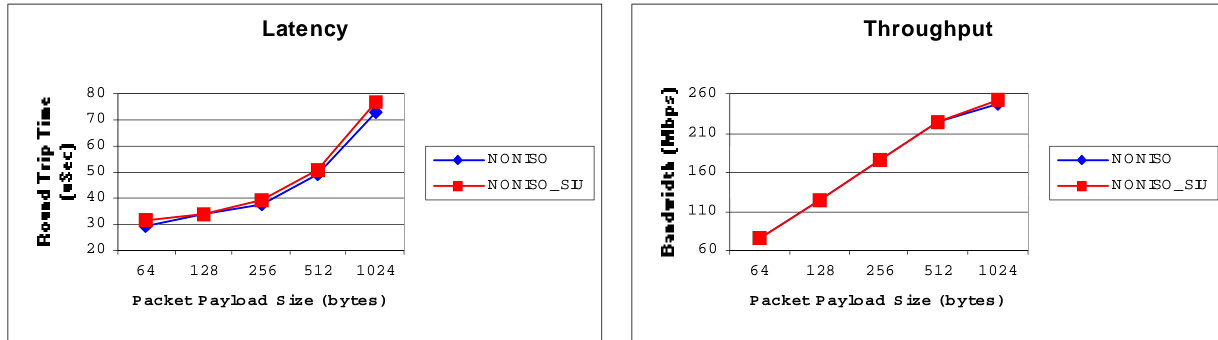


Figure 9

As can be seen from Figure 9, sending Non-Isotach packets through the hardware SIU had a minimal effect on both latency and throughput.

10.4.3. What is the effect of enforcing ordering on the host?

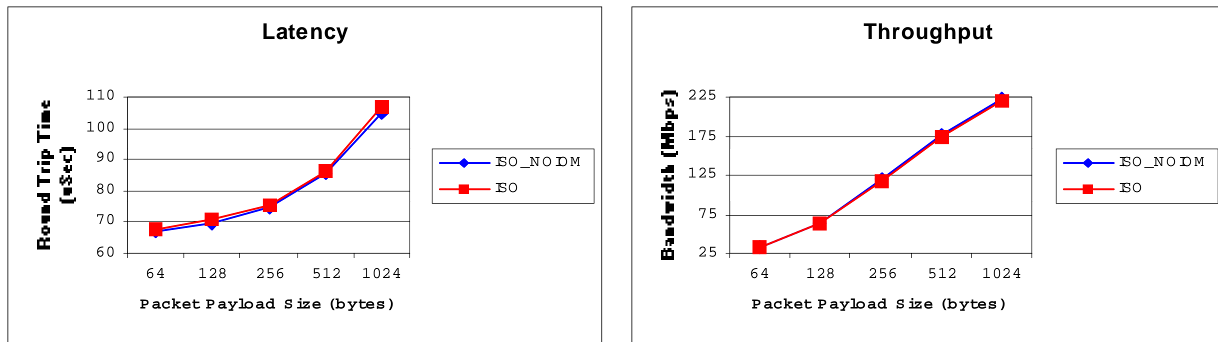


Figure 10

As was mentioned, configurations 5 and 6 can be compared to determine how much ordering costs on the host. In configuration 5, Isotach packets are sent through the hardware SIU's and assigned a timestamp, however, once they are received by the host they are delivered immediately to the application. In configuration 6, the packets are held until the host receives the appropriate EOP marker from the SIU. Figure 10 shows that waiting for the pulse does not significantly affect either throughput or latency.

10.4.4. What is the cause of the additional latency for Isotach MBM?

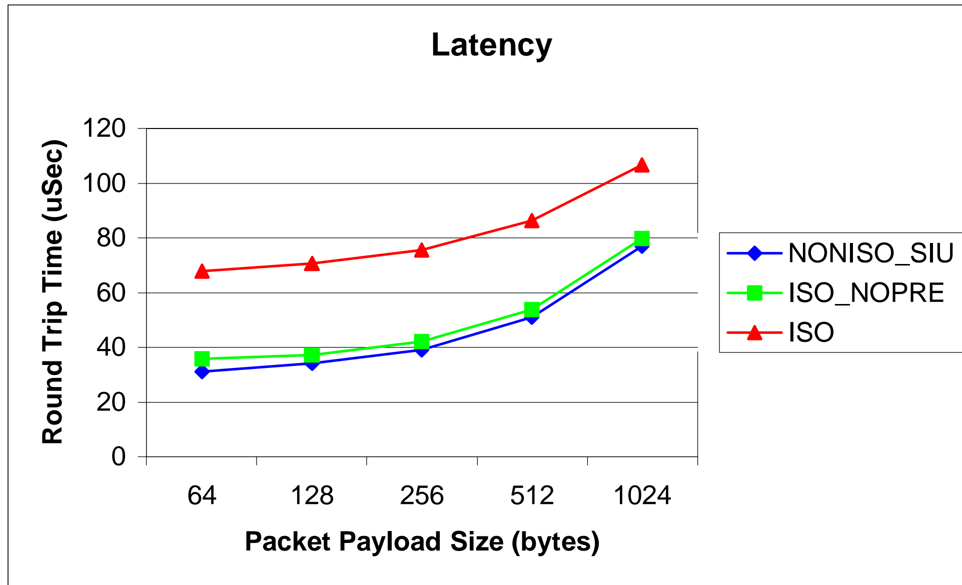


Figure 11

Given that Non-Isotach packets are not significantly slowed by the hardware SIU and that waiting for EOP markers does not increase the latency, we are left with two possibilities: either the Isotach send/receive paths on the host/NIU, or the Isotach send/receive paths on the hardware SIU. As can be seen in Figure 11, the send/receive path on the host does not appear to be causing the increased latency. The jump in the graph occurs when Isotach packets are sent through the Isotach path on the SIU. This leads us to conclude that the hardware SIU's are responsible for the increase in latency for the Isotach MBM protocol.

10.4.5. What is the cause of the lower throughput for Isotach MBM?

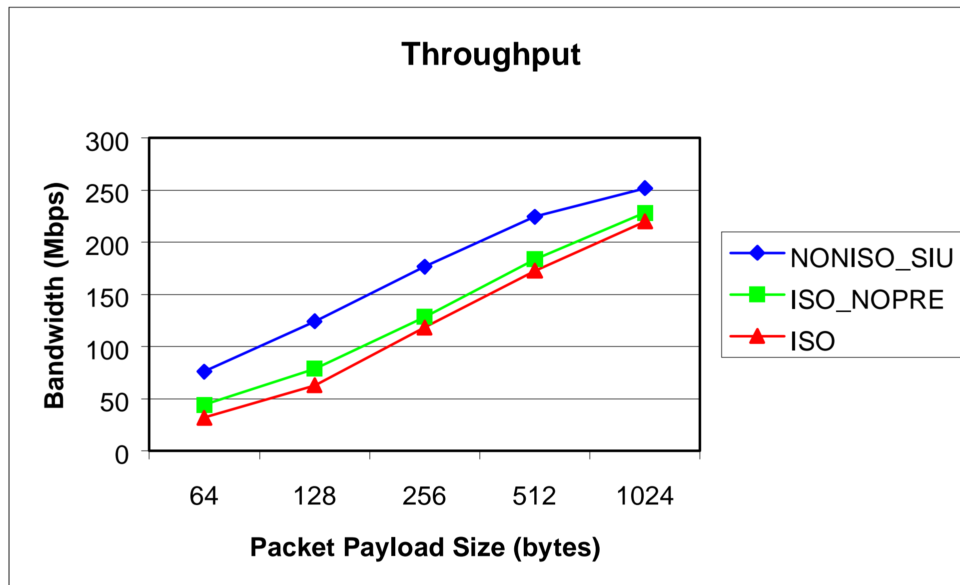


Figure 12

It appears that the send/receive paths on the host are responsible for the lower Isotach throughput. In Figure 12, we observe that the largest gap in the graph is present between the Non-Isotach MBM protocol and Isotach packets that were not sent with a hardware prefix. This appears to indicate that the Isotach paths on the SIU are not significantly decreasing throughput. We believe that the method of sending Isotach messages in two different packets is resulting in the lower throughput figures. Because of this design decision, for each Isotach message there are 4 transfers of data across the PCI bus (2 to send, 2 to receive), versus only 2 transfers for the Non-Isotach protocol. This is most clearly evidenced by the fact that the curves appear to be converging for larger packet sizes. For larger packet sizes, the cost of transferring the additional, much smaller Isotach packet appears to amortized.

It should be noted however that there is a slight difference in the Isotach receive path on the LANai between configurations 4 and 6. In configuration 4, the SIU will not generate any EOP markers (as it is in Host-Host mode). Thus, every Isotach packet is DMA'd up to the host as soon as it is received. In configuration 6, Isotach packets are batched until an EOP marker is received. Thus, even though there is less data (no EOP markers) to be transferred in configuration 4, the DMA engine must be started more frequently. It is unclear what effect this modification has on performance. In actuality, the send/receive paths on the host may not contribute as much to the loss in Isotach throughput as Figure 12 appears to indicate. This hypothesis is difficult to test since without EOP markers, Isotach packets cannot be transferred in a batch.

Regardless, the convergence of the curves shows that the send/receive paths on the host are contributing somewhat to the decrease in throughput.

11. Conclusions and Future Work

11.1. State of the system

With the exception of the Isotach Shared Memory Model, everything described in this document has been implemented, debugged, and tested. Various low-level design issues have been omitted from this document for brevity; however, these details can be found within the source code (See Appendix J). The Ironman system is currently up and running in the Isotach lab using hardware TM's and SIU's. The source can be found on the file server in the Isotach Lab (in Small Hall) at /home2/isotach/ironman/v3.

11.2. Performance

Preliminary performance tests have shown that we have achieved our design goals with regards to Isotach overhead. Depending on the size of exchanged messages, Isotach latency ranges from 1.4 to 2.3 times greater than the Non-Isotach protocol. Isotach throughput ranges from 42% to 89% that of the Non-Isotach protocol.

Through some additional testing, it has been determined that much of this overhead is a direct result of sending data through the hardware SIU. In addition, some of this overhead is a result of having to transfer two separate packets from the LANai to the host for each Isotach message. However, given the delays present in the Isotach paths on the SIU, sending an Isotach message as a single Isotach packet probably will not result in significant improvement in performance. Further study is necessary to determine whether these figures can be improved.

Additionally, the Non-Isotach protocol was tested against FastMessages 1.1 to determine how its performance compared with an established, widely available messaging layer. Whereas our protocol was equal to or better than FastMessages in latency, throughput results showed that there are some severe bottlenecks present in the Ironman design.

11.3. Future Work

Currently, the Isotach Shared Memory Model protocol has not been implemented. The preliminary design is included in this document merely for completeness. Completion of the design and implementation of this protocol still needs to be done.

Additionally, the Isotach protocols currently utilize send and receive frames for transporting packets through different sides of the messaging layer. This design should be extended to the Non-Isotach protocols as well. This would provide for easier readability of the code, and would improve performance for messages that are smaller than the maximum payload size.

Another possible performance improvement would be to implement write combining. Currently, data is transferred across the PCI bus to the LANai's SRAM using C's `memcpy` function. Write combining is a hardware feature of the Pentium Pro processor that can potentially improve programmed I/O writes. Note that this enhancement would most likely result in equal performance gains for all protocols.

Currently, all Isotach MBM messages are sent out as a pair of messages: one `iso_pointer` and one `ordered` packet. For small Isotach MBM messages there may be a significant increase in performance if the payload of the MBM message is piggybacked onto the `iso_pointer`. This would allow smaller packets to avoid the double DMA associated with the `iso_pointer/ordered` packet pairing.

As mentioned in the introductory paragraphs of this paper, an all software implementation of the Isotach mLayer does not exist. This mLayer depends on the Isotach custom hardware devices for correct operation. Without them, there is no logical time or ordering of messages. The mLayer could be modified to perform the same operations that the hardware SIU and TM perform. This would increase the amount of work that the host and LANai need to do, but it would allow Isotach to be used by individuals who do not have access to the Isotach custom hardware devices. Furthermore, with the increases in Myrinet LANai processor speeds and PCI bus speeds, a software implementation of the SIU and TM may actually outperform the hardware devices.

Finally, the system needs to be studied in detail to determine which modules are the bottlenecks in the send and/or receive paths. One potential method is to measure the average execution times of the poll function exported by each module. This would indicate which modules are consuming the most time. This type of measurement would be simple to implement. Another possibility is to measure the latencies and throughputs of each module. This would involve much more intrusive modifications to the code that may skew the results. After these bottlenecks are uncovered, the final step would be to redesign the affected modules to eliminate the bottlenecks and increase the performance of the mLayer.

References

[ABD98] S. Araki, A. Bilas, C. Dubnicki, J. Edler, K. Konishi, J. Philbin, "User-Space Communication: A Quantitative Study", Proc. Supercomputing '98, Nov 1999.

[Bar99] R.G. Bartholet, "A Performance Study of Isotach Version 1.0", Univ. of Virginia, CS Dept., Tech Rep. CS-99-13, Apr 1999.

[BJM95] G. Buzzard, D. Jacobson, S. Marovich, J. Wilks, "Hamlyn: a high-performance network interface with sender-based memory management", CS Lab., Hewlett-Packard Laboratories, Palo, Alto, CA, HPL-95-86, July 1995.

[BSV99] R.G. Bartholet, D.C. Szajda, and R. Venkateswaran, "A Proposed Revised Isotach Architecture", internal document (if you need a copy, ask Craig)

[Lam78] L. Lamport, "Time, clocks, and the Ordering of Events in a Distributed System", Communications of the ACM, 21(7): 558-565, July 1978.

[Myr01] Myrinet Documentation

[Myr02] Myrinet Users Guide

[Myr03] LANai 4.X Documentation

[Reg99] J. Regher, "An Isotach Implementation for Myrinet", Univ. of Virginia, CS Dept., Tech Rep. CS-97-12, May 1997.

[RWW97] P. Reynolds, C. Williams, R. Wagner, "Isotach Networks", IEEE Transactions on Parallel and Distributed Systems, 8(4), April 1997.

[Sza99] D.C. Szajda, "Testing the Isotach Prototype Hardware Switch Interface Unit", Univ. of Virginia, CS Dept., Tech Rep. CS-99-27, Jun 1999.

[Wil93] C. Williams, "Concurrency Control in Asynchronous Computations", PhD thesis, University of Virginia, January 1993.

[Wil99] C. Williams, "The Isotach Prototype Messaging Layer Architecture: Tinman Design", internal document (if you need a copy, ask Craig)

Glossary

copyset - Each page of Isotach shared memory is present in one or more processors. The copyset of a given page of Isotach shared memory is the set of processors at which the page is present.

deleted/freed - An element in a queue is deleted when its contents are no longer needed and is freed when the head pointer of the queue is moved past the element so its space can be reused.

GC - Isotach group communication, i.e., sending and receiving barriers (including host-level tokens) and signals.

NIU mLayer - The part of the mLayer that runs on the NIU.

message vs. packet - A message (as opposed to a packet) is the unit of communication at the application interface. A packet is the unit of communication at the network interface. Thus the application sends and receives messages. The mLayer sends and receives packets. This distinction is really only important in systems that can split a message into multiple packets for sending. Unfortunately "message" has multiple meanings, depending on context. See message vs. access.

message vs. sRef - A message (as opposed to an access) is the unit of communication in the MBM of computation. An sRef is a special type of message occurring in SMM computations that carries instructions for an operation on shared memory.

mLayer - An inclusive, generic term for the software messaging layer. Composed of the host mLayer (running on the host) and the NIU mLayer (running on the NIU).

net-message/net-packet - A message/packet that is sent by a host to a different host, i.e., not a self-message.

net-isochron - A net-isochron is an isochron containing one or more net-packets.

noniso protocol - A simple non-Isotach MBM messaging layer written by us for execution on the Isotach prototype.

non-Isotach messages - Messages that do not carry Isotach TSs and for which the Isotach invariant does not necessarily hold. Non-Isotach messages include messages sent by the noniso protocol but also messages that are sent in service of an Isotach protocol but that do not require Isotach guarantees. Read responses are currently the only example of this latter type of non-Isotach message. Assigns (and cancels) may become non-Isotach messages in the future. (The decision depends on the trade-off between increased complexity in the shmem to deal with early arriving assigns and decreased latency for assigns.)

pinned memory - An area in host memory that cannot be swapped out. The NIU can DMA to and from memory only if it is pinned memory.

self-message/self-packet - A message/packet sent by a host to itself. Isotach self-messages are useful as a way to ensure that a local action takes place at the same logical time as remote actions. Self-messages are not permitted in the noniso protocol. *Cf*. net-message.

self-isochron - A self-isochron is an isochron containing no net-packets. *Cf*. net-isochron.

self-ref. An sRef that accesses a local copy of a shared variable. A self-ref is a special case of a self-packet.

send side/receive side - The send side is the path through the mLayer starting with the host and ending at the network itself. Similarly the receive side is the path through the mLayer starting at the network and ending at the host.

SMM - Shared Memory Model of parallel computation (for readers familiar with the first generation prototype software, do not confuse SMM as used here with the SMM in the first generation software, the module that manages the shared memory.

shmem - The Isotach shared memory module. The shmem is the module at each Isotach host that executes all assesses to the portion of Isotach shared memory local to that host.

specific sender-based flow control - A type of sender-based flow control in which the sender specifies the location into which its messages are to be written at the receiving node.

sRef - An Isotach message that is an access to Isotach shared memory, i.e., an Isotach read, write, sched, or assign operation. A single Isotach SMM call may result in several sRefs, e.g., a call to iso_write() results in the creation of *c* sRefs, where *c* is the number of copies of the variable being written.

packet - The network "on the wire" level message unit. In the current prototype, since we do not packetize messages, "message" and "packet" are the same.

host mLayer - The part of the mLayer that runs on the host. *Cf*. NIU mLayer.

Appendix A. Isotach API

The Isotach API supports both Isotach and Non-Isotach communication and both shared memory model (SMM) and message based model (MBM) programming. The API also supports a Non-Isotach barrier and Isotach group communication (barriers and signals). The four interfaces supported by the API are as follows:

- Non-Isotach Message Based Model (noniso-MBM)
- Non-Isotach Group Communication (noniso-GC)
- Isotach Message Based Model (iso-MBM)
- Isotach Shared Memory Model (iso-SMM)
- Isotach Group Communication (iso-GC)

An application can use any subset of the interfaces, except that it cannot use both Isotach MBM and Isotach SMM. Currently at most, one application per host can use the network.

The v.2 API extends the v.1 API, written by John Regehr, Chris Milner, and Dale Newfield. A discussion of issues related to the API may be found in ~ccw/group/design_issues.fm.

A.1. Initialization/Shut-down

This section describes functions used by the application to open and close its network connection.

`open_net()`

Purpose: Open the application's connection to the network. Initiate the mLayer for a new run. The application must call this function before calling any other function in the API. [Corresponds to `iso_init()` in the v.1 API. Renamed because the API includes a noniso protocol interface.]

Arguments: none.

Return type: int - 0 for success; nonzero for failure

`try_close_net()`

Purpose: Try to close the application's connection to the network. The call will not succeed until after all nodes have invoked try_close_net(). [Corresponds to iso_deint() in the v.1 API.]

Arguments: none

Return type: int - 0 for success; nonzero for failure

Caveat: The application must call this function repeatedly until the call succeeds. In an MBM computation, after an unsuccessful call, the application must check for and handle any incoming messages. For applications only using Non-Isotach messaging, this function call initiates a noniso barrier, which must be completed for success to be returned. For

applications using any Isotach functionality, this function call initiates an Isotach barrier. The Isotach barrier must complete for the function to return success.

A.2. System housekeeping and status functions

This section describes functions that can be called by the application to hand the mLayer control and to obtain system information.

`poll()`

Purpose: Housekeeping. A program that is not frequently sending or receiving messages (in any of the interfaces) must frequently call this function to hand control to the mLayer to allow it to perform its work.

Arguments: none

Return type: int - 0 for success; -1 if no message is available for delivery; some other nonzero value for any other type of failure.

`get_node_number()`

Use: `get_node_number(hostname)`

Purpose: return the NODEID of node *hostname*. [iso_get_node_number() is the corresponding function in the v.1 API.]

Arguments: `char *hostname`

Return type: `int`, the NODEID of node *hostname*, or -1 if not found.

`get_my_node_number()`

Purpose: return the NODEID of this node.

Arguments: none

Return type: `int`, the NODEID of this node *hostname*, or -1 if not defined.

`get_max_payload()`

Purpose: return the maximum payload of a message in bytes.

Arguments: none

Return type: `int`, the max payload size of the mLayer.

`get_number_of_hosts()`

Purpose: return the number of hosts in the system. Excludes any hosts emulating TMs.

Arguments: none

Return type: `int`, the number of hosts in the system.

`get_MBM_ver()`

Purpose: return the level of Isotach ordering supported.

Arguments: none

Return type: `int`, the version number of the mLayer, or -1 for failure.

`get_SIU_state()`

Purpose: returns TRUE if a hardware SIU is configured for this mLayer, FALSE otherwise.

Arguments: none

Return type: `int`, a TRUE or FALSE value.

A.3. Non-Isotach Message Based Model

The only noniso specific function is the function for sending noniso messages.

`send()`

Use: `send(target, data, size)`

Purpose: send a noniso message to *target*, the NODEID of the destination node. *data* is a pointer to the data to be sent, and *size* is the size of the message in bytes.

Arguments: `int target, void *data, int size`. Target must be a node other than my_id, i.e., the application cannot send a noniso message to itself. A call to send that specifies the local node as the target fails.

Return type: `int` — 0 for success, nonzero for failure, e.g., because target is not a valid NODEID. Note that success does not mean that the message has been received, only that the send_message call succeeded.

Caveat: a call to `send()` may fail. The application should check the return code.

`receive()`

Purpose: Return a pointer to a noniso message. The call is non-blocking. The application must either consume the message or copy it before again calling `receive()`.

Arguments: A structure passed as a reference parameter containing: 1) the sender id; 2) the length of the data; 3) a union component containing either a pointer to the data or the data itself. The union is interpreted by reading the size field. If the size is less than or equal to four bytes, the data is contained within the noniso_mbm message. Otherwise, the data is pointed to by the noniso_mbm message.

Return Type: `int` - 0 for success, -1 if no message is available for delivery, other nonzero value for any other type of failure.

`noniso_mbm` structure format:

As mentioned above, the contents of a noniso_mbm structure can be either a pointer to some data of arbitrary size, or 4 bytes of data. The structure is interpreted by inspecting the size field. Any size greater than 4 bytes means that the data is stored elsewhere, and the 3rd field is a pointer. Otherwise, the 3rd field is actual data.

A.4. Non-Isotach Group Communication

The application may participate in a noniso barrier using the functions described in the section.

`initiate_barrier()`

Purpose: Start participation in a noniso barrier. The call will fail if the previous noniso barrier is not locally complete.

Arguments: none.

Return type: `int` - 0 for success; nonzero for failure

`barrier_completed()`

Purpose: Return status of current barrier.

Arguments: none.

Return type: `int` - 0 if the current barrier is complete; nonzero if there is no current barrier or if current barrier is not complete

A.5. Isotach Message Based Model

The only Isotach MBM specific function is for sending Isotach messages. Messages are received through a call to receive(), described in another section.

`iso_send()`

Use: `iso_send(target, data, size, last_in_isochron)`

Purpose: send an Isotach message isochronously to *target*, the NODEID of the destination node. *data* is a pointer to the data to be sent, and *size* is the size of the message in bytes. If this message is the last in the isochron, set *last_in_isochron* to TRUE. [This function (minus the last argument) appears in the v1 API.] The memory pointed to by *data* can be reused as soon as the call returns.

Arguments: `int target, void *data, int size, int last_in_isochron`

Return type: `int` - 0 for success, nonzero for failure, e.g., because target is not a valid NODEID. Note that success does not mean that the message has been received, only that the call to iso_send()succeeded.

***Caveat*:** a call to send() may fail. The application should check the return code.

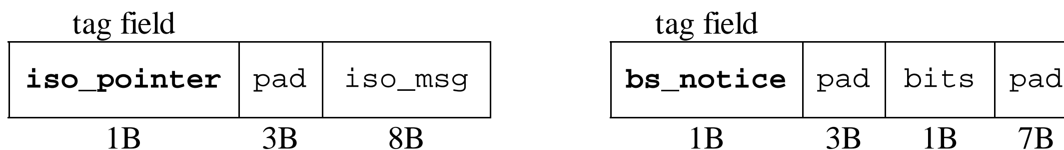
`iso_receive()`

Purpose: Return a pointer to an Isotach message or return a bs-notice. All applications receive bs-notices through this call. An MBM Isotach application also receives messages. The call is non-blocking. The application must either consume the message or copy it into its own space before again calling `iso_receive()`.

Arguments: A structure (`iso_mbm`) passed as a reference parameter containing: 1) the sender id; 2) the length of the data; 3) a union component containing either a pointer to the data or the data itself; and 4) a tag indicating how to interpret the contents of the union (as a pointer, as a bs- notice, or as a data

Return Type: `int` - 0 for success, -1 if no message is available for delivery, other nonzero value for any other type of failure.

`iso_mbm` structure format:



The body of an `iso_mbm` structure either an `iso_msg` structure, or the bits from a `bs_notice`. The tag field allows the application to interpret the contents of an `iso_mbm` message appropriately.

`iso_msg` structure format:

sender	length	*data
2B	2B	4B

An iso_msg contains a sender field, the length of the data and a pointer to the corresponding data.

A.6. Isotach Shared Memory Model

The API supports an Isotach SMM interface only. This section describes the SMM interface.

`iso_read()`

Use: `iso_read(shaddr, laddr, last_in_isochron)`

Purpose: schedule a read access to the specified word of shared memory. When the value returns read, write it in the specified local variable. An `iso_get_read()` call is nonblocking — the process blocks if it tries to retrieve the value (via an `iso_retrieve()` call) before the read has returned. [`iso_read32()` is the analogous function in the v.1 API. (If we want to specify a size, it can be a parameter)] The call fails if the process already has read_cap outstanding reads or if the shared address specified is invalid.

Arguments: `shmem_addr_t shaddr, struct isovar *laddr,`
`int last_in_isochron`

Return type: `int` — 0 for success, nonzero for failure.

`iso_retrieve()`

Use: `iso_retrieve(laddr)`

Purpose: returns when the specified local variable is valid, *i.e.*, when the most recent locally issued `iso_read()` that names the local variable as the location to write its result has returned its result.

Arguments: `struct iso_var32 *laddr`

Return type: `long int`

`iso_write()`

Use: `iso_write(shaddr, argueval, last_in_isochron)`

Purpose: write a value to a shared memory address. It is equivalent to calling `sched()` and then `assign()`. Writes are nonblocking. [This function corresponds to `iso_write32()` in the v.1 API.]

Arguments: `shmem_addr_t shaddr, long int val, int last_in_isochron`

Return type: `int` — 0 for success, nonzero for failure.

`iso_sched()`

Use: `iso_sched(shaddr, last_in_isochron)`

Purpose: schedule an assign to a shared memory location.

Arguments: `shmem_addr_t shaddr, int last_in_isochron`

Return type: `int` — 0 for success, nonzero for failure.

Caveat: A host may send a sched on variable *v* only if it has sent the assign for its previous sched (if any) on *v*.

`iso_assign()`

Use: `iso_assign(shaddr, val, last_in_isochron)`

Purpose: supply the value to be written in an access scheduled by a previously issued `iso_sched()`.

Arguments: `shmem_addr_t shaddr, long int val, int last_in_isochron`

Return type: `int` — 0 for success, nonzero for failure.

Caveat: An application can issue an assign only if it has an outstanding sched on the same shared variable.

A.7. Message functions common to the Isotach SMM and MBM

`iso_end()`

Use: `iso_end()`

Purpose: the application may call this function to inform the mLayer that the last message/access in the current isochron has been issued. (An alternative way to signal the end of an isochron is by setting *last_in_isochron* when issuing the last message/access in an isochron). An iso_end() issued while no isochron is being issued is a NOP. [This function also appears in the v1 API.]

Arguments: none

Return type: `int` - 0 for success, nonzero for failure.

A.8. Isotach Group Communication (GC)

The functions listed below are for registering, releasing, and using GC resources. A GC resource is a signal or barrier channel. The Isotach cluster supports five signal channels (plus a reset signal channel that is not exposed at the application level) and two barrier channels.

A.8.1. Registering and Releasing Signals and Barriers

The application must obtain permission from the mLayer before using a GC resource.

Signals – Sending a signal propagates a single bit of information to all application processes that have registered that signal. All processes receive the signal at the same logical time. Furthermore,

all Isotach messages a process sends before sending a signal will be received before that signal is received. If multiple processes send a signal on the same channel concurrently (so that the signal arrives in the same epoch), only one signal will be received.

Barriers – The GC interface supports two types of barriers: STRONG and WEAK. An execution of a weak barrier can complete only after all processes that registered the barrier have participated in the execution of the barrier. A strong barrier makes the additional guarantee that all Isotach messages issued by a participating process prior to participation are received at a given host before the barrier completes at that host. In other words, after completion of a strong barrier at host h, host h will not receive any Isotach messages issued before the sender participated in the barrier.

The mLayer will grant the application permission to use a GC resource only if the mLayer has not been configured to use the resource itself. The application can register and release (clear) GC resources dynamically.

`iso_register_signal()`

Use: `iso_register_signal(channel)`

Purpose: obtain permission from the mLayer to use the specified signal channel. After successfully registering a signal, the application can send the signal and the mLayer will forward the signals received on that channel to the application (in the usual way, by making the message visible through poll/receive). The application is always implicitly registered to receive reset signals, but it cannot register the reset signal and cannot send a reset signal.

Arguments: `int channel`, specifying which signal channel (1-5) the application intends to use; execute once for each signal channel the application uses;

Return type: `int` 0 for success, nonzero for failure (normally because the mLayer claims the channel or because the argument is out of bounds).

`iso_clear_signal()`

Use: `iso_clear_signal(channel)`

Purpose: relinquish specified signal channel. Clearing an unregistered signal is a NOP.

Arguments: which signal channel (1-5) the application is relinquishing; execute once for each signal channel that the application seeks to relinquish.

Return type: `int` - 0 for success; nonzero for failure.

`iso_register_barrier()`

Use: `iso_register_barrier(bmode, channel)`

Purpose: obtain permission from the mLayer to use the specified barrier channel as a barrier and declare whether the barrier is a weak or strong barrier. After successfully registering a barrier channel, the application can use it.

Arguments: `int bmode, int channel`. The first argument should be STRONG or WEAK (defined constants) to indicate how the barrier will be used. The second argument indicates which barrier channel (0-1) the application intends to use.

Return type: `int` - 0 for success; nonzero for failure (e.g., because the mLayer claims the channel or it is registered for another use).

`iso_clear_barrier()`

Use: `iso_clear_barrier(channel)`

Purpose: relinquish specified barrier channel. Clearing an unregistered barrier is a NOP.

Arguments: `int channel`, which barrier channel (0-1) the application is relinquishing

Return type: `int` - 0 for success; nonzero for failure (normally because argument is out-of-bounds.)

A.8.2. Using GC Resources

The functions below are for *sending* signals and barriers. The application receives notice of the receipt of a signal or of barrier completion receive().

An application must register a signal or barrier channel before attempting to use it.

`iso_send_signal()`

Use: `iso_send_signal(channel)`

Purpose: send specified signal

Arguments: `int channel`, indicating on which signal channel (0-1) the application is sending

Return type: `int` - 0 for success; nonzero for failure (normally because the process is in the middle of sending an isochron, the argument is out-of-bounds, or the application has not registered the specified signal channel.)

Caveat: do not call while sending an isochron

`iso_barrier()`

Use: `iso_barrier(bmode, channel)`

Purpose: participate in the specified barrier. The application must have already registered the barrier. The barrier type must agree with the type specified when the barrier was registered. This function is non-blocking. The barrier is complete when the application receives a barrier/signal notice (through `receive()` call).

Arguments: `int bmode`, indicating the type of barrier and `int channel`, indicating in which barrier channel (0-1) the application is participating

Return type: `int` - 0 for success; nonzero for failure. Fails if the application has not successfully registered the barrier; if the type of barrier specified by the first argument differs from the type specified when the application registered the barrier; if the application is in the middle of sending an isochron; or if the previous execution, if any, of a barrier on the same channel is not locally complete. An execution of a barrier is locally complete at a given host if the mLayer has returned a barrier notice to the host for that barrier execution.

Caveats: Do not call while sending an isochron. Do not call if the previous barrier on the same channel is not locally complete.

Appendix B. Packet Formats

This section describes the format of packets and the format of data items internal to the mLayer.

B.1. Network Packet Formats

This section describes the format of packets as they enter/leave the network interface

B.1.1. Non-Isotach Packet Structure

route	packet type	packet subtype	pad 1	sender	DMA base	payload length	pad 2	credit info	application payload	CRC
1-4B	2B	1B	1B	4B	4B	2B	2B	4B	variable	1B

route: A route can be from 1 to 4 bytes long. Each routing byte is called a 'flit' on a Myrinet network, and is consumed by a switch. Therefore, 4 routing bytes allows us to use a maximum network diameter of 4 switches.

packet type: The noniso packet type will not be registered with Myricom and thus we should make it easy to change (so we can change it in the unlikely event that we discover a conflict with a registered noniso packet type that we think anyone might want to use with Isotach).

packet subtypes: `sync`, `sync_ack`, `sync_done`, `credit`, `iso_credit`, `request_credit`, `request_iso_credit`, `barrier`, `noniso_mbm`, `ordered`, `read_response`

sender: The sending node ID of the packet.

DMA base: The starting location in pinned memory to which the NIU should DMA the message.

payload length: The number of bytes in the application payload. The receiving NIU DMAs the message beginning with the packet type and ending with the CRC.

credit info: The sender writes the head pointer of the receiver's receive buffer at the sender into the credit information field.

Sync packet

Sync packets (aka "hello packets") are sent at system initialization. The application payload contains the sending host's DMA base information for both Isotach and Non-Isotach regions in pinned memory.

Sync-ACK packet

Sync-ACK packets are send in response to sync packets at initialization. The application payload is empty.

Sync-done packet

Sync-done packets are used in the final stage of host synchronization to ensure that all hosts finish synchronization at the same time. Sync-done packets effectively create a simple barrier.

Credit packets (Isotach and Non-Isotach)
A credit packet is a noniso packet of subtype credit or iso_credit sent when there is no normal traffic on which to piggyback credit info. For a given protocol, the DMA base is the same for all credit packets sent by the same sender to the same receiver: credit_packet_base for noniso credit packets and iso_credit_packet_base for Isotach credit packets. The application payload is always zero bytes for any credit packets.

Request-Credit packets (Isotach and Non-Isotach)
A request credit packet is sent when a sending host has attempted to clear a packet for sending a certain number of times, and failed. This situation may be due to a credit packet starvation problem. If one host is sending and one host is receiving, the receiving host must periodically send credit packets back to the sending host. If one of these credit packets cannot be shipped (most likely due to Myrinet hardware flow control) the receiving host may never send another credit packet. This situation is very rare, and has never been observed but is theoretically possible. Therefore, to prevent a deadlock situation, request-credit packets are used.]

Non-Isotach Barrier packets
These packets contain no actual data, other than the sender ID. The sender ID is used to determine who has completed the barrier.

Non-Isotach Message Based Model (MBM) packets
These packets are used to send noniso-MBM information from host to host. They are created upon a call to the API function `send()` and contain a variable payload of data, up to the maximum packet size configured for the mLayer. Credit information is piggybacked on noniso-MBM packets in the credit_info field.

Non-Isotach Ordered packets
These packets are essentially the same as the noniso-MBM packets except that these packets are placed into a different region in the receiving host's pinned memory. In addition, each packet of subtype ordered is associated with a corresponding `iso_pointer`. The ordered packet contains the data for an Isotach MBM packet.

Read Response
A read response is an noniso packet with subtype `read_response`. The application payload contains two fields, each 1 word long: the data (the value that is being returned), followed by a pointer to the local variable in which the data should be written.

B.1.2. Iso-pointer

iso prefix	route pad	route	packet type	packet subtype	TS	sender	credit info	data pointer	CRC
4B	4B		2B	1B	1B	2B	2B	4B	1B

An iso-pointer is an Isotach packet of subtype `iso_pointer` that points to an ordered packet. The pointer field is the address at which to find the corresponding noniso packet. Unfortunately,

packet length is the characteristic that distinguishes Isochron markers from iso-pointers. Iso-pointers are 13 bytes long when received; isochron markers are 8 bytes long. (Ideally, we alter the SIU so that it assigns 0x0602 as the packet_type for isochron markers.)

B.1.3. Shared Memory Reference (sRef)

iso prefix	route pad	route	packet type	packet subtype	TS	sender	credit info	shadder	data / pad	CRC
4B	4B		2B	1B	1B	2B	2B	4B	4B	1B

An sRef is an Isotach packet of subtype `iso_read`, `iso_write`, `iso_sched`, or `iso_assign`. The payload of the message is composed of the shadder, the address of the shared variable, and the operand to the operation, if any. The data field contains the following, depending on the packet subtype:

1. `iso_read`: 4B lvar
2. `iso_write`: 4B value
3. `iso_sched`: 4B pad
4. `iso_assign`: 4B value

B.1.4. BS-marker

010	pad	barrier count	signal bits		barrier bits	CRC
			reset	host signals		
3 bits	5 bits	1B	1 bit	5 bits	2 bits	1B

This format for the bs-marker marker is copied for convenient reference from the Isotach specification. The host sends the SIU a bs-marker to initiate sending a signal or participating in a barrier. The initial 3 bits identify the packet to the SIU as a bs-marker.

B.1.5. Isochron marker

packet type	isochron CRC/ packet subtype	TS	source	iso_id	CRC
2B	1B	1B	2B	1B	1B

This format for the isochron marker is copied for convenient reference from the Isotach specification. When receiving an isochron marker, the NIU OR’s the isochron CRC into the CRC field and writes isochron_marker into the second field (which now becomes the packet subtype field).

B.1.6. EOP marker

copy of token (up to CRC)								
packet type	packet subtype	signal bits	barrier bits	pad	TS	count	sort vector	CRC
2B	1B	6 bits	2 bits	2B	1B	1B	0-32B	1B

Assuming that sort vectors hold a maximum of 32 items, the sort vector can be at most 32B long. Since the header and CRC fields are 9B, an EOP marker is at most 41B (11 words) long.

B.2. Internal Packet Data Structures

This section describes the internal format of packets before they are sent out onto the network, and after they are received from the network. These formats correspond to the structures in the file `utils.h`.

B.2.1. Non-Isotach Packet Structure (`PACKET`)

packet type	packet subtype	pad 1	sender	DMA base	payload length	pad 2	credit info	application payload	route
2B	1B	1B	4B	4B	2B	2B	4B	variable	4B

This packet is used for all noniso messages that are stored on the host. The application payload is configured to be the max payload size + 4 bytes. The extra 4 bytes is there to allow room for the CRC if the packet sent uses all of the available bytes.

B.2.2. Isotach Packet Structure (`ISO_PACKET`)

Isotach Packet

packet type	packet subtype	TS	sender	credit info	[*data, sRef]
2B	1B	1B	2B	2B	8B

sRef

shadder	data
4B	4B

Isotach packets can either be iso_pointers or sRefs. Therefore, the Isotach Packet structure is contains the union of a pointer to data and an sRef structure.

Isotach Send Frame

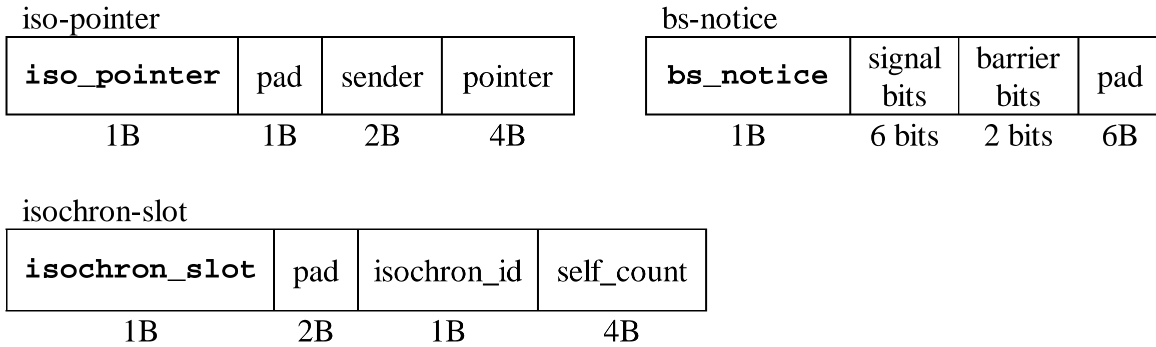
Isotach Prefix	route 2	route 1	pad	`ISO_PACKET`
4B	4B	2B	2B	16B

Isotach Receive Frame

`ISO_PACKET`	CRC
16B	4B

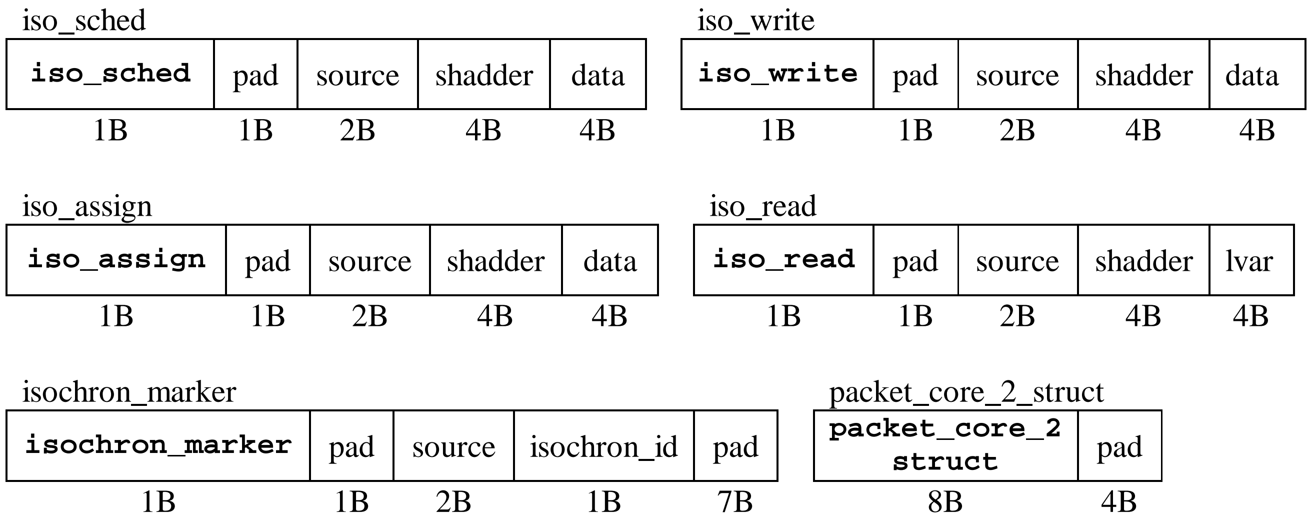
Isotach packets are placed into send/receive frames. The use of send/receive frames allows us to store different information relevant to the packet, which depends on the current path of the packet.

B.2.3. Packet-core-2 structures



These structures appear in `iso_delivery_q`. A `packet-core-2` is 2 words long. The first byte is the packet subtype and determines the interpretation of the remainder of the packet. An isochron-slot contains a count, self_count, of the number of self-messages it represents.

B.2.4. Packet-core-3 structure



These structures appear in the IOM buckets, `self_ref_buf`, and `shmem_buf`. A `packet-core-3` is 3 words long. The first byte is the packet subtype and it controls the interpretation of the remainder of the packet.

Appendix C. Isotach Implementation Constants

These are named values that do not change after initialization. The constants are all tunable parameters for the messaging layer. They are all located in the file hostman/constants.h for ease of modification.

`SIU`

This parameter indicates whether or not the mLayer is configured to use a hardware SIU. Set its value to 1 to enable hardware SIU support, 0 otherwise.
Coded: 0, 1

`MBM_VER`

This parameter defines which version of the IOM is being used. Currently there are two available versions. Setting this value to 1 disables Isotach ordering and self-messages. A value of 2 enables these features.
Coded: 1, 2

`MAX_PAYLOAD_SIZE`

Defines the maximum amount of bytes that may be sent in the payload of an Isotach or Non-Isotach MBM packet. Can be configured to any integer, but normally accepted (and tested) values are: 64, 128, 256, 512 and 1024.
Coded: 64, 128, 256, 512, 1024

`SIZE_OF_PINNED_MEMORY`

Must be configured to be less than or equal to the number of bytes that the Myrinet driver has reserved from the pinned memory area. Standard configuration of the Myrinet driver uses 4MB of pinned memory, so this value is normally 4194304.
Coded: (default 4194304)

`NONISO_RATIO`

This value defines how much of available pinned memory to give to the noniso protocol, and therefore how much is left to the Isotach protocol. Normally noniso and Isotach protocols have equal space so this number is set to be 0.50 (i.e. 50%).
Coded: 0.50

`ISO_SLACK`

This is the amount of 'slack' to place in the host buffers, to account for packets that are not subject to flow control (i.e. Isochron Markers and EOP Markers). This number is only a rough approximation, but should be sufficient if set to at least 1.25.
Coded: 1.25

`SEND_BUF_SIZE`

This is the size of the host's send buffer in number of packets.
Coded: 1024

`ORD_SEND_BUF_SIZE`

This is the size of the host's ordered send buffer in packets.
Coded: 1024

`ISO_SEND_BUF_SIZE`

This is the size of the host's Isotach send buffer in number of packets.
Coded: 1024

`NIU_SEND_SIZE`

The size of the NIU's send buffer in number of packets.
Coded: 64

`ISO_NIU_SEND_SIZE`

The size of the NIU's Isotach send buffer in number of packets.
Coded: 64

`RECV_LIMIT`

The maximum number of packets that can be received in one call to `receive_packets()` on the LANai.
Coded: 256

`ISO_NIU_RECV_SIZE`

The size of each of the NIU's 2 Isotach receive buffers in words.
Coded: 512

`NIU_DELV_SIZE`

The size of the NIU's delivery queue. This queue stores a pointer to each received packet in pinned memory. It must be set relatively large in order to ensure that software flow control works properly. This value is also used in calculating pinned memory allocations.
Coded: 8192

`NIU_RECV_BUF_SIZE`

The total number of packets that can be buffered on the LANai before Myrinet hardware flow control has to be asserted.
Coded: 32

`isochron_allowance`

The maximum number of outstanding Isochrons that can be sent.
Coded: 256

`BUCKET_COUNT`

The number of buckets. Each bucket corresponds to a timestamp, therefore logical time wraps at 256 time units. This is set to be equal to the timestamp wraparound on the SIU (because timestamps in packets are also 1B).
Coded: 256

BUCKET_SIZE

The number of packets that can arrive in one logical time pulse. This number should be set high to prevent overflow. Generally, the number of packets that can arrive in one logical time pulse should be equal to the max number of messages sent in a single isochron. For singleton Isochrons, only 2-3 of these slots should be used at the most.

Coded: 256

Appendix D. Performance Testing Data

Latency						
Average of 500 Round-Trips in usec						
	FastMessages 1.1	Noniso		Isotach MBM		
Packet Size	w/o SIU	w/o SIU[1]	w SIU	No Ordering w/o SIU[2]	No Ordering w/ SIU[2]	Ordering w/SIU
64	25.40	29.05	31.19	35.78	67.06	67.79
128	29.60	34.14	34.18	37.15	69.67	70.61
256	41.10	37.55	39.05	42.14	74.71	75.57
512	62.10	49.49	50.91	53.82	85.34	86.40
1024	103.70	73.39	76.90	79.80	105.08	106.62

Sender Throughput						
Sending 4000000 bytes. Measured in Megabits per second (Mbps)						
	FastMessages 1.1	Noniso		Isotach MBM		
Packet Size	w/o SIU	w/o SIU[1]	w SIU	No Ordering w/o SIU[2]	No Ordering w/ SIU[2]	Ordering w/SIU
64	152.00	75.92	75.93	44.10	32.03	32.05
128	259.80	124.38	124.22	78.78	62.82	62.80
256	321.10	175.73	176.66	128.66	120.73	118.43
512	355.90	222.85	224.60	183.79	176.68	172.67
1024	359.10	247.77	251.79	228.11	222.85	219.81

Receiver Throughput						
Sending 4000000 bytes. Measured in Megabits per second (Mbps)						
	FastMessages 1.1	Noniso		Isotach MBM		
Packet Size	w/o SIU	w/o SIU[1]	w SIU	No Ordering w/o SIU[2]	No Ordering w/ SIU[2]	Ordering w/SIU
64	152.20	75.92	75.94	44.10	32.03	32.05
128	260.00	124.39	124.23	78.78	62.81	62.79
256	321.10	175.75	176.69	128.66	120.69	118.43
512	355.90	222.91	224.95	183.79	176.69	172.67
1024	359.20	247.89	251.91	228.15	222.86	219.83

[1] Without SIU indicates that the physical hardware was removed from the link

[2] Without SIU indicates that packets were sent without the SIU prefix, the hardware was in HH mode, and the TM was off

Appendix E. Configuration Files

An Isotach network is described by the network configuration file (`network.cfg`) located in the root directory. The network configuration file specifies which hosts are on the network and assigns each host a node identification number. The file also provides the route from each host to every other host. Currently, the configuration file must be "created by hand," and must be in the exact format specified below. Future additions to the system could add enhanced parsing routines for the file and/or a graphical tool to generate a network configuration file.

The format of the file is as follows:

```
#number of hosts in the network
hosts=n
#listing of hosts and their node numbers
name,0
name,1
.
.
.
name,n-1
#routes
#from node 1 to all hosts
node 1:
1,loopback
2,route
.
.
.
n-1,route
#end from node 1
.
.
.
#from node n-1 to all hosts
node n-1:
1,route
2,route
.
.
.
n-1,loopback
#end configuration file
```

Within the configuration file, any line with a '#' at the beginning signifies a comment and is ignored by the initialization routines. Furthermore, host names are specified in short form (i.e. `bugs.cs.Virginia.EDU` would be specified as `bugs`). Routes are given in hexadecimal format.

Sample `network.cfg` File:

```
#number of hosts in the network
hosts=5
#listing of hosts and their node numbers
bugs 0
foghorn 1
porky 2
fudd 3
marvin 4
#routes
#from node 0
node 0:
0 80
1 81
2 83
3 85
4 86
#from node 1
node 1:
0 BF
1 80
2 82
3 84
4 85
#from node 2
node 2:
0 BD
1 BE
2 80
3 82
4 83
#from node 3
node 3:
0 BB
1 BC
2 BE
3 80
4 81
#from node 4
node 4:
0 BA
1 BB
2 BD
3 BF
4 80
```

Appendix F. Development Environment

F.1. Directory Structure

The Isotach Ironman design calls for a modular, layered system. Thus, the source code is organized through a directory tree structure. The directory structure consists of a main directory, layer directories, and module directories. A module directory contains all of the source code necessary for that particular module, including a Makefile for that particular module. A layer directory contains module directories for each module in that layer, as well as a Makefile for that particular layer. The main directory contains the following:

- Source code for Isotach applications and the executables
- LANai Control Program (binary file)
- Isotach Library (binary file)
- API Header files
- Configuration scripts for changing system parameters
- The current network configuration file

Additionally, there are directories in the main directory that contain:

- Scripts for performance testing
- Logs for performance testing results
- Different network configuration files

It should be noted that both the Hostman and Netman layers only have a single module, so the source code and Makefiles are located directly in the layer directories.

F.2. Module Components

Every module contains 3 different source files: (module_name).c, locals.h, and exports.h. The (module_name).c file contains the implementation of every function in that module. The .c file includes locals.h, which contains all local function prototypes and variables global to that module. Additionally, this file includes all header files needed by that module. The exports.h file contains all function prototypes and data structures that the module exports to other modules. For example, every module exports an init, deinit, and poll function to Hostman. Thus, although the implementation of these functions is contained in (module_name).c, the prototypes for these functions appear in the exports.h file.

The Netman module does not export to any other module so it does not contain an exports.h file. Additionally, within this module, there is a net_utils.h header file that contains macros and functions used by the LANai Control Program.

The Hostman module does not contain an `exports.h` file as well. Instead, it provides a `utils.h` file that is included by every other module and a `host_utils.h` file that is included by every module except for Netman. The `utils.h` file contains the following:

- Queue manipulation functions
- Data types and typedefs used by all modules
- Constants representing packet types and subtypes
- Various constants used throughout all of the code (`TRUE`, `FALSE`, `SUCCESS`, `FAILURE`, etc)
- Any constants used in more than one module (masks to interpret barrier/signal bits, masks to interpret the prefix sent to the SIU's, etc)

The `host_utils.h` file contains the types of items contained in `utils.h` that are not needed on the LANai. These include data structures and constants used to describe pinned memory, number of nodes, etc. Additionally, `host_utils.h` contains several functions used for debugging purposes.

Finally, Hostman contains a file `constants.h` that stores the values for all tunable system parameters such as packet size, queue lengths, etc.

F.3. Makefiles

Every Isotach Makefile uses environment variables and relative paths whenever possible rather than explicit filenames. This allows one to place a copy of the Isotach environment anywhere on the disk, and then by changing a few environment variables, compile it and run programs. This is particularly useful for running backed up versions of the code. Additionally, all output from the Makefiles (except for error messages) is suppressed, and every Makefile uses echo statements to output what it is doing. This makes it easier to trace through a build and see exactly what is occurring.

As was mentioned, each module has its own Makefile. This Makefile is responsible for creating an object file containing the code for that particular module. The Makefile also contains the dependencies for that particular module. Finally, a module Makefile has a clean option that removes the object file and any backup (~) files. By having module level Makefiles, you can work on a particular module and check to see if it compiles without the overhead of re-compiling the entire Isotach system.

Going up a level, each layer directory has its own Makefile as well. The layer Makefile recursively calls the Makefile for each module in that particular layer. The layer Makefile also has a clean option that recursively calls each module's Makefile with the clean option. Although not entirely necessary, the layer Makefiles help provide abstraction and keep the Makefile hierarchy more intuitive.

The Hostman and Netman layer Makefiles are exceptions. The Hostman Makefile contains code to build the Hostman module, as well as build the runtime Isotach library using ranlib. The runtime library is then moved up into the main directory. Because it is responsible for building the library, the Makefile also recursive calls the Makefiles for each of the layers on the host (api, processing, niu_interface). Hostman's Makefile also provides a clean option that removes the object file, backup files, and recursively calls each layer's Makefile with the clean option.

The Netman Makefile is responsible for building the LANai Control Program. This uses the lanai3-gcc compiler. The Makefile also invokes the compiler in such a way that a special ctr0.o file is linked into the "executable" to support cards with larger memory. There is more information about this in Appendix H. Finally, the Makefile moves the compiled LANai Control Program up to the main directory. To support the LANai debugging tools (see later in this Appendix), the Makefile makes an extra copy of the control program. The clean option for this Makefile removes the control program from the main directory and removes any backup files.

Finally, there is a Makefile in the main directory. This is primarily responsible for recursively calling the other Isotach Makefiles, as well as compiling Isotach applications. We have provided a plethora of options for this Makefile, each of which is detailed below:

- **make** This option builds the Isotach applications. If the runtime library and LANai Control Program are not present in the main directory, it will recursively call the appropriate Makefiles to build them.
- **make clean** This option removes the Isotach applications and any of their backup files. It will not remove the Isotach library or the LANai Control Program.
- **make build** This option first removes the Isotach applications, and then recursively calls Hostman's and Netman's Makefiles. Remember that Hostman's Makefile calls the Makefiles of the other layers. Thus, the Isotach library and the LANai Control Program will be built. Finally, it re-builds the Isotach applications.
- **make clobber** This option removes the Isotach applications, and then recursively calls Hostman's and Netman's Makefiles with the clean option. Thus, after this is run, all executables, object files, backup files, etc. are removed from the source tree and you are left with only source code.
- **make rebuild** After typing "make clobber ; make build" so many times, we have included this option that does both for you.
- **make print_mods** This option prints a list of every module to the screen. This was initially used to generate a list of the modules to keep track of who was working on what, etc.
- **make print_src** This option prints a list of every source file to the screen. You need to call make clobber before running this option.
- **make linecount** This option takes the output of make print_src and runs it through 'wc –l'. This option has been used to track progress throughout the development cycle and to ensure that the source code doesn't get too large.
- **make create_testbed** This option runs the 'create_testbed' script that builds multiple copies of the Isotach system for performance testing.

- **make run_test** This option runs the 'run_test' script that performs virtually automated performance testing.
- **make backup_logs** This option backs up the log files from performance testing
- **make clear_logs** This option clears the log files from performance testing
- **make snapshot** This option clears all executables, object files, backup files, etc and then tars and gzips the entire source tree and moves it to a backup location.

F.4. Scripts

Throughout the course of development, several shell scripts have been developed to assist in all aspects. The following is a brief description of each script and how it is used. The create_testbed and run_test scripts source code is in Appendix I.

- **host_cfg** – The Isotach system uses a network configuration file (network.cfg) to indicate what hosts are on the network and the routes between each host. Often, there are several different configurations that the user may want to switch between. There is a directory directly below the main directory (cfg) that contains several configuration files. They are named network.desc, where 'desc' is a few letters indicating what type of configuration it is. To switch configurations, the host_cfg script can be invoked as "host_cfg desc". This copies the configuration file from the 'cfg' directory up to the main directory and renames it network.cfg.
- **sys_cfg** – There are three main parameters that are changed quite often. The first is whether a hardware SIU is supported, the second is the level of Isotach ordering supported, and the third is packet payload size. All three of these parameters are in a header file "constants.h" in the Hostman layer. The sys_cfg script uses sed to modify these values in the "constants.h" directory. The script takes two arguments: system type and packet size. System type can be one of three values: NOSIU-MBM1, SIU-MBM1, SIU-MBM2, and packet size can be any integer, however traditionally it has been either 64, 128, 256, 512, or 1024.
- **create_testbed** – When doing performance testing, one often wishes to run the same tests over multiple system configurations. This script creates a directory called "testing", directly below the main directory. Then it proceeds to build 15 copies of the Isotach system, with all combinations of the above parameters. It creates a subdirectory structure to store each one. Using this script, one can create a testbed for performance testing by typing one line and letting the system compile for approximately 15 minutes.
- **run_test** – This script sweeps through the test bed, running the performance tests for each combination of system parameters. At various stages of the test, hardware needs to be added/removed/reconfigured. The script prompts the user to perform these actions, and waits on user input before continuing. Additionally, as the Isotach hardware sometimes fails, it checks the return value of each program to ensure that it was successfully run. If the program was abnormally terminated (See the section on program termination), it will re-run that particular test. Each of the test programs writes it output to a particular log file. These log files are managed by this script as well by backing them up and clearing them before beginning the tests.

F.5. Queue Macro Functions

In the initial development of the mLayer, we realized that most of the mLayer's functionality relied on extensive queue manipulation. The previous version of Isotach also extensively used queue manipulation routines, but these routines were placed in the actual code, making the code very unreadable. We desired to have more readable code, but we also did not want the overhead of a function call for every queue manipulation. Therefore, we devised a set of generic queue manipulation macros. These macros are listed and described in detail in the file `hostman/utils.h`. These macros support two types of queues: pointer based queues and index-based queues. For small queues shared between the LANai and the host, it is more efficient to have the head/tail pointers be a one byte index into the array, rather than a four byte data pointer. The advantage is that one-byte indices do not have to be switched between Endians.

F.6. LANai Debugging Tools

Isotach is currently implemented using a gigabit switched network called Myrinet. Myrinet has many advantages, one of which is the ability to program the network card in a familiar language such as C. However, this flexibility comes with its disadvantages. The LANai (the programmable network interface) can be difficult to debug and test. However, there are some tricks and tools that we have discovered that make the process of writing a LANai Control Program (LCP) much easier.

One of the first tricks that we discovered is the equivalent of '`printf`' debugging in C. The basic concept is the following: when in doubt, print out the values of important variables during execution to trace the program. Because the LANai has no console, there is no `printf` or any output at all. Furthermore, there is no debugger available for the LANai (like gdb). In order to find out the values of variables on the LANai it is necessary to have it mapped into host memory, and have the host print out the value. This generally works, except when the host has crashed or is not functioning properly. In order to get around this problem, the following solution was devised.

First, set up a dummy host on the Myrinet network that runs a simple program to receive all bytes sent to its Myrinet card, and echo them to the host's screen. We (along with T.J. Highley) have implemented a simple program to do this called `sr` (short for send/receive). It is available in `/home2/isotach/tools` and is simple to use. Just execute `sr` in this directory, and wait to receive packets from other hosts on the network.

Next, determine the route from the malfunctioning host to the 'debug host' (i.e. the one running send/receive). In the `netman.c` source, there are two functions: `send_word` and `send_packet`. These functions take a word of data and a pointer to an array of characters as parameters, respectively. Change the first byte sent out in each of these functions to be the route to the debug host. During any point of the LANai's execution (after synchronization has completed, these two functions can be called to output information to the debug host. There are two caveats to using this method of debugging. First, because this method of debugging is invasive it may change the behavior/performance of a running program. Second, all of the data sent out is printed out by the debug host in hexadecimal format and needs to be translated to a

more readable format. As long as the programmer is careful, the first problem can be avoided and the second problem usually disappears when you find yourself reading hexadecimal just as well as decimal.

Along with the Myrinet modules and utilities are a few debugging tools that Myricom has written. These tools are available in `/home2/isotach/myrinet/bin/intel_linux`. These tools are described in detail on Myricom's web site. Of particular interest is the `mmon` application. This application is written in Tcl/tk and monitors the status of any memory-mapped variables on the LANai. Simply load up a list of the variables that you wish to monitor, and begin program execution. A list of debugging variables has already been defined in `netman/net_utils.h`. Every debugging variable on the LANai is prefaced with D_ so that it is easily identifiable in the source code.

F.7. API Design Issues

F.7.1. Files

The function prototypes and data structures that comprise the API are spread over four different header files:

`noniso.h` – contains all basic mLayer functions that the application can use, as well as all of the api functions and data structures used by the noniso protocol.

`isotach.h` – contains all of the Isotach function prototypes and data structures that are used by both the MBM and SMM models.

`iso_mbm.h` – contains function prototypes and data structures necessary to support the Isotach MBM model

`iso_smm.h` – contains function prototypes and data structures necessary to support the Isotach SMM model

F.7.2. Use

If the programmer wishes to use Non-Isotach functionality he/she must include the `noniso.h` header file. If the programmer wishes to create an Isotach program, he/she must include either `iso_mbm.h` or `iso_smm.h`, depending on which model they wish to use. Both of those header files include `isotach.h`, providing the remainder of the Isotach API. Both Isotach (either MBM or SMM) functionality and Non-Isotach (`noniso.h`) functionality can be used in the same program.

F.7.3. Preprocessor Tricks

The Ironman API provides for a single poll function, regardless of the system type. Furthermore, the open_net function does not contain any parameters indicating which type of system the programmer wishes to use. Thus, the programmer must indicate the system type by including iso_mbm/smm.h or not. The open_net function exported by Hostman actually accepts a parameter indicating the system type. Thus, based on the header files included, the preprocessor redefines the open_net symbol to pass the appropriate argument. Additionally, Hostman exports two different poll functions -- one for a strictly Non-Isotach system, and one that supports an

Isotach system. Based on which header files are included, the preprocessor redefines the poll symbol to be the appropriate poll function. Finally, there is the restriction that the programmer cannot use both the MBM and the SMM models in the same program. The preprocessor checks for this in an Isotach program by looking at which header files the programmer included. If both iso_mbm.h and iso_smm.h were included, the preprocessor generates an error.

Appendix G. Programming Conventions

This appendix will describe programming conventions used in the mLayer implementation. These conventions were followed as closely as possible to create consistency within the code and to enable the code to be readable.

G.1. Programming Style

a) The open parentheses following a function call or a control structure statement should be directly following the function name/control statement without a space in between. In addition, the brace should appear on the same line as the closing parentheses, with one space in between itself and the closing parentheses.

```
while(1) {
   …
}
```

b) Spaces should be left in between all operators and operands when possible.

```
while(i == 2)
```

c) Indent Tabs are THREE spaces, NOT five.

```
while(1) {
   i = 2;
}
```

d) All variable names should be in lowercase, with an underscore used to separate words. All Non-Isotach variables have no prefix, while all Isotach variables are prefixed with `iso_`. All defined constant names (either `#define` or `const`) should be all uppercase.

e) It is acceptable to use C shortcuts to save space and typing, but only if is immediately obvious what the statement's purpose is. If a C statement is not easily understood upon first reading, it either should have a comment describing it or be made easier to understand. (NOTE: Because efficiency is of primary concern in a messaging layer, readability must sometimes be sacrificed for efficiency.)

f) All testing variables should be prefaced with the Author's initials in capital letters, followed by an underscore. No testing variables should be left in the program for any of the program's releases. This convention allows testing variables to be removed quickly and easily.

g) All C structures (struct) should be declared using typedef to avoid having to use the keyword struct when referencing the type.

G.2. Commenting Style

Comments should be used to describe sections of code, not individual lines of code. An individual line of code should be easy enough to read and interpret, so that a comment is not warranted. Making a line of code easy to read and interpret involves using descriptive variable names and function calls. It also means that sometimes it is better to take one line and split it into several lines (unless doing so would decrease performance for some reason). However, individual lines of code can become complex due to the large number of structures and pointers that are manipulated. These lines of code should be individually commented in order to increase readability.

Each module directory contains three files. A source file for the module implementation (`module.c`), a header file for any constants, structures and global variables for the module (`locals.h`) and a header file for any constants structures and global variables that need to be exported to other modules (`exports.h`). See Appendix F.

Every C file should have a header in the following format.

```
/*
 * --------------------------------------------------------------------------
 *
 * Isotach Module  : 'Module Name' [Local Variable Definitions/Exports]
 * Isotach Layer   : 'Layer Name'
 * Isotach Version : Version 3
 *
 * --------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *          $Source:$
 *          $Revision:$
 *          $Author:$
 *          $Date:$
 *
 * --------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This should be a short description of the module's purpose and basic
 * functionality.
 *
 * --------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * Any comments about general functionality of the module (not on individual
 * functions, however) and also on data structures used in the module.
 *
 * --------------------------------------------------------------------------
 */
```

This header should start at line 1 in the source file. Directly following the header should be all `#include` statements.

All `#define` statements should be either in the `locals.h` or the `exports.h` file located in the module's directory. All module global variables should also be in the locals.h or exports.h files.

Each function should have a comment preceding it describing what the function's purpose is. This description does not have to be elaborate. The reader of this source code will use the Ironman document in conjunction with the commented source code to fully understand the messaging layer.

All comments that are transitory should be commented with a date, the programmer's name and should be made visible. These comments generally describe bugs, work to be done or unexplained behavior and should not remain in the code indefinitely.

G.3. Version Control

The Isotach Version 3 code is being controlled by CVS. The CVS root repository is at:

```
/home2/isotach/CVSROOT
```

The location of the Version 3 source tree is at:

```
/home2/isotach/CVSROOT/v3
```

To create a new repository:

Simply change directory into the checked out source directory and type the following command:

```
cvs import -m "Imported Sources" v3 Isotach start
```

This will create a subdirectory in the `CVSROOT` called `v3` and check-in all files and directories in the current directory into this subdirectory.

To checkout from the repository:

```
cvs checkout v3
```

To release a local file or directory structure:

```
cvs release v3
     // Releases v3, but does not delete the local copy
cvs release -d v3
     // Releases v3 and deletes the local copy
cvs release 'filename'
```

To add a new file to the repository:

```
cvs add 'filename'
cvs commit 'filename'
```

To check changes between a checked out copy and a file in the repository:

```
cvs diff 'filename'
```

To add modules/layers to CVS:

```
cvs checkout CVSROOT/modules
cd CVSROOT
<edit the modules file and add modules
      in using the format:
 'layer' v3/'layer'
 'module' v3/'layer'/'module'
>
cvs commit -m "Added Modules:..." modules
cd ..
cvs release -d CVSROOT
```

Appendix H. Isotach Computers and Myrinet Drivers

This document will serve as an installation and users' guide to the Isotach v3 system.

The Isotach Lab is currently composed of eight Dual Pentium III-450Mhz (256MB SDRAM) computers and one Pentium II-400Mhz (128MB SDRAM) file server. Each computer is running RedHat Linux 6.1. The machines are: `bugs`, `daffy`, `marvin`, `fudd`, `porky`, `foghorn`, `roadrunner` and `coyote`. The fileserver (where the `/home`, `/home2` and `/usr/remote` directories are stored) is `pepe`. All of these computers are in the `cs.Virginia.EDU` domain. The fileserver is backed up on departmental resources, in case of emergencies. Reinstallation of the fileserver involves reinstallation of Linux followed by reconfiguration using the configuration files stored in `/home2/config/server`.

H.1. Installation of the Isotach File Server

The fileserver (currently `pepe.cs.Virginia.EDU` at IP Address `128.143.136.220`) needs to be reinstalled manually. The files for reinstallation are in `/home2/config/server`.

To reinstall this machine, simply install an appropriate version of Red Hat Linux (>= 6.1) and follow the following instructions.

1. During installation, you will need to specify where each partition is located on a physical disk. The following is a list of drives on the machine, and their associated mount points.

```
/dev/sda1              /
/dev/sdb5              /home
/dev/sdb6              /home2
/dev/sda5              /usr
/dev/sdb1              /usr/local
/dev/sda6              swap
```

2. After installation has completed, run the following commands:

```
cp -Rfv /home2/config/server/etc /etc
rpm -Uvh /home2/config/server/RPM
cd /var/yp ; make
```

   The first command copies some configuration files into the server's `/etc` directory. These files are necessary to lock the system down from potential hackers. The second command installs/updates any Red Hat packages present in the specified directory. Please ensure that this directory is periodically updated with the newest Red Hat packages. Finally, the third command creates the YP (Yellow-Pages) databases so that the other clients can access the username/passwords on this machine. After running these commands, the system should be restarted.

3. The Myrinet drivers for the client machines should be in the `/home2/isotach/myrinet` directory. If they are not present (due to a hard disk crash), the following procedure should be following to reinstall them.

a. Obtain the Myrinet driver, `lanai3-gcc` and `lanai3-binutils` from the Myricom site or from the source files in `/home2/isotach/myrinet/source` (obtained from a tape backup).

```
M.intel_linux.325.tar.gz
M.src.325.tar.gz
lanai3-binutils-2.9.1..1.tar.gz
lanai3-gcc-2.8.1..15.tar.gz
```

b. Unzip and untar `M.src.325.tar.gz` `M.intel_linux.325.tar.gz` into `/home2/isotach/myrinet` with the following commands:

```
cd /home2/isotach
tar -zxvf M.src.325.tar.gz
tar -zxvf M.intel_linux.325.tar.gz
```

c. Two files (`myri.c` and `myri.h`) should be located in the same location as the Myrinet source files. These files have been modified to support the bigphysarea kernel patch that allows the Myrinet driver to access more than 120kb of host memory. The current configuration accesses 4MB of host memory. Copy these two files to `/home2/isotach/myrinet/src/intel_linux/module`.

d. Unzip the `lanai3-binutils` and `lanai3-gcc` archives with:

```
mkdir /home2/isotach/myrinet/source
// Move the archives to this new directory
cd /home2/isotach/myrinet/source
tar -zxvf lanai3-binutils-2.9.1..1.tar.gz
tar -zxvf lanai3-gcc-2.8.1..15.tar.gz
```

e. Run the following sequence of commands to make and install the `lanai3-gcc` and `lanai3-binutils`. The prefix directory should be `/home2/isotach/myrinet`.

```
# the binary utilities
cd lanai3-binutils-source-directory
./configure --prefix=PREFIX --target=lanai3
make -k ; make install

# lanai3 compiler
cd lanai3-gcc-source-directory
make distclean
./configure --prefix=PREFIX --target=lanai3
make -k install
```

f. Finally, the Myrinet module needs to be rebuilt with the following commands:

```
cd /home2/isotach/myrinet/src/intel_linux/module
ln -s i386 /home2/isotach/myrinet/module
make clean
make
```

NOTE: On every machine EXCEPT the fileserver, make devices must be run in this directory to create the proper entries in /dev for the Myrinet card. This can be done manually, but is usually done by the install script when a new client machine is installed.

H.2. Installation of the Isotach Client Computers

Installation of a client machine is simple and automated. The following is a list of instructions for (re)-installing an Isotach client machine.

1.) Prepare the machine first by installing an appropriate version (>= 6.1) of Red Hat Linux. The department has configured DHCP to recognize all eight of our client machines. Simply select DHCP on the network configuration screen, and the machine will automatically be configured with the proper hostname and IP address. For new machines, contact the department Systems Administrator and provide him/her with the client's hardware Ethernet address and requested DNS name. Allocate all available hard drive space to the `/` partition of type `Ext2` (except for 128MB which should be allocated to a Linux Swap partition).

2.) Boot the newly installed machine, and log in as the root user. Using FTP, retrieve the file `/home2/config/install_script` from the fileserver. Place this file in the `/root` directory on the client machine. The client installation directory (currently `/home2/config/client`) is stored in the script variable: `CONFIG`. The fileserver name (currently `pepe`) is stored in `SERVER`. This file contains a shell script, which does the following things:
   a. Edits the local `/etc/fstab` file to allow mounting of NFS directories from fileserver. (`/home`, `/home2` and `/usr/remote`). Mounts these directories.
   b. Removes unnecessary daemons from runlevel 5. These daemons are removed to remove unnecessary security holes, and to limit system processor usage. A list of removed daemons is stored in the script variable: `DAEMONS`
   c. Installs any RPM (Red Hat Packages) in the directory `$CONFIG/RPM`. These RPM's are updates to Red Hat packages installed with the OS, and should be updated, as new updates are available. There are also a few utilities included here (such as Xemacs).
   d. Configuration files are copied over from `$CONFIG/etc` to `/etc`. These files include the X11 configuration files. These configuration files are only appropriate for machines with a Voodoo Banshee video card. For other types of machines, this script will need to be customized.
   e. Each machine needs to run a patched version of the Linux kernel. The kernel version needs to match the version of the bigphysarea patch. This patch enables the Myrinet cards to use a much larger region of host memory for buffer space. Currently, we are using the 2.2.13 kernel with the appropriate bigphysarea patch. The script copies over the kernel source from an archive file, patches the kernel, and then compiles/installs it. Finally, the system's boot loader is updated to use the new kernel.
   f. Some other miscellaneous tasks are performed. These can include creating the printer spool directories and editing the `/etc/redhat-release` file. Finally, the command make devices is run in the Myrinet source directory to create entries in `/dev` for the Myrinet cards.

   NOTE: The configuration files copied over from `$CONFIG/etc` are customized to prevent unauthorized access to the Isotach lab. Please do not modify these files without consulting another system administrator. Specifically, the files `hosts_allow`, `hosts_deny`, `inetd.conf` have been edited to reduce security risks.

3.) Reboot the machine to allow the bigphysarea to be installed. Verify that bigphysarea has been installed by checking the file `/proc/bigphysarea`. This file should look similar to this (the used list should indicate that one page less than the size is utilized):

```
Big physical area, size 4100 kB
                       free list:              used list:
number of blocks:            1                       1
size of largest block:       4 kB                 4096 kB
total:                       4 kB                 4096 kB
```

This indicates that the driver is loaded and using 4096kb out of an available 4100kb.

4.) The Myrinet driver will be loaded from the `/etc/rc.d/rc.local` file with the command:

```
/sbin/insmod -v /home2/isotach/myrinet/module/myrinet
/home2/isotach/myrinet/bin/intel_linux/lload \
/home2/isotach/tools/dummy_lcp
```

The second line is necessary to clear the SRAM on the LANai. This prevents the LANai from sending out random packets on the Myrinet network. If random packets are sent out on the network, it may interfere with other functioning Isotach nodes.

5.) The Myrinet driver should be compiled to use all but one page of available bigphysarea in memory. (NOTE: We do not know why the last page of bigphysarea memory cannot be used by the Myrinet driver. This problem might require further investigation, but it is not of very high priority).

H.3. If the Myrinet Driver does not load

If the used list reads 0 and 0kb, the driver may have loaded incorrectly. If the driver is not loaded correctly, it may be trying to reserve more space than bigphysarea has. In this case, either find out how much the Myrinet driver needs and modify lilo.conf so that bigphysarea provides one more pages than requested, or recompile the Myrinet driver to request one page less than bigphysarea has available.

H.4. Modification of bigphysarea

To change the number of pages allocated on a particular machine, simply edit that machines `/etc/lilo.conf` and modify the line containing the text:

```
append="bigphysarea=1025"
```

Change the number from 1025 to any number desired, rerun lilo with the command:

```
/sbin/lilo -v
```

Reboot the system with:

```
/sbin/reboot
```

H.5. Modification of the Myrinet Drivers

The Myrinet source tree is located at (environment variable `MYRI_HOME`):

```
/home2/isotach/myrinet
```

Simply edit the file `$MYRI_HOME/src/intel_linux/module/myri.h`

To disable bigphysarea for the driver, simply comment out the line

```
#define ISOTACH
```

This will disable bigphysarea, and revert to using a 120kb copy block area allocated using `kmalloc()`.

To change the size of the requested bigphysarea modify the following lines:

```
#ifdef ISOTACH
#define COPY_BLOCK_SIZE (1024 * 4 * 1024)
```

The first number in `COPY_BLOCK_SIZE` is the number of requested pages. The second number is the page size. The third number converts from kilobytes to bytes. Just change the first number from 1024 to the number of pages available in bigphysarea less one page.

In any case, after modifying myri.h, the driver should be recompiled.

H.6. Recompilation of the Myrinet Driver

The environment variable `MYRI_HOME` should be set to the location of the Myrinet driver source tree (currently `/home2/isotach/myrinet`), in order for correct compilation of the driver to work.

Enter the directory `$MYRI_HOME/src/intel_linux/module`

```
make clean
make

cd i686
cp myrinet $MYRI_HOME/module
```

Either reload the myrinet driver manually, or reboot the computer to allow the new driver to be loaded.

H.7. Manual Reloading of the Myrinet Driver

Use the following sequence of commands:

```
su
/sbin/rmmod myrinet
/sbin/lsmod -v $MYRI_HOME/module/myrinet
```

H.8. The CRT.O Module for the LANai Compiler

The file `crt0.o` controls where the stack is placed in memory for a given compiler. The default location for the stack in the `lanai3-gcc` is at the 128kb boundary. This means the total memory available for the LCP and stack is 128kb on any Myrinet card. This is a severe limitation, considering that the current Myrinet PCI cards support 1MB of onboard memory. Normally, an application programmer will not encounter memory problems even with the 128kb restriction on memory usage. However, the data structures for this messaging layer were much larger than previous versions of Isotach messaging layers, and 128kb was not enough space for our structures. In order to solve this problem, we manually link in our own `crt0.o` file in the netman module Makefile. This is done with the following line of code in the Makefile:

```
$(CCLAN32) -o lcp -nostdlib crt0.o netman.o –lgcc
```

Currently, the stack frame begins at the 512kb boundary in memory. To change this, simply edit the file `crt0.s` and compile the module. Some common size `crt0.o` and `crt0.s` files are contained in the directory `netman/lanai`. To quickly re-configure the stack frame location, simply use the `cfg` script contained in the netman directory. This script takes a numerical value as an argument. Accepted values are: 128, 256, 512, and 1024. In order for proper functioning of the Isotach system, please do not use any values less than 512.

Appendix I. Installation and Configuration Scripts

create_testbed 133
run_test 134
install_script 137
update_clients 140

create_testbed

```
#!/bin/bash

# ********************************
# Testbed Creation Script
#
# Isotach Group
# Date 3/15/2000
# Authors: Perry Myers & Mike Lack
# ********************************
#
# This script creates a testing directory in the $ISO_HOME directory.  The
# directory structure is the following:
# $ISO_HOME/testing - main directory
# $ISO_HOME/testing/<SYS_TYPE> - 3 directories (SIU-MBM1, SIU-MBM2, NOSIU-MBM1)
# .../<SYS_TYPE>/<PKT_SIZE>    - 5 directories (64, 128, 256, 512, 1024)
#
# In each of the lowest level directories is placed all executables for that
# configuration of the Isotach System.  These executables can then be run
# for testing purposes.  This prevents excessive recompiling of the system in
# the main directory, which can be tedious and time consuming.
#
# To run this script, go to $ISO_HOME and type:
#
# make create_testbed
#

echo REMOVING CURRENT TESTING DIRECTORY!
rm -Rf $ISO_HOME/testing
echo TESTING DIRECTORY REMOVED

echo CREATING NEW TESTING DIRECTORY STRUCTURE...
mkdir $ISO_HOME/testing

#$ISO_HOME/host_cfg siu

for TYPE in SIU-MBM1 SIU-MBM2 NOSIU-MBM1;
do
   echo MAKING $TYPE TESTFILES
   mkdir $ISO_HOME/testing/$TYPE

   for PKT_SIZE in 64 128 256 512 1024;
   do
      echo MAKING $PKT_SIZE FOR $TYPE
      mkdir $ISO_HOME/testing/$TYPE/$PKT_SIZE
      $ISO_HOME/sys_cfg $TYPE $PKT_SIZE
      make -C $ISO_HOME rebuild
      for FILE in network.cfg noniso_test isotest thru latency isolatency isothru lcp;
      do
         cp $ISO_HOME/$FILE $ISO_HOME/testing/$TYPE/$PKT_SIZE
      done
   done
done

echo I MADE THIS!
```

run_test

```
#!/bin/bash

# ********************************
# Semi-Automated Testing Script
#
# Isotach Group
# Date 3/15/2000
# Authors: Perry Myers & Mike Lack
# ********************************
#
# This script was created to aid in the performance testing of the Isotach
# System.  The script performs the following tests:
# NonIsotach Throughput and Latency w/o Hardware SIU Present
# NonIsotach Throughput and Latency w/ Hardware SIU Present
# Isotach Throughput and Latency w/ Hardware SIU Present, but no host ordering
# Isotach Throughput and Latency w/ Harware SIU Present and host ordering
#
# The test proceeds in three phases:
# 1. Fully automated testing of NonIso w/o SIU's.
# 2. Fully automated testing of NonIso/Iso w/ SIU's and no host ordering.
# 3. Semi-Automated testing of Iso w/ SIU's and host ordering.
#
# Between each of the three phases, user input is required to proceed to the
# next phase.  The third phase requires user intervention between each
# individual test.  This is because the hardware is not 100% reliable, and it
# may be necessary to re-run some tests.  After each test in Phase 3 is run,
# the operator will be prompted to either continue with testing, or re-run the
# previous test.  The operator may re-run tests as many times as he/she
# desires.
#
# Each test sends output to files in $ISO_HOME/logs
# The files are:
# latency.log      - Average Latency measured on the Server Side.
# isolatency.log   - Average Isotach Latency measured on the Server Side.
# thru_send.log    - Average Sender Throughput using NonIsotach
# thru_recv.log    - Average Receiver Throughput using NonIsotach
# isothru_send.log - Average Sender Throughput using Isotach
# isothru_recv.log - Average Receiver Throughput using Isotach
#
# To run the performance tests, simply cd to the $ISO_HOME directory and
# type the following:
#
# make run_tests
#
# NOTE: This script will only work if there is a valid testing directory.
# To create a testing directory, run:
#
# make create_testbed
#

LOGS=$ISO_HOME/logs
NODE_NAME=""
NODE_NUM=""


function nohardware_tests() {

if [ "$NODE_NUM" = "0" ] ; then
echo
         echo "The following were without the hardware:" >> $LOGS/latency.log
         echo "The following were without the hardware:" >> $LOGS/thru_send.log
         echo "The following were without the hardware:" >> $LOGS/thru_recv.log
fi

for PKT_SIZE in 64 128 256 512 1024;
   do
      cd  $ISO_HOME/testing/NOSIU-MBM1/$PKT_SIZE
      thru
      sleep 1
      latency
      sleep 1
```

```
done

if [ "$NODE_NUM" = "0" ] ; then
echo
         echo "End without hardware" >> $LOGS/latency.log
         echo "End without hardware" >> $LOGS/thru_send.log
         echo "End without hardware" >> $LOGS/thru_recv.log
fi
}

function nosiu_tests() {
for PKT_SIZE in 64 128 256 512 1024;
   do
      cd  $ISO_HOME/testing/NOSIU-MBM1/$PKT_SIZE
      RETVAL=1
      until [ "$RETVAL" = "0" ] ; do
          thru
          RETVAL=$?
          sleep 1
      done
      RETVAL=1
      until [ "$RETVAL" = "0" ] ; do
          latency
          RETVAL=$?
          sleep 1
      done
      RETVAL=1
      until [ "$RETVAL" = "0" ] ; do
          isothru
          RETVAL=$?
          sleep 1
      done
      RETVAL=1
      until [ "$RETVAL" = "0" ] ; do
          isolatency
          RETVAL=$?
          sleep 1
      done
done
}


function siu_tests() {
for TYPE in SIU-MBM1 SIU-MBM2;
do
   for PKT_SIZE in 64 128 256 512 1024;
   do
      cd  $ISO_HOME/testing/$TYPE/$PKT_SIZE
      RETVAL=1
      until [ "$RETVAL" = "0" ] ; do
          isothru
          RETVAL=$?
          sleep 2
      done
      RETVAL=1
      until [ "$RETVAL" = "0" ] ; do
          isolatency
          RETVAL=$?
          sleep 2
      done
   done
done
}


function chill() {
if [ "$NODE_NUM" = "0" ] ; then
    echo -e "\n\aPress any key to continue..."
    read RESPONSE
    touch $ISO_HOME/go
fi

if [ "$NODE_NUM" != "0" ] ; then
    echo -e "\nWaiting on user input"
```

```
    until [ -f $ISO_HOME/go ] ; do
          FOO=1
    done
    rm $ISO_HOME/go
fi
}

#here is the main part of the script
clear

NODE_NAME=`hostname -s`
NODE_NUM=`grep $NODE_NAME $ISO_HOME/network.cfg | sed -e s/$NODE_NAME// -e 's/ //'`


echo "Welcome to the wonderful world of Isotach Performance Testing..."
echo
echo "This script is running on $NODE_NAME($NODE_NUM)"
echo

if [ "$NODE_NUM" = "0" ] ; then
    echo "Clearing out the previous logs (hope you backed them up...)"
    make -C $ISO_HOME backup_logs
    make -C $ISO_HOME clear_logs
    echo
fi

echo "Please disconnect the hardware SIU's"


#chill
#nohardware_tests


clear
echo "Finished running tests without the hardware."
echo "Please connect the hardware SIU's."
echo "Make sure that they are in Host-Host mode"
echo "and that the token manager is off"


chill
nosiu_tests

clear
echo "Finished running tests that did not use the hardware SIU"
echo "Please set the SIU's to SIU mode, reset them, and"
echo "power up the token manager"
echo "Once we have a yellow light, we will begin running SIU tests"


chill
siu_tests

echo I MADE THIS!
```

install_script

```
#!/bin/bash

# Script Variables - Please Change as Necessary
SERVER=pepe
CONFIG=/home2/config/client
DAEMONS='*apmd *pcmcia *sendmail *kudzu *linuxconf *nfs'
KERNEL_VER=2.2.13
KCFG_DATE=2.29.2000
MYRI_HOME=/home2/isotach/myrinet

echo '************************************************************'
echo CLIENT CONFIGURATION SCRIPT
echo '************************************************************'

echo
echo '************************************************************'
echo Removing original /home directory
echo '************************************************************'

umount /home
rm -Rfv /home

echo
echo '************************************************************'
echo Making /home, /home2 and /usr/remote directories
echo '************************************************************'

mkdir --verbose /home
mkdir --verbose /home2
mkdir --verbose /usr/remote

echo
echo '************************************************************'
echo Editing /etc/fstab file
echo '************************************************************'

sed -e '${' \
    -e 'a\' \
    -e $SERVER':/home                /home                        nfs      defaults\' \
    -e $SERVER':/home2               /home2                       nfs      defaults\' \
    -e $SERVER':/usr/remote  /usr/remote                  nfs     defaults'  \
    -e '}' /etc/fstab > /etc/fstab.new

cp -v /etc/fstab /etc/fstab.old
cp -v /etc/fstab.new /etc/fstab

echo
echo '************************************************************'
echo Mounting NFS directories
echo '************************************************************'

mount -v /home
mount -v /home2
mount -v /usr/remote

echo
echo '************************************************************'
echo Removing unnecessary daemons from startup
echo '************************************************************'

cd /etc/rc.d/rc5.d
rm -fv $DAEMONS
cd /etc

echo
echo '************************************************************'
echo Installing Updates and Software
echo '************************************************************'

rpm -Uvh --force $CONFIG/RPM/*
```

```
echo
echo '************************************************************'
echo Copying over Configuration files from Server
echo '************************************************************'

cp -vR $CONFIG/etc/* /etc

echo
echo '************************************************************'
echo Copying over Kernel Sources
echo '************************************************************'

cd /usr/src
rm -fv /usr/src/linux
rm -Rf /usr/src/linux-$KERNEL_VER
rm -Rf /usr/src/bigphysarea-$KERNEL_VER

tar -zxf $CONFIG/linux-$KERNEL_VER.tar.gz
mv -v linux linux-$KERNEL_VER
cp -R $CONFIG/bigphysarea-$KERNEL_VER /usr/src
cp -v $CONFIG/kernelconfig-$KCFG_DATE /usr/src/linux-$KERNEL_VER

ln -sv /usr/src/linux-$KERNEL_VER /usr/src/linux
/usr/bin/patch -p0 < /usr/src/bigphysarea-$KERNEL_VER/bigphysarea-patch

cd /usr/src/linux
make mrproper
make clean
cp -v kernelconfig-$KCFG_DATE .config
make oldconfig
make dep
make bzImage
make modules
make modules_install

rm -fv /boot/vmlinuz
rm -fv /boot/System.map

cp -v /usr/src/linux/arch/i386/boot/bzImage /boot/vmlinuz-$KERNEL_VER
cp -v /usr/src/linux/System.map /boot/System.map-$KERNEL_VER

ln -sv /boot/System.map-$KERNEL_VER /boot/System.map
ln -sv /boot/vmlinuz-$KERNEL_VER /boot/vmlinuz

/sbin/mkinitrd /boot/initrd-2.2.13.img 2.2.13

echo
echo '************************************************************'
echo Editing /etc/lilo.conf and running lilo
echo '************************************************************'

sed -e '/default=linux/{'                          \
    -e 'a\'                                        \
    -e 'append="bigphysarea=1025"\'                \
    -e '                                        \' \
    -e 'image=/boot/vmlinuz-2.2.13\'               \
    -e '        label=linux\'                      \
    -e '        initrd=/boot/initrd-2.2.13.img\'   \
    -e '        read-only\'                        \
    -e '        root=/dev/hda1'                    \
    -e '}'                                         \
    -e 's/label=linux/label=linux-old/' /etc/lilo.conf > /etc/lilo.conf.new

cp -v /etc/lilo.conf /etc/lilo.conf.old
cp -v /etc/lilo.conf.new /etc/lilo.conf

/sbin/lilo -v

echo
echo '************************************************************'
echo Creating printer spool directories
echo '************************************************************'
```

```
grep -v -e ':[^\\]' -e '#' /etc/printcap | grep '\w' > temp
sed -e 's/:\\/ /' temp > temp2
mkdir `cat temp2`
rm temp temp2

echo
echo '***********************************************************'
echo Modifying /etc/issue
echo '***********************************************************'

cat /etc/redhat-release > /etc/issue
echo Kernel `uname -r` on an `uname -m` >> /etc/issue

echo
echo '***********************************************************'
echo Creating /dev entries for Myrinet Drivers
echo '***********************************************************'

make -C $MYRI_HOME/src/intel_linux/module devices

echo
echo '***********************************************************'
echo FINISHED!!!
echo '***********************************************************'
```

update_clients

```
#!/bin/bash

# 1st Parameter is a list of files to install into /etc
# 2nd Parameter is a list of scripts to run

CONFIG_DIR=/home2/config/client

for CLIENT in `cat $CONFIG_DIR/../clients`;
do
   echo $CLIENT

   for FILE in $1;
   do
      ssh $CLIENT "cp -vR $CONFIG_DIR/etc/$FILE /etc";
   done

   for FILE in $2;
   do
      ssh $CLIENT "cp -vR $CONFIG_DIR/var/$FILE /var";
   done

   for SCRIPT in $3;
   do
      ssh $CLIENT "$CONFIG_DIR/scripts/$SCRIPT";
   done

   case $# in
   4)
      ssh $CLIENT $4;
   esac
done
```

Appendix J. Isotach Source Code

noniso_test.c 143
latency.c 145
thru.c 148
isotest.c 151
isolatency.c 153
isothru.c 156
noniso.h 159
isotach.h 162
iso_mbm.h 165
iso_smm.h 166
hostman/constants.h 167
hostman/utils.h 169
hostman/host_utils.h 175
hostman/locals.h 179
hostman/hostman.c 181
api/send/exports.h 197
api/send/locals.h 198
api/send/send.c 199
api/deliver/exports.h 201
api/deliver/locals.h 202
api/deliver/deliver.c 203
api/barrier/exports.h 205
api/barrier/locals.h 206
api/barrier/barrier.c 207
api/iso_send/exports.h 210
api/iso_send/locals.h 211
api/iso_send/iso_send.c 212
api/iso_deliver/exports.h 217
api/iso_deliver/locals.h 219
api/iso_deliver/iso_deliver.c 220
api/iso_barrier/exports.h 222
api/iso_barrier/locals.h 223
api/iso_barrier/iso_barrier.c 224
api/iso_signal/exports.h 229
api/iso_signal/locals.h 230
api/iso_signal/iso_signal.c 231
api/iso_retrieve/exports.h 235
api/iso_retrieve/locals.h 236
api/iso_retrieve/iso_retrieve.c 237
processing/flow/exports.h 238
processing/flow/locals.h 239

processing/flow/flow.c 241
processing/iso_flow/exports.h 245
processing/iso_flow/locals.h 246
processing/iso_flow/iso_flow.c 248
processing/iom/exports.h 252
processing/iom/locals.h 253
processing/iom/iom.c 255
processing/shmem/exports.h 261
processing/shmem/locals.h 262
processing/shmem/shmem.c 263
niu_interface/shipping/exports.h 264
niu_interface/shipping/locals.h 266
niu_interface/shipping/shipping.c 267
niu_interface/receive/exports.h 271
niu_interface/receive/locals.h 272
niu_interface/receive/receive.c 273
niu_interface/iso_shipping/exports.h 276
niu_interface/iso_shipping/locals.h 277
niu_interface/iso_shipping/iso_shipping.c 278
niu_interface/iso_receive/exports.h 281
niu_interface/iso_receive/locals.h 282
niu_interface/iso_receive/iso_receive.c 283
netman/net_utils.h 286
netman/locals.h 289
netman/netman.c 292

noniso_test.c

```
#include <stdio.h>
#include <string.h>
#include <sys/time.h>
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include "noniso.h"

static inline void subtracttime(struct timeval *t, struct timeval *sub)
{
  signed long sec,usec;

  sec = t->tv_sec - sub->tv_sec;
  usec = t->tv_usec-sub->tv_usec;
  if (usec < 0) {
    sec--;
    usec+=1000000;
  }
  if (sec < 0) {
    t->tv_sec = 0;
    t->tv_usec = 0;
  }
  else {
    t->tv_sec = (unsigned long)sec;
    t->tv_usec = (unsigned long)usec;
  }
}



int main(int argc, char *argv[]) {
  int my_node_id;
  int i, j;
  char msg1[1024];
  noniso_mbm message;
  int ITERATIONS = 1000;
  int PRINT;
  struct timeval start, stop;
  double us;
  int receives = 0;
  int pkt_number = 0;
  char *msg;

  if (argc >= 2)
    ITERATIONS = atoi(argv[1]);
  if (argc >= 3)
    PRINT      = atoi(argv[2]);
  else if (ITERATIONS >= 10)
    PRINT      = ITERATIONS / 10;
  else
    PRINT      = 10;

  memset(msg1,' ',1024);
  msg1[1023] = '\0';


  open_net();
  gettimeofday (&start, NULL);


  my_node_id = get_my_node_number();

  printf("Starting Application.\n");


  for (i = 0; i < ITERATIONS; i++) {
    sprintf(msg1, "Sender->%02d %09d", my_node_id, i);

    for (j = 0; j < get_number_of_hosts(); j++) {
```

```
      if (j != my_node_id) {
          do {
            if (receive(&message) == 0) {
              msg = (char *)message.msg.data_ptr;
              //        pkt_number = atoi((char *)&message.msg.data_ptr[11]);
              pkt_number = atoi(&msg[11]);
              /* if (pkt_number != receives) {
                printf("Received a packet out of order.\n");
                printf("Expecting %d, got %d\n",receives, pkt_number);
                exit(1);
                }*/
              if (((pkt_number % PRINT) == 0))// || (pkt_number > 99900))
                printf("\tGOT PACKET %.20s\n", (char *)message.msg.data_ptr);
              receives++;
            }
            //      } while(send(j, (void *)msg1, strlen(msg1) + 1) != 0);
          } while(send(j, (void *)msg1, 64) != 0);
      }
    }
  }
  printf("Receiving\n");

  while (try_close_net() != 0) {
    //  while(receives < (ITERATIONS * (get_number_of_hosts() - 1))) {
  //while(1) {
    if (receive(&message) == 0) {
      msg = (char *)message.msg.data_ptr;
      //      pkt_number = atoi((char *)&message.msg.data_ptr[11]);
      pkt_number = atoi(&msg[11]);

      /*      if (pkt_number != receives) {
                printf("Received a packet out of order.\n");
                printf("Expecting %d, got %d\n",receives, pkt_number);
                exit(1);
                }*/
      if (((pkt_number % PRINT) == 0))// || (pkt_number > 99900))
          printf("\tGOT PACKET %s\n", (char *)message.msg.data_ptr);
      receives++;
    }
  }

  gettimeofday (&stop, NULL);
  subtracttime (&stop, &start);
  us = (double)stop.tv_sec *1e6 + (double)stop.tv_usec;
  printf("Sent and received %d packets in %.0f usec\n",ITERATIONS,us);

  printf("I made this!\n");

  return 0;
}
```

latency.c

```c
/*
 * fm_lat - latency tester for FM
 */

#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <noniso.h>
#include <time.h>

#define COUNT 50    /* # times to run test */
#define CLIENT 1    /* nodeid of client */
#define SERVER 0    /* nodeid of server */

char FILENAME[128];
#define LOG_FILE strcat(strcpy(FILENAME, getenv("ISO_HOME")), "/logs/latency.log")

int BUF_SIZE;
#define REPS 500

char* buf;
char* usr_buffer;


static inline void subtracttime(struct timeval *t, struct timeval *sub)
{
  signed long sec,usec;

  sec = t->tv_sec - sub->tv_sec;
  usec = t->tv_usec-sub->tv_usec;
  if (usec < 0) {
    sec--;
    usec+=1000000;
  }
  if (sec < 0) {
    t->tv_sec = 0;
    t->tv_usec = 0;
  }
  else {
    t->tv_sec = (unsigned long)sec;
    t->tv_usec = (unsigned long)usec;
  }
}


int main(int argc, char *argv[]) {
  struct timeval start, stop;
  double us, us_sum = 0;
  int i,j;
  noniso_mbm message;
  int sent, received;
  FILE *fp;

  time_t curtime;
  char date[50];
  struct tm *tp;

  if (argc < 2)
    BUF_SIZE = get_max_payload();
  else
    BUF_SIZE = atoi(argv[1]);


  open_net();

  buf = (char*) malloc (BUF_SIZE);
  usr_buffer = (char *)malloc(BUF_SIZE);
```

```
bzero (buf, BUF_SIZE);

if (get_my_node_number() == SERVER) {
  for (i=0; i<COUNT; i++) {
    received = 0;
    for (j=0; j< REPS; j++) {
        if (!received) {
          while(receive(&message)!=0) ;
          memcpy(usr_buffer,message.msg.data_ptr,BUF_SIZE);
          received = 1;
        }
        else {
          while(send(CLIENT,buf,BUF_SIZE)!=0) ;
          received = 0;
        }
    }
  }
  while(try_close_net() != 0)
    poll();
  printf("done\n");
}
else { /* client */

  /* each loop is a test; each test executes REPS # of msg round trips; */
  for (i=0; i<COUNT; i++) {
    sent = 0;
    gettimeofday (&start, NULL);
    for (j=0; j< REPS; j++) {
        if (!sent) {
          while(send(SERVER,buf,BUF_SIZE)!=0) ;
          sent = 1;
        }
        else {
          while(receive(&message)!=0) ;
          memcpy(usr_buffer,message.msg.data_ptr,BUF_SIZE);
          sent = 0;
        }
    }
    gettimeofday (&stop, NULL);
    subtracttime (&stop, &start); /* defined in prof.h */
    us = (double)stop.tv_sec *1e6 + (double)stop.tv_usec;
    printf ("executed %d round trips in %f usecs; %f us/round trip\n",
              REPS, us, us/REPS);
    fflush (stdout);
    us_sum += us;
  }
  printf("Average round trip latency: %f us\n", us_sum/(REPS*COUNT) );
  while(try_close_net() != 0)
    poll();
  printf("done\n");


  fp = fopen(LOG_FILE,"a");
  if (fp == 0) {
    printf("can't open file\n");
    return 1;
  }

/* get the current date */
  time(&curtime);
  tp = localtime(&curtime);
  strftime(date, (size_t) 50, "%a %b %d %R", tp);

  fprintf(fp,"%s\tNonIsotach\t%d",date,get_max_payload());
  if (get_SIU_state() == 1) {
    fprintf(fp,"\tSIU");
  }
  else {
    fprintf(fp,"\tnoSIU ");
  }
  if (get_MBM_ver() == 1) {
    fprintf(fp,"\tnoIOM");
  }
  else {
```

```
      fprintf(fp,"\tIOM");
    }
    fprintf(fp,"\t%.02f us\n", us_sum/(REPS*COUNT) );
    fclose(fp);

  }
  return 0;
}
```

thru.c

```
/*
 * fm_thru - throughput tester for FM
 */

#include <stdio.h>
#include <sys/time.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <noniso.h>
#include <time.h>

#define CLIENT 1    /* nodeid of client */
#define SERVER 0    /* nodeid of server */
int BUF_SIZE; /* size of buffer to be sent in messages */
#define DATA_SIZE 4000000  /* amount of data to shove into the pipe, in units of bytes */
#define ITERATIONS 5  /* number of times to run the test */

char FILENAME[128];
#define SLOG_FILE strcat(strcpy(FILENAME, getenv("ISO_HOME")), "/logs/thru_send.log")
#define RLOG_FILE strcat(strcpy(FILENAME, getenv("ISO_HOME")), "/logs/thru_recv.log")

struct timeval start, stop;
long received_bytes;
long sent_bytes;
char* buf;
char* data;


static inline void subtracttime(struct timeval *t, struct timeval *sub) {
  signed long sec,usec;

  sec = t->tv_sec - sub->tv_sec;
  usec = t->tv_usec-sub->tv_usec;
  if (usec < 0) {
    sec--;
    usec+=1000000;
  }
  if (sec < 0) {
    t->tv_sec = 0;
    t->tv_usec = 0;
  }
  else {
    t->tv_sec = (unsigned long)sec;
    t->tv_usec = (unsigned long)usec;
  }
}

int main(int argc, char *argv[]) {

  double sec;
  double throughput;
  double throughput_sum = 0;
  double avg_throughput = 0;

  int x;
  noniso_mbm message;

  FILE *fp;

  time_t curtime;
  char date[50];
  struct tm *tp;


  if (argc < 2)
    BUF_SIZE = get_max_payload();
  else
    BUF_SIZE = atoi(argv[1]);
```

```
  buf = (char*) malloc (BUF_SIZE);
  data = (char*) malloc (BUF_SIZE);

  bzero (buf, BUF_SIZE);

  open_net();



  for (x = 0; x < ITERATIONS; x++) {
    sec = 0;
    throughput = 0;
    initiate_barrier();
    while (barrier_completed() != 0) ;

    if (get_my_node_number() == SERVER) {
      received_bytes = 0;
      while (receive(&message) != 0);
      gettimeofday (&start, NULL);
      memcpy( (void*) buf, (const void*) data, message.size);
      received_bytes += message.size;

      while (received_bytes < DATA_SIZE) {
          while (receive(&message) != 0);
          memcpy( (void*) buf, (const void*) data, message.size);
          received_bytes += message.size;
      }

      gettimeofday (&stop, NULL);
    }
    else { /* client */
      sent_bytes = 0;
      gettimeofday (&start, NULL);
      while(send(SERVER,buf,BUF_SIZE)!=0);
      sent_bytes += BUF_SIZE;
      while (sent_bytes < DATA_SIZE) {
          while(send(SERVER,buf,BUF_SIZE)!=0);
          sent_bytes += BUF_SIZE;
      }
      gettimeofday (&stop, NULL);
    }

    subtracttime (&stop, &start); /* defined in prof.h; puts result in stop */
    sec = (double)stop.tv_sec + (double)stop.tv_usec / 1e6;

    if (get_my_node_number() == SERVER) {
      throughput = ((received_bytes * 8) / sec) / 1e6; /* in Mbps */
      printf("Received %ld bytes in %f seconds\n", received_bytes, sec);
      printf("Measured receiver throughput for iteration %d is %f Mbps\n",
               x, throughput);
    }
    else { /* CLIENT */
      throughput = ((sent_bytes * 8) / sec) / 1e6; /* in Mbps */
      printf("Sent %ld bytes in %f seconds\n", sent_bytes, sec);
      printf("Measured sender throughput for iteration %d is %f Mbps\n",
               x, throughput);
    }
    fflush (stdout);
    throughput_sum += throughput;
  }
  avg_throughput = throughput_sum / ITERATIONS;
  printf("Average %s throughput is %f\n", get_my_node_number() ? "sender":"receiver",
avg_throughput);

  while(try_close_net() != 0)
    poll();

  if (get_my_node_number() == SERVER)
    fp = fopen(RLOG_FILE,"a");
  else
    fp = fopen(SLOG_FILE,"a");

  /* get the current date */
    time(&curtime);
```

```
    tp = localtime(&curtime);
    strftime(date, (size_t) 50, "%a %b %d %R", tp);

  fprintf(fp,"%s\tNonIsotach\t%d",date,get_max_payload());
  if (get_SIU_state() == 1) {
    fprintf(fp,"\tSIU");
  }
  else {
    fprintf(fp,"\tnoSIU ");
  }
  if (get_MBM_ver() == 1) {
    fprintf(fp,"\tnoIOM");
  }
  else {
    fprintf(fp,"\tIOM");
  }
  fprintf(fp,"\t%.02f Mbps\n", avg_throughput );
  fclose(fp);




  return 0;
}
```

isotest.c

```c
#include <stdio.h>
#include <string.h>
#include <sys/time.h>
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include "iso_mbm.h"
#include "noniso.h"


int main(int argc, char *argv[]) {
  int my_node_id;
  int i, j;
  char msg1[1024];
  iso_mbm message;
  int ITERATIONS = 100000;
  int PRINT;
  int receives = 0;
  int pkt_number;
  int sends = 0;
  int end = 0;
  char *msg;

  if (argc >= 2)
    ITERATIONS = atoi(argv[1]);
  if (argc >= 3)
    PRINT      = atoi(argv[2]);
  else if (ITERATIONS > 10)
    PRINT      = ITERATIONS / 10;
  else
    PRINT      = 1;

  if ((ITERATIONS == 0) && (argc < 3))
    PRINT = 1;


  memset(msg1,' ',1024);
  msg1[1023] = '\0';

  open_net();

  my_node_id = get_my_node_number();

  printf("Starting Application.\n");

  for (i = 0; i < ITERATIONS; i++) {
    sprintf(msg1, "Sender->%02d %09d", my_node_id, i);
    if (i == (ITERATIONS - 1)) end = 1;

    for (j = 0; j < get_number_of_hosts(); j++) {
      if (j != my_node_id) {
          do {
            if (iso_receive(&message) == 0) {
              msg = (char *)message.info.msg.data;
              pkt_number = atoi(&msg[11]);

              if (((pkt_number % PRINT) == 0)) {
                printf("\tGOT PACKET %.20s\n", (char *)message.info.msg.data);
              }
              receives++;
            }
          } while (iso_send(j, (void *)msg1, 64,1) != 0);
          //        } while (iso_send(j, (void *)msg1, strlen(msg1)+1,1) != 0);
          //        } while (iso_send(j, (void *)msg1, strlen(msg1) + 1,end) != 0);
          if (end == 1) end = 0;
          if ((++sends % 1)==0) end = 1;
      }
    }
  }
```

```
  for (i=0; i<10000; i++)
    poll();


  printf("Receiving\n");

  while((receives < (ITERATIONS * (get_number_of_hosts() - 1))) ||
          (ITERATIONS == 0)) {
    //  while(!finished) {
    if (iso_receive(&message) == 0) {
      msg = (char *)message.info.msg.data;
      pkt_number = atoi(&msg[11]);

      if (((pkt_number % PRINT) == 0)) {
          printf("\tGOT PACKET %s\n", (char *)message.info.msg.data);
      }
      receives++;
    }
  }

  printf("Received %d packets\n",receives);

  while (try_close_net() == FAILURE);
  printf("I made this!\n");

  return 0;
}
```

isolatency.c

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <iso_mbm.h>
#include <time.h>


#define COUNT 50    /* # times to run test */
#define CLIENT 1    /* nodeid of client */
#define SERVER 0    /* nodeid of server */

char FILENAME[128];
#define LOG_FILE strcat(strcpy(FILENAME, getenv("ISO_HOME")), "/logs/isolatency.log")

#define REPS 500
int BUF_SIZE;
char* buf;
char* usr_buffer;


static inline void subtracttime(struct timeval *t, struct timeval *sub)
{
  signed long sec,usec;

  sec = t->tv_sec - sub->tv_sec;
  usec = t->tv_usec-sub->tv_usec;
  if (usec < 0) {
    sec--;
    usec+=1000000;
  }
  if (sec < 0) {
    t->tv_sec = 0;
    t->tv_usec = 0;
  }
  else {
    t->tv_sec = (unsigned long)sec;
    t->tv_usec = (unsigned long)usec;
  }
}


int main(int argc, char *argv[]) {
  struct timeval start, stop;
  double us, us_sum = 0;
  int i,j;
  iso_mbm message;
  int sent, received;

  time_t curtime;
  char date[50];
  struct tm *tp;



  FILE *fp;
  if (argc < 2)
    BUF_SIZE = get_max_payload();
  else
    BUF_SIZE = atoi(argv[1]);


  open_net();

  buf = (char*) malloc (BUF_SIZE);
  usr_buffer = (char *)malloc(BUF_SIZE);
  bzero (buf, BUF_SIZE);
```

```
if (get_my_node_number() == SERVER) {
  for (i=0; i<COUNT; i++) {
    received = 0;
    for (j=0; j< REPS; j++) {
        if (!received) {
          while(iso_receive(&message)!=0) ;
          memcpy(usr_buffer,message.info.msg.data,BUF_SIZE);
          received = 1;
        }
        else {
          while(iso_send(CLIENT,buf,BUF_SIZE,1)!=0) ;
          received = 0;
        }
    }
  }
  /*while(try_close_net() != 0)
    poll();*/
  printf("done\n");
}
else { /* client */
  /* each loop is a test; each test executes REPS # of msg round trips; */
  for (i=0; i<COUNT; i++) {
    sent = 0;
    gettimeofday (&start, NULL);
    for (j=0; j< REPS; j++) {
        if (!sent) {
          while(iso_send(SERVER,buf,BUF_SIZE,1)!=0) ;
          sent = 1;
        }
        else {
          while(iso_receive(&message)!=0) ;
          memcpy(usr_buffer,message.info.msg.data,BUF_SIZE);
          sent = 0;
        }
    }
    gettimeofday (&stop, NULL);
    subtracttime (&stop, &start); /* defined in prof.h */
    us = (double)stop.tv_sec *1e6 + (double)stop.tv_usec;
    printf ("executed %d round trips in %f usecs; %f us/round trip\n",
              REPS, us, us/REPS);
    fflush (stdout);
    us_sum += us;
  }
  printf("Average round trip latency: %f us\n", us_sum/(REPS*COUNT) );
  /* while(try_close_net() != 0)
          poll(); */
  printf("done\n");
  fp = fopen(LOG_FILE,"a");
  if (fp == 0) {
    printf("can't open file\n");
    return 1;
  }

  /* get the current date */
  time(&curtime);
  tp = localtime(&curtime);
  strftime(date, (size_t) 50, "%a %b %d %R", tp);


  fprintf(fp,"%s\tIsotach\t%d",date,get_max_payload());
  if (get_SIU_state() == 1) {
    fprintf(fp,"\tSIU");
  }
  else {
    fprintf(fp,"\tnoSIU");
  }
  if (get_MBM_ver() == 1) {
    fprintf(fp,"\tnoIOM");
  }
  else {
    fprintf(fp,"\tIOM");
  }
  fprintf(fp,"\t%.02f us\n", us_sum/(REPS*COUNT) );
  fclose(fp);
```

```
  }
  return 0;
}
```

isothru.c

```
#include <stdio.h>
#include <sys/time.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <iso_mbm.h>
#include <noniso.h>
#include <time.h>

#define CLIENT 0    /* nodeid of client */
#define SERVER 1    /* nodeid of server */
int BUF_SIZE; /* size of buffer to be sent in messages */
#define DATA_SIZE 40000000  /* amount of data to shove into the pipe, in units of bytes */
#define ITERATIONS 5  /* number of times to run the test */

char FILENAME[128];
#define SLOG_FILE strcat(strcpy(FILENAME, getenv("ISO_HOME")), "/logs/isothru_send.log")
#define RLOG_FILE strcat(strcpy(FILENAME, getenv("ISO_HOME")), "/logs/isothru_recv.log")

struct timeval start, stop;
long received_bytes;
long sent_bytes;
char* buf;
char* data;


static inline void subtracttime(struct timeval *t, struct timeval *sub) {
  signed long sec,usec;

  sec = t->tv_sec - sub->tv_sec;
  usec = t->tv_usec-sub->tv_usec;
  if (usec < 0) {
    sec--;
    usec+=1000000;
  }
  if (sec < 0) {
    t->tv_sec = 0;
    t->tv_usec = 0;
  }
  else {
    t->tv_sec = (unsigned long)sec;
    t->tv_usec = (unsigned long)usec;
  }
}

int main(int argc, char *argv[]) {

  double sec;
  double throughput;
  double throughput_sum = 0;
  double avg_throughput = 0;

  int x;
  iso_mbm message;

  time_t curtime;
  char date[50];
  struct tm *tp;


  FILE *fp;

  if (argc < 2)
    BUF_SIZE = get_max_payload();
  else
    BUF_SIZE = atoi(argv[1]);

  buf = (char*) malloc (BUF_SIZE);
  data = (char*) malloc (BUF_SIZE);
```

```
  bzero (buf, BUF_SIZE);

  open_net();


  for (x = 0; x < ITERATIONS; x++) {
    sec = 0;
    throughput = 0;
    initiate_barrier();
    while (barrier_completed() != 0) poll();

    if (get_my_node_number() == SERVER) {
      received_bytes = 0;
      while (iso_receive(&message) != 0);
      gettimeofday (&start, NULL);
      memcpy( (void*) buf, (const void*) data, message.info.msg.length);
      received_bytes += message.info.msg.length;

      while (received_bytes < DATA_SIZE) {
          while (iso_receive(&message) != 0);
          memcpy( (void*) buf, (const void*) data, message.info.msg.length);
          received_bytes += message.info.msg.length;
          /*        if ((received_bytes % (1000*1024)) == 0)
            printf("receoved %d bytes\n",received_bytes);

          if (received_bytes > DATA_SIZE - BUF_SIZE)
          printf("waiting for last packet\n");*/

      }

      gettimeofday (&stop, NULL);
    }
    else { /* client */
      sent_bytes = 0;
      gettimeofday (&start, NULL);
      while(iso_send(SERVER,buf,BUF_SIZE,1)!=0);
      sent_bytes += BUF_SIZE;
      while (sent_bytes < DATA_SIZE) {
          while(iso_send(SERVER,buf,BUF_SIZE,1)!=0);
          sent_bytes += BUF_SIZE;
      }
      gettimeofday (&stop, NULL);
    }

    subtracttime (&stop, &start); /* defined in prof.h; puts result in stop */
    sec = (double)stop.tv_sec + (double)stop.tv_usec / 1e6;

    if (get_my_node_number() == SERVER) {
      throughput = ((received_bytes * 8) / sec) / 1e6; /* in Mbps */
      printf("Received %ld bytes in %f seconds\n", received_bytes, sec);
      printf("Measured receiver throughput for iteration %d is %f Mbps\n",
              x, throughput);
    }
    else { /* CLIENT */
      throughput = ((sent_bytes * 8) / sec) / 1e6; /* in Mbps */
      printf("Sent %ld bytes in %f seconds\n", sent_bytes, sec);
      printf("Measured sender throughput for iteration %d is %f Mbps\n",
              x, throughput);
    }
    fflush (stdout);
    throughput_sum += throughput;
  }

  initiate_barrier();
  while (barrier_completed() != 0) ;



  avg_throughput = throughput_sum / ITERATIONS;
  printf("Average %s throughput is %f\n", get_my_node_number() ? "sender":"receiver",
avg_throughput);

  while(try_close_net() != 0)
```

```
    poll();

  if (get_my_node_number() == SERVER)
    fp = fopen(RLOG_FILE,"a");
  else
    fp = fopen(SLOG_FILE,"a");

  gettimeofday (&start, NULL);

  /* get the current date */
  time(&curtime);
  tp = localtime(&curtime);
  strftime(date, (size_t) 50, "%a %b %d %R", tp);

  fprintf(fp,"%s\tIsotach\t%d",date,get_max_payload());
  if (get_SIU_state() == 1) {
    fprintf(fp,"\tSIU");
  }
  else {
    fprintf(fp,"\tnoSIU ");
  }
  if (get_MBM_ver() == 1) {
    fprintf(fp,"\tnoIOM");
  }
  else {
    fprintf(fp,"\tIOM");
  }
  fprintf(fp,"\t%.02f Mbps\n", avg_throughput );
  fclose(fp);


  return 0;
}
```

noniso.h

```
/*
 * -------------------------------------------------------------------------
 *
 * Isotach Module  : Non-Iso API Header File
 * Isotach Layer   : Main
 * Isotach Version : Version 3
 *
 * -------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/noniso.h,v $
 *       $Revision: 1.4 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:15 $
 *
 * -------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This header file is included by the Isotach Application Programmer when
 * Non-Isotach functionality is requested.  It exports all of the Non-Isotach
 * API functions to the application writer.
 *
 * -------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * If the Application programmer wishes to use ISO API Functions as well,
 * he/she must also include either the iso_smm.h or the iso_mbm.h file.
 *
 * -------------------------------------------------------------------------
 */

#ifndef NONISO_H
#define NONISO_H

/* Hostman API Functions */
#ifndef HOSTMAN_EXP
#define HOSTMAN_EXP

#define SUCCESS 0
#define FAILURE 1
#define TRUE    1
#define FALSE   0

/* Initializes Isotach system and synchronizes hosts.  Must be called at the
   beginning of any program using Isotach or Non-Isotach functionality, before
   any other API functions are called */
extern int open_net(int mode);

/* Attempts to stop all communication by using an approrpiate barrier, and
   shutting down when all nodes have completed the barrier */
extern int try_close_net();

/* Can be called explicitly by the Application, or is called by any of the
   other API functions listed below.  Passes control to the mLayer so that
   message processing can be done.  MUST be called frequently, either directly
   or indirectly
   NOTE: The application programmer should ONLY call poll().  The function
   poll() is mapped to one of the below poll functions depending on the
   api header files that are included. */
extern int noniso_poll();
extern int iso_poll();

/* Returns the Node Number of the selected host */
extern int get_node_number(char *hostname);

/* Returns the Node Number of the currently running host */
extern int get_my_node_number();
```

```c
/* Returns the total number of active hosts in the system */
extern int get_number_of_hosts();

/* Returns the value of MAX_PAYLOAD_SIZE, which is set in constants.h */
extern int get_max_payload();

/* Returns 1 is SIU is enabled, 0 if SIU is not enabled */
extern int get_SIU_state();

/* Returns the MBM Version Number.  1 if Ordering is not supported, and 2 if
   Ordering is supported (functionality from IOM Module) */
extern int get_MBM_ver();

#endif

/*
 * This code allows the correct poll function to be mapped to an application's
 * call to poll().  If both (iso_mbm.h or iso_smm.h) and noniso.h files are
 * included by the application, then poll() is mapped to iso_poll().
 * Otherwise, poll() is mapped to noniso_poll().  The arguments passed to
 * open_net() indicate whether the application has requested an ISO, NONISO or
 * BOTH environment. This is so the mLayer can initialize its data structures
 * appropriately.
 */

#ifdef ISOTACH_H
#undef poll()
#define poll() iso_poll()
#undef open_net()
#define open_net() open_net(2)
#else
#define poll() noniso_poll()
#define open_net() open_net(0)
#endif

/* Non-Iso Data Structure used in calls to receive() */
typedef struct {
  int sender_id;     // The originator of this message
  int size;          // The length of the data being pointed to by data_ptr
                     // if this is a -1, then the data is stored in data
  union msg_tag {
    void *data_ptr; // Pointer to pinned memory where data is kept
    int   data;     // An actual word of data
  } msg;
} noniso_mbm;

/* Non-Iso Protocol Specific API Functions */

/* Called to send a Non-Isotach Message to the host specified by the argument
   target.  A pointer to the data, which is the body of the message, is sent
   with the *data parameter.  The size of the data must also be sent.
   NOTE: Target must be a valid host number, and cannot be the host number of
   the sending node. (i.e. self-messages for NONISO are not supported) */
extern int send(int target, void *data, int size);

/* Called to recieve a Non-Isotach Message.  If a message is placed in the
   noniso_mbm struct, the function returns a value of 0 (SUCCESS).  If there
   was no message to be delivered, the function returne 1 (FAILURE) and there
   is no data placed in the noniso_mbm structure

   NOTE: The Application Programmer must copy the data pointed to by the
   noniso_mbm struct before the next call to recieve() or the data will be
   lost. */
extern int receive(noniso_mbm *recv_msg);

/* Called to begin a new barrier.  Only one barrier may be initiated at a time.
   To end the barrier, barrier_completed() must be called */
extern int initiate_barrier();

/* Called to check the status of an initiated barrier.  When the barrier is
   completed, the function returns 0 (SUCCESS), otherwise it returns 1
   (FAILURE). */
extern int barrier_completed();
```

```
#endif
```

isotach.h

```
/*
 * ---------------------------------------------------------------------------
 *
 * Isotach Module  : Isotach Main Header File
 * Isotach Layer   : Main
 * Isotach Version : Version 3
 *
 * ---------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/isotach.h,v $
 *       $Revision: 1.6 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:15 $
 *
 * ---------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This header file is included by either the iso_mbm.h or iso_smm.h header
 * files, which are included by the application programmer.  This file
 * contains the API functions that are common to both the SMM and MBM
 * interfaces.
 *
 * ---------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * If the Application programmer wishes to use NONISO Send and Receive,
 * he/she must also include the noniso_api.h file.
 *
 * ---------------------------------------------------------------------------
 */


#ifndef ISOTACH_H
#define ISOTACH_H

/* Hostman API Functions */
#ifndef HOSTMAN_EXP
#define HOSTMAN_EXP

#define SUCCESS 0
#define FAILURE 1
#define TRUE    1
#define FALSE   0

/* Initializes Isotach system and synchronizes hosts.  Must be called at the
   beginning of any program using Isotach or Non-Isotach functionality, before
   any other API functions are called */
extern int open_net(int mode);

/* Attempts to stop all communication by using an apprpriate barrier, and
   shutting down when all nodes have completed the barrier */
extern int try_close_net();

/* Can be called explicitly by the Application, or is called by any of the
   other API functions listed below.  Passes control to the mLayer so that
   message processing can be done.  MUST be called frequently, either directly
   or indirectly.
   NOTE: The application programmer should ONLY call poll().  The function
   poll() is mapped to one of the below poll functions depending on the
   api header files that are included. */
extern int noniso_poll();
extern int iso_poll();

/* Returns the Node Number of the selected host */
extern int get_node_number(char *hostname);

/* Returns the Node Number of the currently running host */
```

```c
extern int get_my_node_number();

/* Returns the total number of active hosts in the system */
extern int get_number_of_hosts();

/* Returns the value of MAX_PAYLOAD_SIZE, which is set in constants.h */
extern int get_max_payload();

/* Returns 1 is SIU is enabled, 0 if SIU is not enabled */
extern int get_SIU_state();

/* Returns the MBM Version Number.  1 if Ordering is not supported, and 2 if
   Ordering is supported (functionality from IOM Module) */
extern int get_MBM_ver();

#endif

/*
 * This code allows the correct poll function to be mapped to an application's
 * call to poll().  If both (iso_mbm.h or iso_smm.h) and noniso.h files
 * are included by the application, then poll() is first undefined and then
 * mapped to iso_poll().  Otherwise, poll() is directly mapped to iso_poll().
 * The arguments passed to open_net() indicate whether the application has
 * requested an ISO, NONISO or BOTH environment.
 * This is so the mLayer can initialize its data structures appropriately.
 */

#ifdef NONISO_H
#undef poll()
#define poll() iso_poll()
#undef open_net()
#define open_net() open_net(2)
#else
#define poll() iso_poll()
#define open_net() open_net(1)
#endif

/* Group Communication API Functions */

/* Registers a channel to a Isotach signal.  Before a signal can be used, it
   must be registered.  Valid channels are: 1-5.  The function returns 0
   (SUCCESS) if the signal was properly registered, and 1 (FAILURE) if not. */
extern int iso_register_signal(int channel);

/* Clears a signal channel.  Can be called with values of 1-4, and returns
   0 (SUCCESS) if the signal channel was cleared, otherwise 1 (FAILURE). */
extern int iso_clear_signal(int channel);

/* Registers a channel to a barrier with a specified barrier mode.  Valid
   channels are 0-1, and valid barrier modes are ISO_BARRIER_[WEAK|STRONG]
   Returns 0 (SUCCESS) if the barrier could be registered, 1 (FAILURE)
   otherwise. */
#define ISO_BARRIER_WEAK   0
#define ISO_BARRIER_STRONG 1
extern int iso_register_barrier(int channel, unsigned char barrier_mode);

/* Clears the specified barrier channel.  Returns 0 (SUCCESS) if the barrier
   channel could be successfully cleared, and 1 (FAILURE) otherwise. */
extern int iso_clear_barrier(int channel);

/* Sends a signal on the specified channel.  The channel must be registered
   by using iso_register_signal, and it must be owned by the Application.
   Valid channels are 1-4.  If the signal could be sent, the function returns
   0 (SUCCESS) otherwise, 1 (FAILURE). */
extern int iso_send_signal(int channel);

/* Initiates a barrier on the specified channel with the specified Barrier
   Mode.  The channel must have been registered previously with a call to
   iso_register_barrier, and be in the range 0-1.  The barrier_mode must match
   the barrier mode that the channel was registered under, and be either
   ISO_BARRIER_WEAK or ISO_BARRIER_STRONG.  If the barrier could be initiated,
   the function returns 0 (SUCCESS) otherwise it returns 1 (FAILURE). */
extern int iso_barrier(int channel, unsigned char barrier_mode);
```

```c
/* Signify End of Isochron.  This is called when you wish to complete
   an Isochron. */
extern int iso_end();

/* Isotach Message Data Structure used for iso_mbm */
typedef struct {
  unsigned short sender;       // The ID of the sending host
  unsigned short length;       // The length in bytes of the data
  void *data;                  // A pointer to the actual data
} iso_msg;

/* Isotach Data Structure used by the Application in calls to iso_receive() */
typedef struct {
  unsigned char  tag;          // The type of the structure (pointer, bs)

  union {
    unsigned char bits;        // Barrier and Signal Bits
    iso_msg msg;               // Pointer to the Data in an iso_msg
  } info;
} iso_mbm;

#endif
```

iso_mbm.h

```c
/*
 * ---------------------------------------------------------------------------
 *
 * Isotach Module  : Isotach Message Based Model (MBM) Header File
 * Isotach Layer   : Main
 * Isotach Version : Version 3
 *
 * ---------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/iso_mbm.h,v $
 *       $Revision: 1.6 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:15 $
 *
 * ---------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This header file is included by the application programmer when he/she
 * wishes to use the MBM interface.  This cannot be included with the
 * iso_smm.h file, as the SMM and MBM interface cannot be used concurrently.
 *
 * ---------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * If the Application programmer wishes to use NONISO Send and Receive,
 * he/she must also include the noniso_api.h file.
 *
 * ---------------------------------------------------------------------------
 */

#ifdef ISO_SMM_H
#error Cannot Include both the MBM and SMM Interfaces.  Aborting.
#endif

#ifndef ISO_MBM_H
#define ISO_MBM_H

/* This includes API Functions that are common to both the SMM and MBM
   interfaces */
#include <isotach.h>

/* Isotach Message Based Model API Functions */

/* This sends an Isotach Message to the specified host.  The target host must
   be a valid host in the Isotach Network, and can be the current host (i.e.
   self-messages ARE supported).  A pointer to the data must be sent, along
   with the data size.  Finally, last_in_isochron must be sent.  A value
   of TRUE indicates that this is the last message in the pending isochron,
   while a value of FALSE indicates that more messages need to be sent in this
   isochron.  Setting TRUE here is the equivalent of setting FALSE and then
   calling iso_end(). */
extern int iso_send(int target, void *data, int size, int last_in_isochron);

/* This function attempts to receive an Isotach Message.  When called, if
   there is a message to be received, it is placed in the iso_mbm structure
   and a value of 0 (SUCCESS) is returned.  Otherwise, 1 (FAILURE) is returned
   to indicate that there was no message to be received.  The message can
   either contain an iso_msg struct which has a pointer to some data, or it
   can contain a byte of bits which can be interpreted to determine if signals
   have arrived.

   NOTE: The Application Programmer MUST copy the data pointed to by the
   iso_msg structure before the next call to iso_receive() or the data will
   be lost. */
extern int iso_receive(iso_mbm *data);

#endif
```

iso_smm.h

```
/*
 * ---------------------------------------------------------------------------
 *
 * Isotach Module  : Isotach Shared Memory Model (SMM) Header File
 * Isotach Layer   : Main
 * Isotach Version : Version 3
 *
 * ---------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/iso_smm.h,v $
 *       $Revision: 1.6 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:15 $
 *
 * ---------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This header file is included by the application programmer when he/she
 * wishes to use the SMM interface.  This cannot be included with the
 * iso_mbm.h file, as the SMM and MBM interface cannot be used concurrently.
 *
 * ---------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * If the Application programmer wishes to use NONISO Send and Receive,
 * he/she must also include the noniso_api.h file.
 *
 * NOTE: The Isotach Shared Memory Model has NOT been implemented yet.
 *       Please DO NOT include this header file, as the SMM code is still
 *       under development.  The Isotach Message Based Model, and the Non-
 *       Isotach systems have been implemented, so please include either:
 *       noniso.h and/or iso_mbm.h
 *
 * ---------------------------------------------------------------------------
 */

#ifdef ISO_MBM_H
#error Cannot Include both the MBM and SMM Interfaces.  Aborting.
#endif

#ifndef ISO_SMM_H
#define ISO_SMM_H

/* This includes API Functions that are common to both the SMM and MBM
   interfaces */
#include <isotach.h>

/* Shared Memory Messages API Functions */

/* This may or may not be accurate.  The design has not been finalized at this
   point, and these functionalities have not been implemented yet. */
extern int iso_sched_read(shmem_addr_t shaddr, iso_var32 *laddr,
                              int last_in_isochron);
extern int iso_get_read(iso_var32 *var, int last_in_isochron);
extern int iso_write(shmem_addr_t shaddr, long int val, int last_in_isochron);
extern int iso_sched(shmem_addr_t shaddr, int last_in_isochron);
extern int assign(shmem_addr_t shaddr, long int val, int last_in_isochron);

#endif
```

hostman/constants.h

```
/*
 * ---------------------------------------------------------------------------
 *
 * Isotach Module  : Global Constants Header File
 * Isotach Layer   : Hostman
 * Isotach Version : Version 3
 *
 * ---------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *        $Source: /home2/isotach/cvsroot/v3/hostman/host_utils.h,v $
 *        $Revision: 1.9 $
 *        $Author: pnm2h $
 *        $Date: 1999/11/04 22:29:16 $
 *
 * ---------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This header file contains the globally used constants, type
 * definitions and macros used throught the messaging layer.
 *
 * This file contains definitions and variables that may be changed to tweak
 * Isotach performance.  The other header files (utils.h and host_utils.h)
 * contain definitions and variables that should not normally be changed.
 *
 * ---------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * None.
 *
 * ---------------------------------------------------------------------------
 */

#ifndef CONSTANTS_H
#define CONSTANTS_H

/* ************************************************************************* */
/* MODIFY THESE VALUES TO TUNE PERFORMANCE OF MESSAGING LAYER                */
/* ************************************************************************* */

/* Set this to 1 if we want to use a hardware SIU, 0 otherwise */
#define SIU 0

/* Set this to one to remove support for isotach ordering and self messages,
   or 2 to keep that support in */
#define MBM_VER 1

/* MAX_PAYLOAD_SIZE can be changed to allow more data to be transmitted with
   each packet.  This size is in bytes */
#define MAX_PAYLOAD_SIZE 1024

/* This is the size of pinned memory in bytes.  PLEASE change this value
   if you decide to change the size of pinned memory in lilo.conf */
#define SIZE_OF_PINNED_MEMORY  4194304

/* Changing the value of ISO_NONISO_RATIO allows the application designer to
   distribute resources among Isotach and NonIsotach protocols */
#define NONISO_RATIO           0.50

/* These percentages allow us to place a little 'slack' in the host and NIU
   buffers, to account for control messages that are not subject to software
   flow control. */
#define ISO_SLACK              1.25

/* The Size of the Host's send buffer in # of PACKETS */
#define SEND_BUF_SIZE          1024

/* The Size of the Host's ord_send buffer in # of PACKETS */
```

```c
#define ORD_SEND_BUF_SIZE      1024

/* The Size of the Host's iso_send buffer in # of PACKETS */
#define ISO_SEND_BUF_SIZE      1024

/* These values correspond to buffers on the LANai */

#define NIU_SEND_SIZE          64
#define ISO_NIU_SEND_SIZE      64
#define RECV_LIMIT             256
#define ISO_NIU_RECV_SIZE      512
#define NIU_DELV_SIZE          8192
#define NIU_RECV_BUF_SIZE      32


/* Isotach specific constants */
#define isochron_allowance     256
#define BUCKET_COUNT           256
#define BUCKET_SIZE            256


/* ************************************************************************ */
#endif
```

hostman/utils.h

```
/*
 * ------------------------------------------------------------------------
 *
 * Isotach Module  : Utilities Library
 * Isotach Layer   : Hostman
 * Isotach Version : Version 3
 *
 * ------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/hostman/utils.h,v $
 *       $Revision: 1.14 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:16 $
 *
 * ------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This header file contains the globally used structures, constants, type
 * definitions and macros used throught the messaging layer.
 *
 * ------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * None.
 *
 * ------------------------------------------------------------------------
 */

#ifndef UTILS_H
#define UTILS_H

#include <hostman/constants.h>


/* ************************************************************************
 * Generic Definitions and Constants
 * ************************************************************************/

typedef unsigned char  UCHAR;
typedef unsigned short USHORT;
typedef unsigned long  ULONG;

/* Some useful boolean values */
#define TRUE    1
#define FALSE   0
#define SUCCESS 0
#define FAILURE 1

/* Masks to extract the nth byte of a word */
#define FIRST_BYTE  0xFF000000
#define SECOND_BYTE 0x00FF0000
#define THIRD_BYTE  0x0000FF00
#define FOURTH_BYTE 0x000000FF

/* The different stages of initialization are enumerated here */
enum init_stage {
  START = 0x0, START_LANAI, INIT_DMA_TEST, CHECK_DMA_TEST, SYNCHRONIZE, FINISHED, BAD_CRC, STOP
};


/* ************************************************************************
 * Isotach Specific Definitions and Constants
 * ************************************************************************/

#define NONISOTACH 0
#define ISOTACH    1
#define BOTH       2
```

```c
/* PACKET_HEADER_SIZE should remain constant at 20 bytes */
#define PACKET_HEADER_SIZE 20

/* First Byte of Prefix in an Isotach Packet */
#define BS_MARKER_MASK   0x40000000
#define EOI_MASK         0x20000000
#define CHANNEL_MASK     0x10000000
#define SEQ_CON_SET_MASK 0x08000000
#define SEQ_CON_MASK     0x04000000
#define LOG_TS_MASK      0x02000000
#define HOST_TS_MASK     0x01000000

/* Packet Types */

static const USHORT NONISO            = 0x0620;
static const USHORT ISO               = 0x0600;
static const USHORT EOP               = 0x0601;

/* Packet Subtypes */

// Synchronization Subtypes - Used in NONISO typed packets
static const UCHAR SYNC               = 0x00;
static const UCHAR SYNC_ACK           = 0x01;
static const UCHAR SYNC_DONE          = 0x02;

// Flow Control Subtypes - Used in NONISO typed packets
static const UCHAR CREDIT             = 0x03;
static const UCHAR ISO_CREDIT         = 0x04;
static const UCHAR REQUEST_CREDIT     = 0x05;
static const UCHAR REQUEST_ISO_CREDIT = 0x06;

// Noniso GC and MBM Subtypes - Used in NONISO typed packets
static const UCHAR BARRIER            = 0x07;
static const UCHAR NONISO_MBM         = 0x08;
static const UCHAR ORDERED            = 0x09;

// Isotach Subtypes - Used in ISO typed packets
static const UCHAR ISO_MBM            = 0x10;
static const UCHAR ISO_READ           = 0x11;
static const UCHAR ISO_WRITE          = 0x12;
static const UCHAR ISO_ASSIGN         = 0x13;
static const UCHAR ISO_SCHED          = 0x14;
static const UCHAR READ_RESPONSE      = 0x15;
static const UCHAR BS_MARKER          = 0x16;
static const UCHAR ISO_MARKER         = 0x17;
static const UCHAR EOP_MARKER         = 0x18;
static const UCHAR ISO_SLOT           = 0x19;

// Stop Packet - This is so we don't have to worry about switching on the LANAI
static const ULONG STOP_PACKET        = 0x99999999;


/* ************************************************************************
 * Packet Structures Section
 * ************************************************************************/

/* The PACKET structure - Essentially, every NonISO and Ordered message
   are sent out in PACKETs.  Also, all control messages are sent out in
   PACKETs (i.e. credit packets, etc...) */
typedef struct {
  USHORT type;              // field that designates packet type
  UCHAR  subtype;           // subtype field
  UCHAR  pad1;              // padding
  ULONG  sender;            // field that holds sender ID
  ULONG  address;           // field that holds an address or a pad
  USHORT payload_length;    // length of data in bytes
  USHORT pad2;              // pad
  ULONG  credit_info;       // credit information piggybacked on packet
  UCHAR  data[MAX_PAYLOAD_SIZE + 4]; // payload of packet, 4 bytes added
                                     // to account for crc field when packets
                                     // are received.
  ULONG  route;             // 4-byte route with padding if necessary
} PACKET;
```

```c
/* used when you want queues or arrays of pointers to packets */
typedef PACKET* PACKET_PTR;


/* All of the fields in the following packet structures are explained in the
   Ironman document in one of the Appendixes.  Please reference that document
   for more details */

/* Isotach is interesting in that we send out 3 different types
   of packets: Iso-pointers, SRefs, and BS-Markers, but we can
   receive 4 different types of packets: Iso-pointers, SRefs,
   Isochron Markers, and EOP Markers.  Here are the data structures
   for receiving:  There is an iso_recvframe that incorporates pointers
   and srefs, and roughly corresponds to what is received off of the
   network.  In the case of a pointer, the crc should be copied into
   the crc field of the receive frame.  Additionally, we have isochron
   markers, where the isochron crc needs to be or'ed with the packet crc
   and stored in the crc field.  After which, the isochron_marker subtype
   should be written into the subtype field.  Then there are eop markers
   which can be of variable length.  When an eop marker is received, the
   crc is copied into the crc field of the structure, and the sequence
   number is overwritten with subtype eop_marker.  Note that the myrinet
   packet type will not be copied into the iso_niu_recive_buffer or the
   iso_receive_buffer */


/* this is the body of an sref */
typedef struct {
  ULONG shadder;            // shared mem address
  ULONG data;               // shared mem data
} iso_sref;

typedef struct {
  USHORT type;
  UCHAR subtype;
  UCHAR TS;                 // isotach timestamp
  USHORT sender;            // sender node number
  USHORT credit_info;       // credit information
  union {
    ULONG pointer;          // Either a pointer to the data, or
    iso_sref sref;          // an SREF
  } body;
} ISO_PACKET;

/* this data structure is what is received off of the network
   in some/most ? cases, the crc will be found inside the packet
   body.  It should be copied into the crc field of this structure
   so iso_receiving can check the crc */

/* note that this structure is actually word aligned... */
typedef struct {
  ISO_PACKET packet;       //a union containing all possible types
  UCHAR crc;               //the crc field
} ISO_RECVFRAME;

/* now the size in words */
#define ISO_RECVFRAME_SIZE 5

/* this is an isochron marker */
typedef struct {
  USHORT type;
  UCHAR subtype;
  UCHAR TS;
  USHORT source;
  UCHAR iso_id;
  UCHAR crc;
} isochron_marker;

/* now the size in words */
#define ISO_MARKER_SIZE 2

/* this is an eop marker */
typedef struct {
```

```
  USHORT type;
  UCHAR subtype;
  UCHAR bits;
  USHORT pad;
  UCHAR TS;
  UCHAR count;
  UCHAR sort_vector[32];
  UCHAR crc;
} eop_marker;

/* now the size in words */
#define EOP_MARKER_SIZE 11

/* this data structure is what will be dma'd onto the network.
   It contains the prefix, the route, the myrinet packet type,
   the isotach packet, and a size field.  The size field is
   necessary since different packet types have different sizes.
   For example, a BS_Marker will only be 4 bytes long and thus
   is contained only in the prefix.  An iso_pointer is shorter
   than an sref as another example.
*/

typedef struct {
  ULONG prefix;           // isotach prefix -- can also contain
                          // entire bs_marker
  ULONG route2;
  USHORT route1;
  USHORT pad;
  ISO_PACKET packet;      // isotach packet
} ISO_SENDFRAME;


/* ************************************************************************
 * Isotach CORE Structures
 * ************************************************************************/

typedef struct {
  UCHAR subtype;
  UCHAR pad;
  USHORT sender;
  PACKET_PTR pointer;
} isopointer;

typedef struct {
  UCHAR subtype;
  UCHAR bits;
  UCHAR pad[6];
} bsnotice;

typedef struct {
  UCHAR subtype;
  UCHAR iso_id;
  USHORT pad;
  ULONG self_count;
} isoslot;

typedef struct {
  UCHAR subtype;
  UCHAR pad;
  USHORT source;
  ULONG shadder;
  ULONG data;
} isosched;

typedef struct {
  UCHAR subtype;
  UCHAR pad;
  USHORT source;
  ULONG shadder;
  ULONG data;
} isowrite;

typedef struct {
  UCHAR subtype;
```

```
  UCHAR pad;
  USHORT source;
  ULONG shadder;
  ULONG data;
} isoassign;

typedef struct {
  UCHAR subtype;
  UCHAR tag;
  USHORT source;
  ULONG shadder;
  ULONG lvar;
} isoread;

typedef struct {
  UCHAR subtype;
  UCHAR pad;
  USHORT sender;
  UCHAR iso_id;
  UCHAR pad2[7];
} isomarker;

typedef struct {
  UCHAR subtype;
  UCHAR pad;
  USHORT pad2;
  ULONG data;
  ULONG lvar;
} readresponse;

typedef struct {
  UCHAR subtype;
  UCHAR data[7];
} PACKET_CORE_2;

typedef struct {
  UCHAR subtype;
  UCHAR pad;
  USHORT sender;
  UCHAR data[8];
} PACKET_CORE_3;


/* ***************************************************************************
 * Queue Macro Functions
 * ***************************************************************************/

/* inc_idx(P,S)
 *
 * This macro defines a function which takes an integer pointer value, and the
 * size of a queue and increments the pointer.  If the pointer is at the last
 * slot in the queue, it is wrapped around to the beginning of the queue.
 * This macro is used for incrementing head and tail pointers of index based
 * queues.
 */
#define inc_idx(I,S) (I=((I+1)%S))

/* inc_ptr(P,S,Q)
 *
 * This macro defines a function which takes a pointer, the size of a queue
 * in bytes and the starting address of the queue.  It increments the pointer
 * by x bytes, where x is the size of the element type stored in the queue.
 * If the address of the pointer is equal to the starting address of the queue
 * plus the size of the queue, the address of the pointer is wrapped
 * around the queue by setting it to the base address of the queue.
 * This macro is used for incrementing head and tail pointers of address based
 * queues.
 */
#define inc_ptr(P,S,Q) \
{ P++; \
P = (P==(Q+S)) ? Q : P; }

/* queue_empty(H,T)
```

```
 *
 * This macro returns TRUE if the head of the queue is equal to the tail.
 * Otherwise, it returns FALSE.  This (and all of the following queue macros)
 * works for both indexed and address based queues.
 */
#define queue_empty(H,T) (H==T)

/* queue_index(P,Q)
 *
 * Gives the index number of a pointer (either head or tail) into a queue.
 * For address based queues, the pointer is passed along with the base address
 * of the queue.  For indexed queues the pointer (an integer index) can be
 * passed with zero (the integer index of the beginning of the integer queue).
 * However, this really should only be used for address based queues.
 */
#define queue_index(P,Q) (P-Q)

/* queue_size(H,T,S)
 *
 * This returns the current size of the queue in terms of the number of
 * elements.  To determine the size of the queue in bytes, simply multiply
 * the result of this macro by the size of each element in the queue.
 */
#define queue_size(H,T,S) ((H<=T) ? (T-H) : (S-(H-T)))

/* queue_full(H,T,S)
 *
 * This returns TRUE is the queue is full.  A queue is full if:
 * The current queue size is equal to the queue-size - 1.
 * If the queue is not full, this returns false.
 */
#define queue_full(H,T,S) ((queue_size(H,T,S)==(S-1)) ? TRUE : FALSE)

/* lanai_queue_full(H,T,S)
 *
 * This returns TRUE if the queue is full, just as the above macro does, but
 * it does not use the queue_size() macro.  This is used on the lanai instead
 * of queue_full because it is less computationally expensive.
 */
#define lanai_queue_full(H,T,S) (((H + 1)%S)==T)

/* q_last_item(T,S,Q)
 *
 * This macro returns the last item of the specified Queue.  It is used in
 * the IOM module.  It is a peek function, as it does not actually remove
 * the last item from the queue.
 */
#define q_last_item(T,S,Q) ( ((T-1)>=Q) ? (T-1) : (Q+S-1))

#endif
```

hostman/host_utils.h

```
/*
 * -------------------------------------------------------------------------
 *
 * Isotach Module   : Host Utilities Library
 * Isotach Layer    : Hostman
 * Isotach Version  : Version 3
 *
 * -------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/hostman/host_utils.h,v $
 *       $Revision: 1.9 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:16 $
 *
 * -------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This header file contains the globally used structures, constants, type
 * definitions and macros used throught the messaging layer.
 *
 * -------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * None.
 *
 * -------------------------------------------------------------------------
 */

#ifndef HOST_UTILS_H
#define HOST_UTILS_H

#include <stdio.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#include <lanai_device.h>
#include <hostman/utils.h>
#include <sys/time.h>

/* The system type.  It can be either ISOTACH, NONISOTACH or BOTH */
extern UCHAR SYS_TYPE;


/* ************************************************************************
 * Pinned Memory Variables, Definitions and Structures
 * ************************************************************************/

/* Each Non-Isotach Recieve Buffer has 6 Packet slots allocated for packets
   that are used by the mLayer.  If more slots need to be added, change the
   below value to be one more than the number of slots.  (i.e. we have 5
   control packet slots, and the value of CONTROL_SLOTS is 6).  This creates
   a buffer between the last control slot, and the first packet of real data.
*/
#define CONTROL_SLOTS 6
// Control Slot 0 -> Credit Packet
// Control Slot 1 -> Credit Request Packet
// Control Slot 2 -> Barrier Packet
// Control Slot 3 -> Iso Credit Packet
// Control Slot 4 -> Iso Credit Request

/* The size in bytes of the Isotach Pinned Memory region */
ULONG ISO_PINNED_SIZE;
/* The size in bytes of the Non-Isotach Pinned Memory region */
ULONG NONISO_PINNED_SIZE;

/* The size of the Isotach Recieve Buffer */
ULONG ISO_RECV_SIZE;
```

```
/* The total number of Isotach Credits at this node */
ULONG ISO_CREDITS;
/* The total number of Non-Isotach Credits at this node */
ULONG NONISO_CREDITS;

/* Information about queues in pinned memory.  Each one of these structures
   contains an address that points to the beginning of a queue in pinned memory
   and a size in # of PACKETS */
typedef struct {
  PACKET_PTR base;
  ULONG      size;
} pinned_mem_info;

ULONG offset;


/* ************************************************************************
 * Barrier, Signal and Credit Enumerated Types and Structures
 * ************************************************************************/

/* This enumeration defines the different ownerships that a signal or barrier
   may have. Used by both iso_barrier, iso_signal and hostman */
enum bs_type {
  APPLICATION = 0x0, MLAYER, UNCLAIMED
};

/* Array of masks to extract the nth bit of a byte
   The array is actually initialized in one of the modules. */
UCHAR BITS[8];

#define NULL_CREDIT 0


/* ************************************************************************
 * LANAI Specific Variables and Data Structures
 * ************************************************************************/

lanai_symbol_table *symbol_table;
volatile char *lanai0;


/* ************************************************************************
 * General Isotach System Information
 * ************************************************************************/

/* Synchronization variable used during system initialization.  It allows
   the host and LANAI to inform each other of events during initialization */
volatile UCHAR *init_stage;

/* The total number of hosts in the Isotach network, and this node's host ID */
extern ULONG number_of_hosts;
extern ULONG host_node_id;


/* ************************************************************************
 * Network Status Table Structures
 * ************************************************************************/

/* The structure for elements in the Network Status Table (nst).  Each entry
   in the nst contains an id, a name, a route from the current host to this
   host, the status of the node (alive, dead) and pinned memory information.
   remote_noniso (iso)  points to the base of the current node's noniso (iso)
   queue on that remote host.  local_noniso (iso) points to the remote host's
   noniso (iso) queue in the current host's pinned memory */
typedef struct {
  ULONG node_id;
  char host_name[32];
  ULONG route;
  UCHAR alive;
  pinned_mem_info remote_noniso;
  pinned_mem_info remote_iso;
  pinned_mem_info local_noniso;
  pinned_mem_info local_iso;
} node_info;
```

```
/* ************************************************************************
 * General System Functions
 * ************************************************************************/

/* This function maps a host variable to a lanai memory-mappable variable.
   It is used by hostman open_net and the other host module initialization
   functions */
static inline ULONG *get_lanai_sym (char *name) {
  ULONG *x = NULL;
  ULONG y = lanai_symbol_value (symbol_table, name);

  if (y == 0) {
    printf ("ERROR: lanai symbol '%s' not found\n", name);
    exit (1);
  }

  x = (ULONG *)&lanai0[y];
  return x;
}

static void callback () {
  // Dummy Function for passing to lanai_load_and_reset()
}


/* ************************************************************************
 * Debugging Functions
 * ************************************************************************/

/* This small function is used for debugging, and prints out the contents of
   the packet passed in as a parameter */
static void print_out_packet(PACKET *p) {
  int i;

  printf("Type->%04X\n", ntohs(p->type));
  printf("Subtype->%02X\n", p->subtype);
  printf("Pad1->%02X\n", p->pad1);
  printf("Sender->%lu\n", p->sender);
  printf("Address->%lu\n", (ULONG)ntohl((ULONG)p->address));
  printf("Payload_length->%04X\n", ntohs(p->payload_length));
  printf("Pad2->%04X\n", ntohs(p->pad2));
  printf("Credit_info->%lu\n", p->credit_info);
  printf("Data->");

  for (i = 0; i < ntohs(p->pad2) - 16; i++) {
    printf("%02X",p->data[i]);
  }

  printf("\n");
  fflush(stdout);
  return;
}

/* This small function is used for debugging, and prints out the contents of
   the isotach packet passed in as a parameter */
static void print_out_iso_packet(ISO_PACKET *p) {
  printf("Type->%04X\n", ntohs(p->type));
  printf("Subtype->%02X\n", p->subtype);
  printf("TS->%02X\n", p->TS);
  printf("Sender->%04X\n", p->sender);
  printf("Credit_info->%04X\n", p->credit_info);

  if (p->subtype == ISO_MBM) {
    printf("Pointer->%04X\n",p->body.pointer);
  }
  else {
    printf("SHADDR->%04X\n",p->body.sref.shadder);
    printf("Data->%04X\n",p->body.sref.data);
  }

  fflush(stdout);
  return;
```

```c
}


/* This small function is used for debugging, and prints out the contents of
   the isochron marker passed in as a parameter */
static void print_out_iso_marker(isochron_marker *p) {
  int i;

  printf("Received isochron marker: ");

  for (i=0; i< 8; i++) {
    printf("%02X ",(UCHAR)*((UCHAR *)p + i));
  }

  printf("\n");
  fflush(stdout);
  return;
}

/* This function is used for timing and performance debugging */
static inline void subtracttime(struct timeval *t, struct timeval *sub) {
  signed long sec,usec;

  sec = t->tv_sec - sub->tv_sec;
  usec = t->tv_usec-sub->tv_usec;
  if (usec < 0) {
    sec--;
    usec+=1000000;
  }
  if (sec < 0) {
    t->tv_sec = 0;
    t->tv_usec = 0;
  }
  else {
    t->tv_sec = (unsigned long)sec;
    t->tv_usec = (unsigned long)usec;
  }
}


#endif
```

hostman/locals.h

```
/*
 * ---------------------------------------------------------------------------
 *
 * Isotach Module  : Hostman Local Variable Header File
 * Isotach Layer   : Hostman
 * Isotach Version : Version 3
 *
 * ---------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/hostman/locals.h,v $
 *       $Revision: 1.11 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/02 01:22:49 $
 *
 * ---------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This file contains all of the local variable and data structure definitions.
 * It also includes all other header files necessary for the module's function.
 * It also prototypes all functions internal to that module.
 *
 * ---------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * ---------------------------------------------------------------------------
 */

#ifndef HOSTMAN_LOCALS_H
#define HOSTMAN_LOCALS_H

#include <signal.h>
#include <unistd.h>
#include <hostman/host_utils.h>

#include <api/send/exports.h>
#include <api/deliver/exports.h>
#include <api/iso_deliver/exports.h>
#include <api/iso_send/exports.h>
#include <api/barrier/exports.h>
#include <api/iso_signal/exports.h>
#include <api/iso_barrier/exports.h>
#include <api/iso_retrieve/exports.h>

#include <processing/flow/exports.h>
#include <processing/iso_flow/exports.h>
#include <processing/iom/exports.h>
#include <processing/shmem/exports.h>

#include <niu_interface/shipping/exports.h>
#include <niu_interface/receive/exports.h>
#include <niu_interface/iso_shipping/exports.h>
#include <niu_interface/iso_receive/exports.h>

UCHAR SYS_TYPE;


/* ***************************************************************************
 * Hostman Level System Information
 * ***************************************************************************/

/* The LANAI Control Program and Network Configuration File should both be
 * in the same directory as the application. */
//char FILENAME[128];
//#define LCPFILE     strcat(strcpy(FILENAME, getenv("ISO_HOME")),"/lcp")
//#define CONFIG_FILE strcat(strcpy(FILENAME, getenv("ISO_HOME")),"/network.cfg")
#define LCPFILE "./lcp"
```

```c
#define CONFIG_FILE "./network.cfg"


/* The Network Status Table.  This table contains a listing of all hosts,
   and their memory addresses in pinned memory (for local buffers and remote
   buffers.  This is local to hostman, and passed to each module that needs it
   during initialization */
node_info *nst;

/* Stores the total number of hosts in the system and the current node's ID. */
ULONG number_of_hosts;
ULONG host_node_id;

/* Base address of where the Isotach region begins in Pinned Memory */
ULONG *iso_base_ptr;

/* NEVER NEVER NEVER SET THIS VALUE BELOW 2...
   This is the amount of time that we wait for the SIU hardware to reset
   during initialization. */
#define SIU_SLEEP_TIMER 2

/* ***************************************************************************
 * LANAI Mapped Variables
 * ***************************************************************************/

/* netman_hostbase contains the virtual address used by the LANAI in
 * referencing the beginning of the pinned memory area on the host.
 */
volatile ULONG *netman_hostbase;

/* Set to either ISO, NONISO or BOTH depending on which header files the
 * application programmer has included in the application program.
 */
UCHAR *NIU_SYS_TYPE;

/* hostman_hostbase contains the real address used by the host in
 * referencing the beginning of the pinned memory area on the host.
 */
PACKET_PTR hostman_hostbase;

/* netman_maxlen contans the maxlen variable, so that the LANAI can access it*/
volatile ULONG *netman_maxlen;
/* This contains the base address in pinned memory of the iso_recv buffer */
volatile ULONG *niu_iso_recv_base;


/* ***************************************************************************
 * Local Function Prototypes
 * ***************************************************************************/

int allocate_pinned_memory(int max_length);
static void callback();
void print_network_configuration();
int initialize_configuration_table();
int synchronize();
void print_config();
void handle_sigint(int s);
int isotach_init();
int isotach_deinit();

#endif
```

hostman/hostman.c

```
/*
 * ---------------------------------------------------------------------------
 *
 * Isotach Module  : Hostman
 * Isotach Layer   : Hostman
 * Isotach Version : Version 3
 *
 * ---------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/hostman/hostman.c,v $
 *       $Revision: 1.16 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:16 $
 *
 * ---------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This module is the 'main' module in the messaging layer.  It is responsible
 * for calling the init, poll and deinit functions for all other modules.  It
 * is also responsbile for coordinating the actions of all other modules.
 *
 * ---------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 *
 * ---------------------------------------------------------------------------
 */

#include <hostman/locals.h>

/* ************************************************************************
 * Internal Hostman Functions: allocate_pinned_memory,
 *                             callback,
 *                             initialize_configuration_table,
 *                             synchronize
 *                             isotach_init
 *                             isotach_deinit
 *                             shutdown
 * ************************************************************************
 */


/* This function is called by open_net and it assigns the correct addresses
   to each entry in the nst (for local_noniso and local_iso).
   For more information see the Ironman Document */
int allocate_pinned_memory(int max_length) {
  int i;

  /* The number of noniso and iso credits assigned to each host in the
     system */
  int noniso_host_credits;
  int iso_host_credits;

  /* Pointer to the first packet in the ord buffer region and the noniso
     buffer region */
  PACKET_PTR ord_base_ptr;
  PACKET_PTR noniso_base_ptr;

  /* This section uses defined values in the utils header file to determine
     the size of the NONISO region in pinned memory. */
  noniso_base_ptr = (PACKET_PTR)DBLOCK[0];

  /* Calculate the Total Size of NonIso Region in Pinned Memory.  This is done
     by using the total SIZE_OF_PINNED_MEMORY and the NONISO_RATIO defined
     in constants.h */
  NONISO_PINNED_SIZE = ((((SIZE_OF_PINNED_MEMORY * NONISO_RATIO) / sizeof(PACKET)) *
sizeof(PACKET)) / number_of_hosts) * number_of_hosts;
```

```
  /* The number of total NONISO Credits is dependent upon the size of the
     NIU Delivery Queue.  This protects the queue from being overfilled */
  NONISO_CREDITS     = (NIU_DELV_SIZE / number_of_hosts) * number_of_hosts;

  /* Calculate the size in packets of each hosts noniso queue in
     pinned memory.  If the packet size is large, we need to restrict the
     number of host_credits to fit into the NONISO region in pinned memory,
     otherwise use the full amount of NONISO_CREDITS split among the number
     of hosts. */
  if (sizeof(PACKET) * NONISO_CREDITS >= NONISO_PINNED_SIZE) {
    noniso_host_credits = (NONISO_PINNED_SIZE / number_of_hosts) / sizeof(PACKET);
    NONISO_CREDITS = noniso_host_credits * number_of_hosts;
  }
  else
    noniso_host_credits = NONISO_CREDITS / number_of_hosts;

  /* End NONISO Pinned Memory Calculations */


  /* This section uses defined values in the utils header file to determine
     the size of the ORDERED and ISO regions in pinned memory. */


  /* ISO_PINNED_SIZE corresponds to the amount of memory we have to
     store ordered and isotach messages */
  ISO_PINNED_SIZE = SIZE_OF_PINNED_MEMORY - NONISO_PINNED_SIZE;

  /* Unnecessary sanity check.  If everything above is correct, then
     ISO_PINNED_SIZE should already be word aligned. */
  ISO_PINNED_SIZE -= ISO_PINNED_SIZE % 4;

  /* this calculates the address where the ord queues start */

  ord_base_ptr = (PACKET_PTR)((ULONG)noniso_base_ptr + NONISO_PINNED_SIZE);

  /* Calculate how many isotach messages we can store in the remainder of
     pinned memory.  We need to be able to store an equal amount of
     ordered messages and isotach receive frames, since potentially
     we can receive one ordered message for every isotach message
     we receive.  To calculate iso_credits, we use the following formulas:

     size of ordered region = iso_credits * packetsize
     size of isotach region = iso_credits * 2 * sizeo(recvframe) * slack
     iso_pinned_mem_size    = size of ordered + size of isotach
  */

  ISO_CREDITS = ISO_PINNED_SIZE / (sizeof(PACKET) + 2*sizeof(ISO_RECVFRAME)*ISO_SLACK);

  ISO_CREDITS     = (ISO_CREDITS / number_of_hosts) * number_of_hosts;
  ISO_RECV_SIZE   = ISO_CREDITS * sizeof(ISO_RECVFRAME);
  iso_base_ptr = (ULONG *)(ord_base_ptr + ISO_CREDITS);
  iso_host_credits = ISO_CREDITS / number_of_hosts;

  /* Clears the pinned mem info sections of each nst entry */
  for (i = 0; i < number_of_hosts; i++)
    memset(&(nst[i].remote_noniso), 0, (sizeof(pinned_mem_info) * 4));

  /* For each host, calculate the starting address for their local noniso
     queue area in pinned memory.  If the system has Isotach functionality,
     also calculate the starting address for the node's local iso queue area
     in pinned memory */
  for (i = 0; i < number_of_hosts; i++) {
    nst[i].local_noniso.base = noniso_base_ptr + i*noniso_host_credits;
    nst[i].local_noniso.size = noniso_host_credits;
    nst[i].local_iso.base = ord_base_ptr + i*iso_host_credits;
    nst[i].local_iso.size = iso_host_credits;
  }

  return SUCCESS;
}


/* This function reads in the configuration file specified by CONFIG_FILE
```

```
   (which is usually in "./network.cfg") */
int initialize_configuration_table() {
  FILE *fp;

  char host_node_name[32]; // The hostname
  char line[128];          // Temporarily stores a line read in from a file.
  char search_str[16];     // String used to find your route in the table.
  char temp_host_name[32];

  int i;
  int node;
  int route;


  /* Find out the name of the current host */
  fp = popen("/bin/hostname -s","r");

  if (fp == (FILE *)NULL) {
    printf("ERROR: Cannot determine current host name\n");
    return FAILURE;
  }

  fgets(host_node_name, 31, fp);
  fclose(fp);
  host_node_name[strlen(host_node_name) - 1] = '\0';

  /* Time to read in the configuration file */
  fp = fopen(CONFIG_FILE,"r");

  if (fp == (FILE *)NULL) {
    printf("ERROR: Cannot open configuration file\n");
    return FAILURE;
  }

  /* ignore the first comment line */
  fgets(line, 127, fp);

  /* find the number of hosts */
  fscanf(fp, "hosts=%lu\n", &number_of_hosts);

  /* allocate space in the table */
  nst = (node_info *)malloc(number_of_hosts * sizeof(node_info));

  /* ignore the second comment line */
  fgets(line, 127, fp);

  /* read in all of the host names and IDs */
  for (i = 0; i < number_of_hosts; i++) {
    fscanf(fp, "%s %lu\n", temp_host_name, &nst[i].node_id);
    strcpy(nst[i].host_name, temp_host_name);

    /* set alive flag to FALSE */
    nst[i].alive = FALSE;

    if (!strcmp(host_node_name, nst[i].host_name)) {
      host_node_id = nst[i].node_id;
    }
  }

  /* find your routing table in the configuration file */
  sprintf(search_str, "node %lu:", host_node_id);

  while (!feof(fp)) {
    fgets(line, 127, fp);

    if (!strncmp(line, search_str, strlen(search_str))) {
      break;
    }
  }

  /* read in your route and put in the correct format */
  for (i = 0; i < number_of_hosts; i++) {
    fscanf(fp, "%d %x\n", &node,&route);
    nst[i].route = (ULONG)htonl(route);
```

```
  }

  fclose(fp);

  return SUCCESS;
}


/* This function implements an n-squared synchronization algorithm, adapted
   from the Illinois Fast Messages synchronization algorithm */
int synchronize() {
  /*
    The algorithm proceeds as thus:
    This host sends out sync packets to every other host on the network.
    The host waits until it has received sync_ack packets from all of the
    hosts it has sent sync packets to.  It also waits to receive sync packets
    from every other host, and sends out sync_ack packets to any host that
    has sent it a sync packet.  Sending out sync_packets lets you know that
    every other host is alive, and it also transmits pinned memory information
    to these hosts.  Receiving sync packets and responding with sync_ack
    packets, lets the other hosts know that you are alive and that you
    received their pinned memory information.

    Once this host has received all sync packets (and responded to them) from
    every other host, and has sent out its own sync packets (and gotten
    responses back from other hosts) it is assured that every other host
    is alive, and that every other host knows that it is alive.

    However, to ensure that every other host in the network has also finished
    synchronizing, sync_done packets are sent out.  A host may not leave the
    synchronization loop, and return to the application, until it has both
    sent all of its sync_done packets and received all of the other hosts'
    sync_done packets.
  */

  /* A temporary niu_send_buf memory mapped variable to be used for
     synchronization.  NOT to be confused with the niu_send_buf in
     the Shipping Module */
  volatile PACKET_PTR niu_send_buf;
  volatile UCHAR *niu_send_h;
  volatile UCHAR *niu_send_t;

  volatile ULONG *lanai_received;
  ULONG host_received = 0;

  PACKET_PTR bad_packet;
  PACKET sync_packet;
  PACKET sync_ack_packet;
  PACKET recv_packet;
  PACKET done_packet;

  ULONG my_lanai_received;

  int i;
  const int SYNC_DELAY = 500000;
  int dead_hosts;
  int notified_hosts;
  int done_hosts;
  int delay = SYNC_DELAY;

  /* Used to create the nice little spinny thingy when you are synchronizing */
  char spin_chars[4] = {'|','/','-','\\'};
  int j = 0;

  /* Initialize the following arrays */
  int *alive = (int *)malloc(number_of_hosts*sizeof(int));
  int *toldalive = (int *)malloc(number_of_hosts*sizeof(int));
  int *done_sync = (int *)malloc(number_of_hosts*sizeof(int));

  /* Memory Map these variables on the LANAI */
  niu_send_buf        = (PACKET_PTR)get_lanai_sym("_niu_send_buf");
  niu_send_h          = (UCHAR *)get_lanai_sym("_niu_send_h");
  niu_send_t          = (UCHAR *)get_lanai_sym("_niu_send_t");
  lanai_received      = get_lanai_sym("_lanai_received");
```

```
/* By default you have notified yourself that you are alive, and you also
   know that you are alive. This also implies that you are done with
   yourself */
notified_hosts = 1;
dead_hosts = number_of_hosts-1;
done_hosts = 1;

printf("\nSynchronizing with other hosts: %c ",spin_chars[j++]);
fflush(stdout);

/* Initialize the arrays to indicate that all other hosts are dead, not done
   synchronizing and have not been told that you are alive */
for (i=0; i< number_of_hosts; i++) {
  alive[i] = FALSE;
  toldalive[i] = FALSE;
  done_sync[i] = FALSE;
}

/* Change your entry in the arrays to indicate that you are alive, and
   you have told yourself that you are alive */
alive[host_node_id] = toldalive[host_node_id] = TRUE;
done_sync[host_node_id] = TRUE;

/* Initialize sync packet with default info */
sync_packet.type = htons(NONISO);
sync_packet.subtype = SYNC;
sync_packet.pad1 = 0;
sync_packet.sender = host_node_id;
sync_packet.address = 0;
sync_packet.payload_length = htons((USHORT)sizeof(pinned_mem_info)*2);
sync_packet.pad2 = 0;
sync_packet.credit_info = 0;


/* Initialize sync ack packet with default info.*/
sync_ack_packet.type = htons(NONISO);
sync_ack_packet.subtype = SYNC_ACK;
sync_ack_packet.pad1 = 0;
sync_ack_packet.sender = host_node_id;
sync_ack_packet.address = 0;
sync_ack_packet.payload_length = 0;
sync_ack_packet.pad2 = 0;
sync_ack_packet.credit_info = 0;
memset(&sync_ack_packet.data,' ',MAX_PAYLOAD_SIZE);

/* Initialize sync done packet with default info.*/
done_packet.type = htons(NONISO);
done_packet.subtype = SYNC_DONE;
done_packet.pad1 = 0;
done_packet.sender = host_node_id;
done_packet.address = 0;
done_packet.payload_length = 0;
done_packet.pad2 = 0;
done_packet.credit_info = 0;
memset(&done_packet.data,' ',MAX_PAYLOAD_SIZE);


while ((dead_hosts) || (notified_hosts < number_of_hosts)) {
  /* As long as there are dead hosts in the network, or there are hosts
     that you have not notified of your aliveness keep looping */

  if (*init_stage == BAD_CRC) {
    /* If you receive a bad CRC, bail out.  This should really never happen,
        but eventually support may be put in for bad CRC recovery during
        synchronization */
    printf("ERROR: Sync packet received with bad CRC\n");
    fflush(stdout);
    bad_packet = hostman_hostbase;
    print_out_packet(bad_packet);
    return FAILURE;
  }

  /* do a receive */
```

```
/* check to see if we've received any packets */
my_lanai_received = ntohl(*lanai_received);

while (host_received < my_lanai_received) {
  /* retrieve the packet off of the queue */
  memcpy(&recv_packet, hostman_hostbase + host_received, sizeof(PACKET));

  /* record that we've received it */
  host_received++;

  /* examine the subtype */
  /* if it is an ack packet, set alive flag to true */
  if (recv_packet.subtype == SYNC_ACK) {

      if (!alive[recv_packet.sender]) {
        alive[recv_packet.sender] = TRUE;
        dead_hosts--;
      }
  }
  /* otherwise, if its a plain sync packet.  Retrieve the info and send
      an ack */
  else if (recv_packet.subtype == SYNC) {

      if (!toldalive[recv_packet.sender]) {
        /* copy the noniso dma information into the nst */
        /* note, this will also copy over isotach information, since it is
           second in the payload, and it immediately follows the non_iso
           memory info stuff in the node_info structure */

        memcpy(&nst[recv_packet.sender].remote_noniso,recv_packet.data,
                ntohs(recv_packet.payload_length));

        if (nst[recv_packet.sender].remote_noniso.base == 0) {
          print_out_packet(&recv_packet);
          continue;
        }

        /* now send back an ack */
        sync_ack_packet.route = nst[recv_packet.sender].route;
        /* make sure Lanai's send buffer is not full */

        while (queue_full(*niu_send_h, *niu_send_t, NIU_SEND_SIZE)) {}
        /* insert the packet onto the send queue and inc the tail pointer */
        memcpy((PACKET_PTR)niu_send_buf + *niu_send_t,
                &sync_ack_packet,sizeof(PACKET));
        inc_idx(*niu_send_t, NIU_SEND_SIZE);
        toldalive[recv_packet.sender] = TRUE;
        notified_hosts++;
      }
  }
  /* we could receive a done packet from someone else */
  else if (recv_packet.subtype == SYNC_DONE) {
      done_sync[recv_packet.sender] = TRUE;
      done_hosts++;
  }
  /* else bail and terminate */
  else {
      printf("Received Incorrect Packet from %lu\n", recv_packet.sender);
      fflush(stdout);
      print_out_packet(&recv_packet);
  }
}

if (++delay > SYNC_DELAY) {
  delay = 0;

  /* This is to create the nice spinny thingy for synchronization */
  printf("%c%c",8,8);
  printf("%c ",spin_chars[j]);
  fflush(stdout);
  j = (j+1) % 4;

  /* send out sync packets to all dead hosts */
  for (i = 0; i < number_of_hosts; i++) {
```

```
        if (!alive[i]) {

          /* copy pinned memory info into payload */
          memcpy(&sync_packet.data, &nst[i].local_noniso,
                 2 * sizeof(pinned_mem_info));

          /* add the route */
          sync_packet.route = nst[i].route;

          /* make sure Lanai's send buffer is not full */
          while (queue_full(*niu_send_h, *niu_send_t, NIU_SEND_SIZE)) { }

          /* insert the packet onto the send queue and increment the tail
             pointer */
          memcpy((PACKET_PTR)niu_send_buf + *niu_send_t, &sync_packet,
                 sizeof(PACKET));
          inc_idx(*niu_send_t, NIU_SEND_SIZE);
        }
      }
    }
  }


  /* now tell everyone that we in fact synchronized with everyone else */
  for (i=0; i< number_of_hosts; i++) {

    if (i != host_node_id) {

      /* add the route */
      done_packet.route = nst[i].route;

      /* make sure Lanai's send buffer is not full */
      while (queue_full(*niu_send_h, *niu_send_t, NIU_SEND_SIZE)) ;

      /* insert the packet onto the send queue and increment the tail pointer*/
      memcpy((PACKET_PTR)niu_send_buf + *niu_send_t, &done_packet,
             sizeof(PACKET));

      inc_idx(*niu_send_t, NIU_SEND_SIZE);
    }
  }

  /* now wait for done packets from everyone else */
  while (done_hosts < number_of_hosts) {

    if (++delay > SYNC_DELAY) {
      delay = 0;
      printf("%c%c",8,8);
      printf("%c ",spin_chars[j]);
      fflush(stdout);
      j = (j+1) % 4;
    }

    if (*init_stage == BAD_CRC) {
      /* If you receive a bad CRC, bail out.  This should really never happen,
         but eventually support may be put in for bad CRC recovery during
         synchronization */
      printf("ERROR: Sync_done packet received with bad CRC\n");
      fflush(stdout);
      bad_packet = hostman_hostbase;
      print_out_packet(bad_packet);
      return FAILURE;
    }

    if (host_received < ntohl(*lanai_received)) {

      /* retrieve the packet off of the queue */
      memcpy(&recv_packet, hostman_hostbase + host_received, sizeof(PACKET));

      /* record that we've received it */
      host_received++;

      if (recv_packet.subtype == SYNC_DONE) {
          done_sync[recv_packet.sender] = TRUE;
```

```
          done_hosts++;
        }
        else if (recv_packet.subtype == SYNC) {
          printf("received an extra sync packet from %lu\n", recv_packet.sender);
          fflush(stdout);
        }
        else {
          printf("received other gibberish from %lu\n",recv_packet.sender);
          fflush(stdout);
          print_out_packet(&recv_packet);
        }
      }
    }

    /* This is to create the nice spinny thingy for synchronization */
    printf("%c%c",8,8);
    printf("Done\n");
    fflush(stdout);

    nst[host_node_id].remote_noniso.size = nst[(host_node_id + 1) %
number_of_hosts].remote_noniso.size;

    nst[host_node_id].remote_iso.size = nst[(host_node_id + 1) % number_of_hosts].remote_iso.size;

    *init_stage = FINISHED;
    return SUCCESS;
}


/* Initialize the Isotach System */
int isotach_init() {

    /* Initialize remaining modules */
    shipping_init(nst);
    flow_init(nst);
    send_init();
    receive_init(nst);
    deliver_init(nst);
    barrier_init(nst);

    if ((SYS_TYPE == ISOTACH) || (SYS_TYPE == BOTH)) {
      /* Call Isotach Modules inits */
      iso_send_init();
      iso_flow_init(nst);
      iso_receive_init(iso_base_ptr);
      iso_deliver_init();
      iso_shipping_init(nst);
      iso_signal_init();
      iom_init();
      iso_barrier_init(nst);
    }

    *init_stage = FINISHED;

    /* Print out configuration information */
    print_config();


    /* Isotach Reset and Initialization - This is only executed if SIU == 1
     * (that is, we are using SIU hardware functionality.
     */
#if (SIU == 1)
    if ((SYS_TYPE == ISOTACH) || (SYS_TYPE == BOTH)) {
      /* If you are node 0, send out a reset bs-marker */
      if (host_node_id == 0) {
        printf("Host 0 Sending Out Reset Signal.\n");
        iso_send_mLayer_signal(RESET_SIGNAL);
      }

      /* Wait until the reset signal has been received and then continue */
      printf("Waiting for Reset to Complete.\n");
      while (reset_count == 0) {
        iso_poll();
      }
```

```
    printf("Waiting for SIU...\n");
    sleep(SIU_SLEEP_TIMER);
  }
#endif

  return SUCCESS;
}

/* Deinitialize the Isotach System */
int isotach_deinit() {
  /* Deinitialize remaining modules */
  flow_deinit();
  send_deinit();
  receive_deinit();
  deliver_deinit();
  barrier_deinit();
  shipping_deinit();

  if ((SYS_TYPE == ISOTACH) || (SYS_TYPE == BOTH)) {
    /* Call Isotach Modules deinits */
    iso_send_deinit();
    iso_flow_deinit();
    iso_receive_deinit();
    iso_deliver_deinit();
    iso_shipping_deinit();
    iso_signal_deinit();
    iom_deinit();
    iso_barrier_deinit();
  }

  fprintf (stderr, "Shutting down the Isotach System...\n");

  fprintf (stderr, "Loading the LANai with %s\n", LCPFILE);
  lanai_load_and_reset (0, LCPFILE , 0, 0, 0, callback);

  return SUCCESS;
}


/* The following lines of code load a dummy mcp so that garbage is not
   spewed onto the network by the previous mcp. */
static inline void shutdown(int type) {

  if (type == 0) {
    isotach_deinit();
    exit (1);
  }
  else if (type == 1) {
    fprintf(stderr, "Waiting for CONTROL-C Interrupt...");
    while(1);
  }
}


/* ***********************************************************************
 * End Internal Hostman Functions
 * ***********************************************************************
 */



/* ***********************************************************************
 * The following three functions are the POLL functions
 * ***********************************************************************
 */

/* In the Application Level Header files, the application poll() is mapped to
   one of the three following functions.  If the system requested is purely
   Non-Isotach, then noniso_poll() is used.  If the system is purely Isotach
   then iso_poll() is used.  If both Isotach and Non-Isotach functionality is
   requested by the application, then iso_poll() is used.
   These poll functions are also called by some of the modules, so the polling
   can be done implicitly instead of explictly by the application programmer */
```

```
int iso_poll() {
  receive_poll();
  iso_receive_poll();
  flow_poll();
  iso_flow_poll();
  shipping_poll();
  iso_shipping_poll();
  iso_signal_poll();
  return SUCCESS;
}

int noniso_poll() {
  receive_poll();
  flow_poll();
  shipping_poll();
  return SUCCESS;
}

/* ***********************************************************************
 * End of POLL functions
 * ***********************************************************************
 */



/* ***********************************************************************
 * API Functions - open_net,
 *                 get_node_number,
 *                 get_number_of_hosts,
 *                 get_my_node_number,
 *                 try_close_net
 * ***********************************************************************
 */


/* Returns the host node ID of this host. */
int get_my_node_number() {
  return host_node_id;
}

/* Returns the total number of hosts on the Isotach Network. */
int get_number_of_hosts() {
  return number_of_hosts;
}

/* Returns the maximum payload that the application may send in a noniso or
   Isotach message */
int get_max_payload() {
  return MAX_PAYLOAD_SIZE;
}

/* Returns 1 if the SIU functionality is enabled, 0 if it is disabled */
int get_SIU_state() {
  return SIU;
}

/* Returns a numeric value which indicates which version of the IOM is being
   used.  1 signifies that no host ordering is used, 2 signifies that host
   ordering is being used */
int get_MBM_ver() {
  return MBM_VER;
}

/* Returns the host node ID of the host with name hostname.  This function
   is invoked by the application */
int get_node_number(char *hostname) {
  int i;
  for (i=0; i<number_of_hosts; i++)
    if (!strcmp(nst[i].host_name, hostname))
      return (int)nst[i].node_id;
  return -1;
}
```

```c
/* This function is called by the application to initialize the mLayer and
   to load the LANAI with the Lanai Control Program (LCP).  This MUST be called
   before any messages (Isotach or NonIsotach) can be sent.  The only API
   functions that will work properly before open_net is called are the ones
   listed directly above this function.  (i.e. get_node_number, etc...) */
int open_net(int mode) {
  volatile ULONG *dma_sts;     // Burst mode information for DMA transfer
  int units;
  int max_length;              // Size of Pinned Memory Area on Host in bytes.
  USHORT sts;
  int i;

  /* Initialize the BITS array so that we can extract the nth bit of a byte
     by masking it with BITS[n] */
  for (i = 0; i < 8; i++) {
    BITS[i] = 1 << i;
  }

  /* Is the system ISO, NONISO or BOTH? */
  SYS_TYPE = mode;


  /* ************************************ */
  /* Section 1: lanai_read_symbol_table() */
  /* ************************************ */

  printf ("Loading Netman onto Myricom Interface Board...\n");

  symbol_table = lanai_read_symbol_table(LCPFILE);
  if (!symbol_table) {
    printf ("ERROR: Could not read LANAI symbol table.  Aborting.");
    exit(1);
  }


  /* ********************************** */
  /* Section 2: open_lanai_copy_block() */
  /* ********************************** */

  units = open_lanai_copy_block (&max_length, &sts);

  printf ("units= %d  max_length= %d  sts= %d\n", units, max_length, sts);

  if ((units <= 0) || (max_length <= 0)) {
    printf("ERROR: Could not open LANAI copy block.  Aborting.\n");
    exit(1);
  }
  else if (max_length != SIZE_OF_PINNED_MEMORY) {
    printf("ERROR: LANAI Reports that Pinned Memory is: %lu\n", (ULONG)max_length);
    printf("Change SIZE_OF_PINNED_MEMORY to equal this amount.  Aborting\n");
    shutdown(1);
  }


  /* ********************************************************* */
  /* Section 3: Mapping Hostman Variables                      */
  /* Other module's mapped variables will be done in their own */
  /* initialization functions.                                 */
  /* ********************************************************* */

  lanai0            = (char *)LANAI[0];
  netman_hostbase   = get_lanai_sym("_netman_hostbase");
  netman_maxlen     = get_lanai_sym("_netman_maxlen");
  niu_iso_recv_base = get_lanai_sym("_niu_iso_recv_base");
  dma_sts     = get_lanai_sym("_dma_sts");
  init_stage = (UCHAR *)get_lanai_sym("_init_stage");
  NIU_SYS_TYPE      = (UCHAR *)get_lanai_sym("_NIU_SYS_TYPE");


  /* *********************************************************** */
  /* Section 4: Set init_stage to START                          */
  /* *********************************************************** */

  *init_stage = START;
```

```
/* ************************************************************** */
/* Section 5: Seed beginning of pinned memory woth a set value for */
/*            DMA test.                                           */
/* ************************************************************** */

for (i = 0; i < 8; i++) {
  (ULONG *)(UBLOCK[0])[i] = htonl(0xDEADBEEF);
}

/* *************************************************************** */
/* Section 6: lanai_load_and_reset and initialize the signal handler */
/* *************************************************************** */

lanai_load_and_reset (0, LCPFILE, 0, 0, 0, callback);

signal (SIGINT, handle_sigint);


/* ************************************************************** */
/* Section 7: Send LANAI a pointer to beginning of Pinned Memory   */
/* ************************************************************** */

*netman_hostbase = htonl((ULONG)DBLOCK[0]);
*netman_maxlen   = htonl((ULONG)max_length);
*NIU_SYS_TYPE    = SYS_TYPE;
*dma_sts         = htonl(sts);
hostman_hostbase = (PACKET_PTR)UBLOCK[0];
offset           = (ULONG)hostman_hostbase - (ULONG)ntohl(*netman_hostbase);


/* ***************************************************************** */
/* Section 8: INIT_STAGE: START_LANAI - Tell LANAI to being DMA Test   */
/* ***************************************************************** */

*init_stage = START_LANAI;


/* ***************************************************************** */
/* Section 9: Host Checks pinned memory to ensure DMA engine is        */
/*            working, and terminates with an error message if it is   */
/*            not working.                                             */
/* ***************************************************************** */

/* The host waits until the LANAI has completed the DMA Test */
while (*init_stage != INIT_DMA_TEST) {}

for (i = 0; i < 8; i++) {
  long int j = htonl((ULONG)((UBLOCK[0])[i]));

  if (j != 0x11223344) {
    printf("ERROR: DMA problem at location %d.  Value = %lx\n  Aborting.",
           i, j);
    shutdown(1);
  }
}


/* ****************************************************************** */
/* Section 10: INIT_STAGE: CHECK_DMA_TEST - Tell LANAI that host is     */
/*             finished checking the DMA Engine.                        */
/* ****************************************************************** */

*init_stage = CHECK_DMA_TEST;


/* ****************************************************************** */
/* Section 11: Read in the Network Configuration File and Initialize    */
/*             Network Status Table                                     */
/* ****************************************************************** */

if (initialize_configuration_table() == FAILURE) {
  printf("ERROR: Configuration File could not be processed.  Aborting.\n");
```

```
    shutdown(1);
  }


  /* ******************************************************************** */
  /* Section 12: Allocate Pinned Memory to ISOTACH and NONISOTACH and     */
  /*             print out the network configuration                      */
  /* ******************************************************************** */

  allocate_pinned_memory(max_length);
  print_network_configuration();


  /* ******************************************************************** */
  /* Section 13: Begin Host Synchronization Routine                       */
  /* ******************************************************************** */

  /* Tell the LANAI that we are ready to begin synchronization */
  *init_stage = SYNCHRONIZE;

  /* Synchronization Loop */
  if (synchronize() == FAILURE) {
    printf("ERROR: Hosts would not synchronize properly.  Aborting.\n");
    shutdown(1);
  }


  /* ******************************************************************** */
  /* Section 14: Initialize Isotach Modules and perform an Isotach Reset  */
  /*             if necessary                                             */
  /* ******************************************************************** */

  if (isotach_init() == FAILURE) {
    printf("FAILURE: Isotach Initialization Failed.  Aborting.\n");
    shutdown(1);
  }


  /* ******************************************************************** */
  /* Section 15: Return to the Application                                */
  /* ******************************************************************** */

  return SUCCESS;
}


/* This function attempts to shutdown the Isotach Network through the use of
   a barrier.  If the system is Isotach or Both and using hardware SIUs, the
   system attempts to initiaite an Isotach barrier to ensure that all hosts
   have completed message sending, and then shuts down the system when the
   barrier has completed.  In NonIsotach mode, or when the hardware SIU is
   disabled, this function uses a NonIsotach barrier to accomplish the same
   thing. */
int try_close_net() {
  /* NOTE: It is expected that after an application calls try_close_net, it
     will no longer be SENDING any more messages.  It may need to recieve more
     messages, but by the time the barrier is completed there should be no
     further messages to complete. */

  static int initiated = FALSE;
  int i;

  /* If we are NONISOTACH or do not use an SIU, then use this barrier method */
  if ((SYS_TYPE == NONISOTACH) || (MBM_VER < 2)) {

    /* If the barrier has not been initiated yet, initiate it. */
    if (!initiated && queue_empty(send_h, send_t)) {
      initiate_barrier();
      initiated = TRUE;
    }

    /* Poll no matter what */
    noniso_poll();
```

```
    /* If there is nothing left in the delivery_q, then check to see if the
       barrier is completed.  If it is, shutdown, otherwise, inform the
       application to keep checking */
    if (queue_empty(delivery_h,delivery_t)) {
      if (barrier_completed() == SUCCESS) {
         isotach_deinit();
         return SUCCESS;
      }
      else {
         return FAILURE;
      }
    }
    else {
      return FAILURE;
    }
  }
  /* Else, if we are using an SIU in an ISOTACH system, use this method */
  else if (((SYS_TYPE == ISOTACH) || (SYS_TYPE == BOTH)) && (MBM_VER > 1)) {
    return SUCCESS;

    /* If we have not initiated an iso_barrier yet, try to.  Only set initiated
       to TRUE if we can start the barrier on the
       SHUTDOWN_BARRIER channel.  SHUTDOWN_BARRIER is defined in iso_barrier's
       exports.h */

    if ((!initiated) && (queue_empty(iso_send_h, iso_send_t))) {
      if (iso_mLayer_barrier(SHUTDOWN_BARRIER, STRONG) == FAILURE) {
         return FAILURE;
      }

      initiated = TRUE;
    }

    /* Poll No matter what */
    iso_poll();

    /* Peek at the top of the iso_delivery Queue.  If the top item is a
       BS_MARKER and it has bits that indicate that the SHUTDOWN_BARRIER has
       completed, then shutdown.  Otherwise, return FAILURE to the application
       so that the application keeps checking. */
    if (!queue_empty(iso_delivery_h, iso_delivery_t)) {
      if ((iso_delivery_t->subtype == BS_MARKER)) {
         printf("got a bsmarker for shutdown");
         printf("bits->%0X\n", ((bsnotice *)iso_delivery_t)->bits);
         if (((bsnotice *)iso_delivery_t)->bits & BITS[SHUTDOWN_BARRIER] == 1) {
           isotach_deinit();
           return SUCCESS;
         }
      }
    }
    else {
      return FAILURE;
    }
  }

  /* Return FAILURE just in case... */
  return FAILURE;
}


/* ************************************************************************
 * End API Functions
 * ************************************************************************
 */



/* ************************************************************************
 * Utility Functions (Debugging, status printing)
 * ************************************************************************
 */

/* This prints out the network configuration, and some basic system
   configuration.  Useful to see, but not necessary.  Can be removed from
```

```
   open_net() */
void print_network_configuration() {
  int i;

  printf("Welcome to Isotach\n");
  printf("MAX_PAYLOAD_SIZE = %d\n",MAX_PAYLOAD_SIZE);

  if (MBM_VER == 1)
    printf("There is no support for self messages and Isotach ordering\n");

  if (SIU == 0)
    printf("There is no support for a hardware SIU\n");

  printf("I am node %s(%lu)\n\n",nst[host_node_id].host_name,host_node_id);

  for (i = 0; i < number_of_hosts; i++) {
    printf("Node ID: %lu\tName: %s\tRoute: %08X\n",
             nst[i].node_id,nst[i].host_name,ntohl(nst[i].route));
  }

  return;
}


/* This prints out the Pinned Memory Configuration for all hosts.  Again, it
   is often useful to see this, but it is not necessary.  Disable by commenting
   out the call in open_net() */
void print_config() {
  int i;

  printf("NONISO Pinned Memory Configuration:\n");

  for (i = 0; i < number_of_hosts; i++) {
    printf("Node ID: %lu\tBase: %8u\tLength: %5lu\tBaseL: %8u\tLengthL: %5lu\n",nst[i].node_id,
(ULONG)nst[i].remote_noniso.base, nst[i].remote_noniso.size, (ULONG)nst[i].local_noniso.base,
nst[i].local_noniso.size);
  }

  printf("ISO Pinned Memory Configuration:\n");

  for (i = 0; i < number_of_hosts; i++) {
    printf("Node ID: %lu\tBase: %8u\tLength: %5lu\tBaseL: %8u\tLengthL: %5lu\n",nst[i].node_id,
(ULONG)nst[i].remote_iso.base, nst[i].remote_iso.size, (ULONG)nst[i].local_iso.base,
nst[i].local_iso.size);
  }

  return;
}


/* ***********************************************************************
 * End Utility Functions
 * ***********************************************************************
 */


/* ***********************************************************************
 * Misc. Functions: handle_sigint
 * ***********************************************************************
 */

/*
 * This is the signal handler for CONTROL-C before FM_initialize has been
 * executed.  It does not call iso_print_stats.  It simply notifies the user
 * that a sigint was received and then loads the LANai with a dummy_lcp
 * to reset it and prevent miscellaneous packets from being sent out.
 * Perry (9/15/99)
 */

void handle_sigint (int s) {
  shutdown(0);
}

/* ***********************************************************************
```

```
* End Misc. Functions
* ************************************************************************
*/
```

api/send/exports.h

```c
/*
 * -------------------------------------------------------------------------
 *
 * Isotach Module  : Send Module Exported Functions/Variables
 * Isotach Layer   : API
 * Isotach Version : Version 3
 *
 * -------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/api/send/exports.h,v $
 *       $Revision: 1.6 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/02 01:24:00 $
 *
 * -------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This file contains all of the exported functions, variables and data
 * structures for the given module.  Other modules which need to access these
 * variables/functions should include this header file.
 *
 * -------------------------------------------------------------------------
 *
 * COMMENTS:
 * Included by the Hostman Module and the Flow Module
 *
 * -------------------------------------------------------------------------
 */

#ifndef SEND_EXPORTS_H
#define SEND_EXPORTS_H

/* Hostman's open_net calls this to initialize the send module */
int send_init();
int send_deinit();

/* The application calls this to perform a non-iso send */
int send(int target, void *data, int size);

#endif
```

api/send/locals.h

```
/*
 * --------------------------------------------------------------------------
 *
 * Isotach Module  : Send Module Local Variable Header File
 * Isotach Layer   : API
 * Isotach Version : Version 3
 *
 * --------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/api/send/locals.h,v $
 *       $Revision: 1.5 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/02 01:24:00 $
 *
 * --------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This file contains all of the local variable and data structure definitions.
 * It also includes all other header files necessary for the module's function.
 * It also prototypes all functions internal to that module.
 *
 * --------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * --------------------------------------------------------------------------
 */

#ifndef SEND_LOCALS_H
#define SEND_LOCALS_H

#include <hostman/host_utils.h>
#include <api/send/exports.h>
#include <niu_interface/shipping/exports.h>
#include <processing/flow/exports.h>

#endif
```

api/send/send.c

```
/*
 * ---------------------------------------------------------------------------
 *
 * Isotach Module  : Send Module
 * Isotach Layer   : API
 * Isotach Version : Version 3
 *
 * ---------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/api/send/send.c,v $
 *       $Revision: 1.6 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:15 $
 *
 * ---------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This module exports the API send function, which is called by the
 * applications.  It provides the functionality for sending non-iso messages.
 * Each message is built into the send_buf at send_t.  Then, the message is
 * examined by Flow Module to determine if there is enough send credits to send
 * the message.  After the message has cleared Flow, it is sent to shipping
 * where the route is assigned and the packet is sent to the LANAI via the
 * memory-mapped niu_send_buf.  The Netman module reads packets off of the
 * niu_send_buf and transmits them onto the network.
 *
 * ---------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * ---------------------------------------------------------------------------
 */

#include <api/send/locals.h>

/* Module De-Initialization Function */
int send_deinit() {
  /* Stub Function */
  return SUCCESS;
}

/* Module Initialization Function */
int send_init() {
  return SUCCESS;
}

/* The Application calls this to initiate an Non-Isotach Message Based Packet.
   Self-Messages are not supported, and the target must be a valid node-id in
   the network. *data points to the data to be sent, and size represents the
   amount of bytes to be sent. */
int send(int target, void *data, int size) {

  /* If the send_buf is full, first invoke the poll() function and then return
     a FAILURE code to the application */
  if (queue_full(send_h, send_t, SEND_BUF_SIZE)) {
    if (SYS_TYPE == NONISOTACH)
      noniso_poll();
    else
      iso_poll();

    //    printf("Leaving send(): send_buf has reached maximum capacity.\n");
    return FAILURE;
  }

  /* If the target host for the send is not in the allowed range of hosts, OR
     is to this host, poll and then return a FAILURE */
  if ((target < 0) || (target >= number_of_hosts) || (target == host_node_id)){
    if (SYS_TYPE == NONISOTACH)
```

```
      noniso_poll();
    else
      iso_poll();

    printf("Leaving send(): Invalid recepient.\n");
    return FAILURE;
  }

  /* If the payload size is greater than MAX_PAYLOAD_SIZE, poll and then
     return a FAILURE code */
  if (size > MAX_PAYLOAD_SIZE) {
    if (SYS_TYPE == NONISOTACH)
      noniso_poll();
    else
      iso_poll();

    printf("Leaving send(): Payload size too large.\n");
    return FAILURE;
  }


  /* Create the Packet to be sent in the tail of the send_buf */

  /* The target is written into the route, so that shipping will know which
     entry off of routes[] to use */
  send_t->route   = (ULONG)target;
  send_t->type    = htons(NONISO);
  send_t->subtype = NONISO_MBM;
  send_t->sender  = host_node_id;

  /* The payload length, and contents of the payload are written into the
     packet */
  send_t->payload_length = htons((USHORT)size);
  memcpy(&send_t->data, data, size);

  //printf("Sent packet to %lu\n", send_t->route);

  /* The send_t is incremented to tell flow that there is a new packet to be
     examined */
  inc_ptr(send_t, SEND_BUF_SIZE, send_buf);


  /* Poll before exiting */
  if (SYS_TYPE == NONISOTACH)
    noniso_poll();
  else
    iso_poll();

  return SUCCESS;
}
```

api/deliver/exports.h

```
/*
 * -------------------------------------------------------------------------
 *
 * Isotach Module  : Deliver Module Exported Functions/Variables
 * Isotach Layer   : API
 * Isotach Version : Version 3
 *
 * -------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/api/deliver/exports.h,v $
 *       $Revision: 1.3 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:15 $
 *
 * -------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This file contains all of the exported functions, variables and data
 * structures for the given module.  Other modules which need to access these
 * variables/functions should include this header file.
 *
 * -------------------------------------------------------------------------
 *
 * COMMENTS:
 * Included by the Hostman Module
 *
 * -------------------------------------------------------------------------
 */

#ifndef DELIVER_EXPORTS_H
#define DELIVER_EXPORTS_H

int DELIVERY_SIZE;

typedef struct {
  int sender_id;     // The originator of this message
  int size;          // The length of the data being pointed to by data_ptr
                     // if this is a -1, then the date is stored in data
  union msg_tag {
    void *data_ptr;  // Pointer to pinned memory where data is kept
    int   data;      // An actual word of data
  } msg;
} noniso_mbm;


/* The Delivery Queue.  Messages are placed onto this queue for delivery to
   the application.  The application reads messages off of this queue through
   calls to receive() */
PACKET_PTR *delivery_q;
ULONG delivery_h;
ULONG delivery_t;

int receive(noniso_mbm *recv_msg);
int deliver_init(node_info *nst);
int deliver_deinit();


#endif
```

api/deliver/locals.h

```
/*
 * ---------------------------------------------------------------------------
 *
 * Isotach Module  : Deliver Module Local Variable Header File
 * Isotach Layer   : API
 * Isotach Version : Version 3
 *
 * ---------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/api/deliver/locals.h,v $
 *       $Revision: 1.4 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:15 $
 *
 * ---------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This file contains all of the local variable and data structure definitions.
 * It also includes all other header files necessary for the module's function.
 * It also prototypes all functions internal to that module.
 *
 * ---------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * ---------------------------------------------------------------------------
 */

#ifndef DELIVER_LOCALS_H
#define DELIVER_LOCALS_H

#include <hostman/host_utils.h>
#include <api/deliver/exports.h>
#include <niu_interface/receive/exports.h>
#include <processing/flow/exports.h>

/* A pointer to the last packet delivered, so it can be deleted when the next
   call to recieve() is processed.  This gives the application time to copy
   the message contents into local memory, before it is deleted from pinned
   memory. */
PACKET_PTR last_packet;

#endif
```

api/deliver/deliver.c

```
/*
 * ------------------------------------------------------------------------
 *
 * Isotach Module  : Deliver Module
 * Isotach Layer   : API
 * Isotach Version : Version 3
 *
 * ------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/api/deliver/deliver.c,v $
 *       $Revision: 1.2 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:15 $
 *
 * ------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 *
 * ------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * ------------------------------------------------------------------------
 */

#include <api/deliver/locals.h>

/* Deliver Module DeInitialization Function */
int deliver_deinit() {
  return SUCCESS;
}

/* Delviery Module Initialization Function */
int deliver_init(node_info *nst) {
  DELIVERY_SIZE = number_of_hosts * nst[host_node_id].remote_noniso.size;
  delivery_q = (PACKET_PTR *)malloc(DELIVERY_SIZE * sizeof(PACKET_PTR));

  if (delivery_q == (PACKET_PTR *)NULL) {
    printf("Could not allocate space for delivery_q! Aborting.\n");
    shutdown(1);
  }

  last_packet = NULL;
  delivery_t = 0;
  delivery_h = 0;

  return SUCCESS;
}

/* API Function called by the Application to recieve pending messages from the
   delivery_q.  The pending message is placed in the noniso_mbm structure,
   which has a data pointer to pinned memory where the body of the message is
   stored.  The application MUST copy the data out of pinned memory before the
   next call to receive or it will be lost. */
int receive(noniso_mbm *recv_msg) {

  /* If there is a packet waiting to be deleted, delete it. */
  if (last_packet != NULL) {
    delete_packet(last_packet->sender, last_packet);
    last_packet = NULL;
  }

  if (SYS_TYPE == NONISOTACH)
    noniso_poll();
  else
    iso_poll();

  /* Nothing to receive... */
```

```
  if (queue_empty(delivery_h, delivery_t)) {
    return FAILURE;
  }
  /* Take the message to be received off of the delivery_q and place it in the
     noniso_mbm structure passed into receive() */
  else {
    last_packet = delivery_q[delivery_h];

    inc_idx(delivery_h, DELIVERY_SIZE);

    recv_msg->sender_id    = last_packet->sender;
    recv_msg->size         = ntohs(last_packet->payload_length);
    recv_msg->msg.data_ptr = last_packet->data;
  }

  return SUCCESS;
}
```

api/barrier/exports.h

```
/*
 * ---------------------------------------------------------------------------
 *
 * Isotach Module  : Barrier Module Exported Functions/Variables
 * Isotach Layer   : Processing
 * Isotach Version : Version 3
 *
 * ---------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/flow_control/flow/exports.h,v $
 *       $Revision: 1.6 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:15 $
 *
 * ---------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This file contains all of the exported functions, variables and data
 * structures for the given module.  Other modules which need to access these
 * variables/functions should include this header file.
 *
 * ---------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * Included by the Hostman and Receive Modules
 *
 * ---------------------------------------------------------------------------
 */

#ifndef BARRIER_EXPORTS_H
#define BARRIER_EXPORTS_H


/* receiving calls this when it receives a barrier packet */
void process_barrier(ULONG sender);

/* Hostman's open_net calls this to initialize the Flow Module */
int barrier_init();
int barrier_deinit();

/* called by mLayer and perhaps eventually the application */
int initiate_barrier();

/* a "poll" function that returns success when we have received
   barrier packets from every host */
int barrier_completed();

#endif
```

api/barrier/locals.h

```
/*
 * ---------------------------------------------------------------------------
 *
 * Isotach Module  : Barrier Module Local Variable Header File
 * Isotach Layer   : Processing
 * Isotach Version : Version 3
 *
 * ---------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/flow_control/flow/locals.h,v $
 *       $Revision: 1.5 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:15 $
 *
 * ---------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This file contains all of the local variable and data structure definitions.
 * It also includes all other header files necessary for the module's function.
 * It also prototypes all functions internal to that module.
 *
 * ---------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * ---------------------------------------------------------------------------
 */

#ifndef BARRIER_LOCALS_H
#define BARRIER_LOCALS_H

#include <hostman/host_utils.h>
#include <api/barrier/exports.h>
#include <niu_interface/shipping/exports.h>

/* array of flags used to store receipt of barrier */
int *barrier_recv;

/* flag to signify whether we are in a barrier or not */
int in_barrier;

PACKET_PTR *barrier_packet_base;

#endif
```

api/barrier/barrier.c

```
/*
 * ------------------------------------------------------------------------
 *
 * Isotach Module  : Barrier Module
 * Isotach Layer   : Processing
 * Isotach Version : Version 3
 *
 * ------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/flow_control/flow/flow.c,v $
 *       $Revision: 1.3 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:15 $
 *
 * ------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This module provides functionality for a non-isotach barrier, initially
 * used by try_close_net in noniso, but may eventually used by an
 * application.  It provides init and de-init functions to be called by
 * hostman, and initiate function that sends barrier packets to all hosts,
 * a receive function that the receive module can call to notify us that
 * we have received a barrier packet, and a poll function that allows the
 * initiator of the barrier to see when it completes.
 *
 * ------------------------------------------------------------------------
 *
 * COMMENTS:
 * Included by the Hostman Module, Shipping Module and Send Module
 *
 * ------------------------------------------------------------------------
 */

#include <api/barrier/locals.h>

/* receiving calls this when it receives a barrier packet */
void process_barrier(ULONG sender) {
  barrier_recv[sender] = TRUE;
  return;
}

/* Barrier Module Initialization Function */
int barrier_init(node_info *nst) {
  int i;

  barrier_recv = (int *)malloc(number_of_hosts * sizeof(int));
  barrier_packet_base = (PACKET_PTR *)malloc(number_of_hosts *
                                                   sizeof(PACKET_PTR));

  in_barrier = FALSE;

  /* Set all of the barrier recieves to FALSE and set each barrier_packet_base
     to the appropriate slot in the noniso buffer */
  for (i = 0; i < number_of_hosts; i++) {
    barrier_recv[i] = FALSE;
    barrier_packet_base[i] = nst[i].remote_noniso.base + 2;
  }

  return SUCCESS;
}

/* Barrier Module DeInitialization Function */
int barrier_deinit() {
  free(barrier_recv);
  return SUCCESS;
}

/* Initiates a noniso barrier.  Called now by the try_close_net function
```

```
   in a purely NONISO system.  May also be called directly by the application,
   though that feature is yet untested */
int initiate_barrier() {
  int i;
  int iterations = 0;
  PACKET barrier_packet;

  /* Cannot Initiate a new barrier, if we are already in one */
  if (in_barrier) {
    return FAILURE;
  }

  in_barrier = TRUE;

  /* Initialize all of the packet fields... */
  barrier_packet.type           = htons(NONISO);
  barrier_packet.subtype        = BARRIER;
  barrier_packet.pad1           = 0;
  barrier_packet.sender         = host_node_id;
  barrier_packet.payload_length = 0;
  barrier_packet.pad2           = 0;
  barrier_packet.credit_info    = NULL_CREDIT;

  /* Send the barrier packet out to each node in the network, and to yourself
     set your barrier_recv value to TRUE to indicate that you have sent the
     barrier to yourself. */
  for (i = 0; i < number_of_hosts; i++) {
    if (i == host_node_id)
      barrier_recv[i] = TRUE;
    else {
      barrier_packet.route   = i;
      barrier_packet.address = (ULONG)htonl((ULONG)barrier_packet_base[i]);

      /* ship_packet only attempts to send once...  we want this packet to
           go out, so we will keep calling */
      while (ship_packet(&barrier_packet) == FAILURE) {
          if (++iterations % 10 == 0)
            printf("Looping in ship_packet (barrier)\n");
      }
      printf("Sending Barrier Packet to %d\n", i);
    }
  }
  return SUCCESS;
}


/* a "poll" function that returns success when we have received
   barrier packets from every host */
int barrier_completed() {
  int i;
  int done = TRUE;

  if (SYS_TYPE == BOTH)
    iso_poll();
  else
    noniso_poll();

  /* If you haven't initiated a barrier, then you cannot check for barrier
     completion */
  if (in_barrier == FALSE)
    return FAILURE;

  /* Only if you have recieved a barrier packet from ALL hosts, should you
     complete the barrier */
  for (i = 0; i < number_of_hosts; i++) {
    if (barrier_recv[i] == FALSE) {
      done = FALSE;
      break;
    }
  }

  if (done) {
    /* no longer in a barrier */
    in_barrier = FALSE;
```

```
    /* reset everything to false */
    for (i = 0; i < number_of_hosts; i++)
      barrier_recv[i] = FALSE;

    return SUCCESS;
  }
  else
    return FAILURE;
}
```

api/iso_send/exports.h

```
/*
 * ---------------------------------------------------------------------------
 *
 * Isotach Module  : Iso-Send Module Exported Functions/Variables
 * Isotach Layer   : API
 * Isotach Version : Version 3
 *
 * ---------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/api/send/exports.h,v $
 *       $Revision: 1.6 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/02 01:24:00 $
 *
 * ---------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This file contains all of the exported functions, variables and data
 * structures for the given module.  Other modules which need to access these
 * variables/functions should include this header file.
 *
 * ---------------------------------------------------------------------------
 *
 * COMMENTS:
 * Included by the Hostman Module and the Flow Module
 *
 * ---------------------------------------------------------------------------
 */

#ifndef ISO_SEND_EXPORTS_H
#define ISO_SEND_EXPORTS_H

/* The max number of messages stored in the hit buffer.  This parameter can
   be changed, and should be if there is overflow in the hit buffer. */
#define HIT_BUF_SIZE 512

/* This is set to TRUE if we are currently in the middle of sending out an
   Isochron, and FALSE if we are not. */
int mid_net_isochron;

/* The hit buffer is used to store locally executed messages/srefs.  For each
   self message sent, there is an entry for that message in the hit buffer. */
PACKET     hit_buf[HIT_BUF_SIZE];
PACKET_PTR hit_h;
PACKET_PTR hit_t;

int iso_send(int target, void *data, int size, int last_in_isochron);
int iso_write();
int iso_sched();
int iso_assign();
int iso_end();
int iso_send_init();
int iso_send_deinit();

#endif
```

api/iso_send/locals.h

```c
/*
 * --------------------------------------------------------------------------
 *
 * Isotach Module  : Iso-Send Module Local Variable Header File
 * Isotach Layer   : API
 * Isotach Version : Version 3
 *
 * --------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/api/send/locals.h,v $
 *       $Revision: 1.5 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/02 01:24:00 $
 *
 * --------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This file contains all of the local variable and data structure definitions.
 * It also includes all other header files necessary for the module's function.
 * It also prototypes all functions internal to that module.
 *
 * --------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * --------------------------------------------------------------------------
 */

#ifndef ISO_SEND_LOCALS_H
#define ISO_SEND_LOCALS_H

#include <hostman/host_utils.h>
#include <api/iso_send/exports.h>
#include <processing/iom/exports.h>
#include <processing/iso_flow/exports.h>
#include <niu_interface/shipping/exports.h>
#include <niu_interface/iso_shipping/exports.h>

/* This is initially set to 1, and alternates between 0 and 1 each time an
   Isotach packet is sent.  This is flipped each Isotach packet to maintain
   sequential consistency */
int seq_con_set;

/* A counter (0-255) used to track Isochron ID's. Initialized to zero and
   incremented for each Isochron */
UCHAR net_isochrons_sent;

/* This variable is set to TRUE if the EOI status of the item in the tail slot
   of iso_send_buf has been determined, or if iso_send_buf is empty.  It is
   FALSE otherwise.  Initialized to TRUE and set to FALSE when a packet is
   constructed in iso_send_buf. */
int EOI_decided;

/* The number of self-messages in the current Isochron. */
int self_count;


void EOI_found();
int start_iso_packet();

#endif
```

api/iso_send/iso_send.c

```c
/*
 * ------------------------------------------------------------------------
 *
 * Isotach Module  : Iso-Send Module
 * Isotach Layer   : API
 * Isotach Version : Version 3
 *
 * ------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/api/send/send.c,v $
 *       $Revision: 1.6 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:15 $
 *
 * ------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This module exports the API functions:
 *
 * iso_send()
 * iso_write()
 * iso_sched()
 * iso_assign()
 * iso_end()
 *
 * See API Documentation for Details
 *
 * ------------------------------------------------------------------------
 *
 * COMMENTS: Currently only Iso-MBM messages are supported
 *
 * ------------------------------------------------------------------------
 */

#include <api/iso_send/locals.h>

/* This function is called at the end of each Isochron */
void EOI_found() {

  if (EOI_decided == FALSE) {
    /* If EOI_decided is FALSE, then make this packet the End Of Isochron
       packet, by setting the EOI bit in the packet header. */
    iso_send_t->prefix |= EOI_MASK;

    /* If a send-delta is required (see SIU Spec) it should be added here.
       The random number was purely for testing puposes. */
    //iso_send_t->prefix |= (rand()%16);

    /* Increment the tail of the iso_send_buf to clear the packet. */
    inc_ptr(iso_send_t, ISO_SEND_BUF_SIZE, iso_send_buf);
    EOI_decided = TRUE;
  }

  if (mid_net_isochron == TRUE) {
    /* If we have sent any network messages in this Isochron, call
       post_isochron with the self_count number */
    post_isochron(mid_net_isochron, self_count, net_isochrons_sent);

    /* Increment the number of net isochrons sent. */
    net_isochrons_sent = (net_isochrons_sent + 1) % isochron_allowance;
    mid_net_isochron = FALSE;
  }
  else {
    /* If it is not a network isochron, then just post the isochron without
       incrementing the net_isochrons_sent */
    post_isochron(mid_net_isochron, self_count, -1);
  }
```

```
  /* Reset self-count back to zero for the next Isochron */
  self_count = 0;
  return;
}

/* This function is called for each Isotach Packet that needs to be sent. */
int start_iso_packet() {

  /* If this is not the EOI packet, send the previous packet to shipping by
     incrementing the iso_send_buf tail. */
  if (EOI_decided == FALSE) {
    inc_ptr(iso_send_t, ISO_SEND_BUF_SIZE, iso_send_buf);
  }

  EOI_decided = FALSE;

  /* Initialize the prefix... */
  iso_send_t->prefix = 0;

  /* If mid_net_isochron is FALSE, this must be the first Isotach packet in the
     Isochron to be sent on the network.  In this case, mark this packet as a
     SOI (start of Isochron) packet. */
  if (mid_net_isochron == FALSE) {
    /* We are now in the middle of a network Isochron... */
    mid_net_isochron = TRUE;

    if (seq_con_set == TRUE) {
      /* Set the seq_con_bit if it is TRUE and then flip it to false.
         This alternates the seq_con_bit for every packet sent. */
      iso_send_t->prefix |= (SEQ_CON_SET_MASK | SEQ_CON_MASK | LOG_TS_MASK);
      seq_con_set = FALSE;
    }
    else {
      /* Otherwise just set the seq_con_mask and log_ts_mask.  See the
         hardware spec for more details on these... */
      iso_send_t->prefix |= (SEQ_CON_MASK | LOG_TS_MASK);
      seq_con_set = TRUE;
    }

    /* Set the Isochron ID in the prefix */
    iso_send_t->prefix |= ((ULONG)net_isochrons_sent << 16);

    /* If you want to use send deltas, set them here. */
    // iso_send_t->prefix |= 0x00000001;
  }

  return SUCCESS;
}

/* The application calls this to indicate that the last packet has been sent
   in the current Isochron. */
int iso_end() {
  /* Call EOI_found to complete the current Isochron, and then call poll. */
  EOI_found();
  iso_poll();

  return SUCCESS;
}

/* Module de-initialization Function */
int iso_send_deinit() {
  return SUCCESS;
}

/* Module initialization Function */
int iso_send_init() {
  int i;

  seq_con_set         = TRUE;
  net_isochrons_sent = 0;
  EOI_decided         = TRUE;
  self_count          = 0;
  mid_net_isochron    = FALSE;
```

```
  /* Initialize all pointers into the hit_buf */
  hit_t = hit_h = hit_buf;

  /* Set up the hit_buf */
  for (i = 0; i < HIT_BUF_SIZE; i++) {
    hit_buf[i].type    = (USHORT)htons(NONISO);
    hit_buf[i].subtype = ORDERED;
    hit_buf[i].sender  = host_node_id;
  }

  return SUCCESS;
}

/* This function is called by the application to send out an MBM packet.
   The arguments are a host ID (target), a pointer to the data to be sent and
   the size of that data.  If last_in_isochron is TRUE, then this packet is
   the last packet in the isochron.  Otherwise, there will be other packets
   in the isochron or a call to iso_end(). */
int iso_send(int target, void *data, int size, int last_in_isochron) {
  /* If the target host for the send is not in the allowed range of hosts, OR
     is to this host, poll and then return a FAILURE */
  if ((target < 0) || (target >= number_of_hosts)) {
    iso_poll();
    printf("Leaving iso_send(): Invalid recepient.\n");
    return FAILURE;
  }

  /* If the payload size is greater than MAX_PAYLOAD_SIZE, poll and then
     return a FAILURE code */
  if (size > MAX_PAYLOAD_SIZE) {
    iso_poll();
    printf("Leaving iso_send(): Payload size too large.\n");
    return FAILURE;
  }

  /* Is this message a self-message?  If not, then skip this section and go to
     the section for sending NON-self-messages */

  if (target == host_node_id) {
    /* If the target is this host, make sure there is enough space in the
       hit_buf If there is not enough space, return FAILURE */

    if (queue_full(hit_h, hit_t, HIT_BUF_SIZE)) {
      iso_poll();
      printf("Leaving iso_send(): Hit Buffer is full for self-message.\n");
      printf("hit_h = %lu, hit_t = %lu, size =
%d\n",(ULONG)hit_h,(ULONG)hit_t,queue_size(hit_h,hit_t,HIT_BUF_SIZE));
      exit(1);
      return FAILURE;
    }

    /* Construct the self-message in the tail of the hit_buf */
    hit_t->type    = htons(NONISO);
    hit_t->subtype = ORDERED;
    hit_t->sender  = host_node_id;

    /* The payload length, and contents of the payload are written into the
       packet */
    hit_t->payload_length = htons((USHORT)size);
    memcpy(&hit_t->data, data, size);

    //printf("Sent ordered packet to Self\n");

    inc_ptr(hit_t, HIT_BUF_SIZE, hit_buf);

    self_count++;
  }
  /* If this is NOT a self-message, execute the following code: */
  else {
    /* Make sure there is room in the ord_send_buf and iso_send_buf before
       proceeding */

    if ((!EOI_decided)
         && ((queue_full(ord_send_h, ord_send_t, ORD_SEND_BUF_SIZE))
```

```
              || (queue_full(iso_send_h, iso_send_t + 1, ISO_SEND_BUF_SIZE)))) {
        iso_poll();
        //printf("Leaving iso_send(): Either Ord or Iso send buffers are full.\n");
        return FAILURE;
      }
      else  if ((queue_full(ord_send_h, ord_send_t, ORD_SEND_BUF_SIZE)) ||
               (queue_full(iso_send_h, iso_send_t, ISO_SEND_BUF_SIZE))) {
        iso_poll();
        //printf("Leaving iso_send(): Either Ord or Iso send buffers are full.\n");
        return FAILURE;
      }

      if (start_iso_packet() == FAILURE) {
        iso_poll();
        printf("Leaving iso_send(): start_iso_packet() Failed.\n");
      }

      /* Start building the Isotach Packet in the iso_send_buf */
      iso_send_t->route2          = target;
      iso_send_t->packet.subtype = ISO_MBM;
      iso_send_t->packet.sender  = (USHORT)host_node_id;


      /* Create the Packet to be sent in the tail of the ord_send_buf */

      /* The target is written into the route, so that shipping will know which
         entry off of routes[] to use */
      ord_send_t->route    = (ULONG)target;

      /* The payload length, and contents of the payload are written into the
         packet */
      ord_send_t->payload_length = htons((USHORT)size);
      memcpy(&ord_send_t->data, data, size);

      //    printf("Sent ordered packet to %lu\n", ord_send_t->route);

      /* The ord_send_t is incremented to tell flow that there is a new packet
         to be examined */
      inc_ptr(ord_send_t, ORD_SEND_BUF_SIZE, ord_send_buf);
    }

    /* If this is the last packet in the isochron (set by last_in_isochron)
       finish the isochron by sending out an EOI if this was a network
       isochron. */
    if (last_in_isochron == TRUE) {
      EOI_found();
    }

    /* Always poll on every API call... */
    iso_poll();

    return SUCCESS;
  }

  /* The following Functions pertain to the Isotach Shared Memory Manager
     Interface (SMM).  This interface has not been implemented yet in this
     version of the code.  These functions are stubs which need to be filled
     in. */

  int iso_read() {

    return SUCCESS;
  }

  int iso_write() {

    return SUCCESS;
  }

  int iso_assign() {

    return SUCCESS;
  }
```

```
int iso_sched() {

  return SUCCESS;
}
```

api/iso_deliver/exports.h

```
/*
 * -------------------------------------------------------------------------
 *
 * Isotach Module  : Iso-Deliver Module Exported Functions/Variables
 * Isotach Layer   : API
 * Isotach Version : Version 3
 *
 * -------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/api/send/exports.h,v $
 *       $Revision: 1.6 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/02 01:24:00 $
 *
 * -------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This file contains all of the exported functions, variables and data
 * structures for the given module.  Other modules which need to access these
 * variables/functions should include this header file.
 *
 * -------------------------------------------------------------------------
 *
 * COMMENTS:
 * Included by the Hostman Module and the Flow Module
 *
 * -------------------------------------------------------------------------
 */

#ifndef ISO_DELIVER_EXPORTS_H
#define ISO_DELIVER_EXPORTS_H

/* Size in Bytes of the Iso Delivery Queue.  Changing this value changes the
   allocation of memory in allocate_pinned_memory in hostman.  The iso delivery
   queue is protected from overflow by Isotach Flow Control.  Because of this
   the number of Isotach Credits is calculated based upon this number. It is
   set sufficiently large to prevent Isotach Pinned memory from being
   constrained by it. */
#define ISO_DELIVERY_SIZE 16384

/* The Isotach Delivery Queue.  Messages are processed by the mLayer, and then
   placed on this queue for delivery to the application.  The application reads
   messages off of this queue through calls to iso_receive. */
PACKET_CORE_2 iso_delivery_q[ISO_DELIVERY_SIZE];
PACKET_CORE_2 *iso_delivery_h;
PACKET_CORE_2 *iso_delivery_t;

/* The tail pointer of the Ordered Message Receive Buffers. */
PACKET_PTR *ord_receive_t;

typedef struct {
  unsigned short sender;        // The sending ID of the message
  unsigned short length;        // The length of the message
  void *data;                   // Pointer to the data in pinned memory
} iso_msg;

typedef struct {
  unsigned char  tag;           // The type of the structure (pointer, bs)

  union {
    unsigned char bits;         // Barrier and Signal Bits
    iso_msg msg;                // Pointer to the Iso-data
  } info;
} iso_mbm;

int iso_receive(iso_mbm *data);
int iso_deliver_init();
int iso_deliver_deinit();
```

```
#endif
```

api/iso_deliver/locals.h

```
/*
 * -------------------------------------------------------------------------
 *
 * Isotach Module  : Iso-Deliver Module Local Variable Header File
 * Isotach Layer   : API
 * Isotach Version : Version 3
 *
 * -------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/api/send/locals.h,v $
 *       $Revision: 1.5 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/02 01:24:00 $
 *
 * -------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This file contains all of the local variable and data structure definitions.
 * It also includes all other header files necessary for the module's function.
 * It also prototypes all functions internal to that module.
 *
 * -------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * -------------------------------------------------------------------------
 */

#ifndef ISO_DELIVER_LOCALS_H
#define ISO_DELIVER_LOCALS_H

#include <hostman/host_utils.h>
#include <api/iso_deliver/exports.h>
#include <api/iso_send/exports.h>
#include <processing/iom/exports.h>

/* The last packet that was delivered to the application.  This packet is
   deleted on the subsequent call to iso_recieve(). */
PACKET_PTR last_packet;
/* The sending node of the last_packet */
USHORT     packet_sender;

int thru_counter = 0;

#endif
```

api/iso_deliver/iso_deliver.c

```
/*
 * -------------------------------------------------------------------------
 *
 * Isotach Module  : Iso-Deliver Module
 * Isotach Layer   : API
 * Isotach Version : Version 3
 *
 * -------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *        $Source: /home2/isotach/cvsroot/v3/api/send/send.c,v $
 *        $Revision: 1.6 $
 *        $Author: pnm2h $
 *        $Date: 1999/11/04 22:29:15 $
 *
 * -------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This module exports the API functions:
 *
 * iso_receive()
 *
 * See API Documentation for Details
 *
 * -------------------------------------------------------------------------
 *
 *
 * -------------------------------------------------------------------------
 */

#include <api/iso_deliver/locals.h>

/* Isotach Deliver Module Initialization */
int iso_deliver_init() {
  last_packet = (PACKET_PTR)NULL;
  iso_delivery_h = iso_delivery_t = iso_delivery_q;
  return SUCCESS;
}

/* Isotach Deliver Module DeInitialization */
int iso_deliver_deinit() {
  return SUCCESS;
}

/* API: Application calls this function to recieve pending Isotach Messages.
   Messages can be either MBM data, or Signals/Barriers.  If there is no
   message to recieve, the function returns FAILURE, otherwise the function
   returns SUCCESS and the message is placed in the iso_mbm struct */
int iso_receive(iso_mbm *data) {

  /* Delete the last packet retrieved, if there is one to be deleted.  The
     application must copy or use the data contents before another call to
     iso_recieve() or the data may be deleted. */
  if (last_packet != NULL) {
    if (packet_sender == host_node_id) {
      inc_ptr(hit_h, HIT_BUF_SIZE, hit_buf);
    }
    else {
      iso_delete_packet(packet_sender);
    }
    last_packet = NULL;
  }

  iso_poll();

  /* If there is nothing to recieve, return FAILURE */
  if (queue_empty(iso_delivery_h, iso_delivery_t)) {
    return FAILURE;
  }
```

```
  // Assert (removable) that the item at the head of iso_delivery_q is an
  // iso_pointer (pointing to an ordered net-packet), an isochron_slot
  // (indicating the number of self-packets that should be delivered next) or
  // a bs-notice.

  /* If the head message on the iso_delivery queue is a message deliver the
     message to the application from the iso_delivery_q */
  if (iso_delivery_h->subtype == ISO_MBM) {
    last_packet   = (PACKET_PTR)((ULONG)(((isopointer *)iso_delivery_h)->pointer) + offset);
    packet_sender = ((isopointer *)iso_delivery_h)->sender;

    /* Check to see that the Ordered Packet corresponding to the Isotach Packet
       has arrived.  Because of FIFO, this should not happen. */
    if (ord_receive_t[packet_sender] == last_packet) {
      //printf("FAILURE: Ordered Packet Correspoinding to this pointer has not arrived.
Aborting.\n");
      last_packet = (PACKET_PTR)NULL;
      return FAILURE;
    }

    /* Set the iso_mbm for the Application */
    data->tag             = ISO_MBM;
    data->info.msg.data   = last_packet->data;
    data->info.msg.length = ntohs(last_packet->payload_length);
    data->info.msg.sender = packet_sender;
    /* Remove the message from the iso_delivery_q */
    inc_ptr(iso_delivery_h, ISO_DELIVERY_SIZE, iso_delivery_q);
  }
  /* If the message at the head of the iso_delivery_q is not a message, it
     could be an ISO_SLOT which points to a self message.  Retrieve the
     self-message off of the hit_q. */
  else if (iso_delivery_h->subtype == ISO_SLOT) {
    last_packet   = hit_h;
    packet_sender = host_node_id;

    /* Set tje iso_mbm for the Application */
    data->tag             = ISO_MBM;
    data->info.msg.data   = last_packet->data;
    data->info.msg.length = last_packet->payload_length;
    data->info.msg.sender = packet_sender;

    /* If this is the last message pointed to by the ISO_SLOT, remove the
       ISO_SLOT from the iso_delivery_q, otherwise just decrement the ISO_SLOT
       self count */
    if (((isoslot *)iso_delivery_h)->self_count == 1) {
      inc_ptr(iso_delivery_h, ISO_DELIVERY_SIZE, iso_delivery_q);
    }
    else {
      ((isoslot *)iso_delivery_h)->self_count--;
    }
  }
  /* Finally, the message could be a BS_MARKER.  In this case, create a message
     for the Application that contains the Barrier/Signal Bits */
  else if (iso_delivery_h->subtype == BS_MARKER) {
    data->tag = BS_MARKER;
    data->info.bits = ((bsnotice *)iso_delivery_h)->bits;
  }
  else {
    printf("FAILURE: Item at head of iso_delivery_q was not valid.  Aborting.\n");
    return FAILURE;
  }

  return SUCCESS;
}
```

api/iso_barrier/exports.h

```
/*
 * -------------------------------------------------------------------------
 *
 * Isotach Module  : Iso Barrier Module Exported Functions/Variables
 * Isotach Layer   : API
 * Isotach Version : Version 3
 *
 * -------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/api/send/exports.h,v $
 *       $Revision: 1.6 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/02 01:24:00 $
 *
 * -------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This file contains all of the exported functions, variables and data
 * structures for the given module.  Other modules which need to access these
 * variables/functions should include this header file.
 *
 * -------------------------------------------------------------------------
 *
 * COMMENTS:
 * Included by the Hostman Module and the Flow Module
 *
 * -------------------------------------------------------------------------
 */

#ifndef ISO_BARRIER_EXPORTS_H
#define ISO_BARRIER_EXPORTS_H

/* This is the default barrier to be used by the try_close_net function in
   hostman.  This can be either 0 or 1, and 0 is picked arbitrarily. */
#define SHUTDOWN_BARRIER 0

/* Some types used by the mLayer.  The Application should use the types
   ISO_BARRIER_WEAK and ISO_BARRIER_STRONG which correspond to the values of
   WEAK and STRONG respectively. */
enum barrier_mode {
  WEAK = 0x0, STRONG, TICK, MAX
};



int iso_register_barrier(int channel, UCHAR bmode);
int iso_clear_barrier(int channel);
int iso_barrier(int channel, UCHAR bmode);
int iso_mLayer_barrier(int channel, UCHAR bmode);
int iso_barrier_notify(UCHAR bits);
int iso_barrier_poll();

#endif
```

api/iso_barrier/locals.h

```
/*
 * ---------------------------------------------------------------------------
 *
 * Isotach Module   : Iso Barrier Module Local Variable Header File
 * Isotach Layer    : API
 * Isotach Version : Version 3
 *
 * ---------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/api/send/locals.h,v $
 *       $Revision: 1.5 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/02 01:24:00 $
 *
 * ---------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This file contains all of the local variable and data structure definitions.
 * It also includes all other header files necessary for the module's function.
 * It also prototypes all functions internal to that module.
 *
 * ---------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * ---------------------------------------------------------------------------
 */

#ifndef ISO_BARRIER_LOCALS_H
#define ISO_BARRIER_LOCALS_H

#include <hostman/host_utils.h>
#include <niu_interface/iso_shipping/exports.h>
#include <api/iso_signal/exports.h>
#include <api/iso_deliver/exports.h>
#include <api/iso_send/exports.h>
#include <api/iso_barrier/exports.h>

/* Some Constants Pertaining to Barriers */
#define NUM_BARRIERS       2    // Number of usable barriers in the system
#define NUM_BARRIER_MODES 4     // Number of modes usable by the mLayer
#define MIN_BARRIER_COUNT 2     // Minimum number of ticks for a barrier
#define MAX_BARRIER_COUNT 37    // Max number of ticks for a barrier. See spec.

/* The different states that each barrier might be in.  See Ironman for more
   details on the states. */
enum barrier_state {
  HOLDING = 0x0, NOT_HOLDING, SEND, TRANSITION, READY
};

/* Each barrier has a record associated with it that records its state, node
   owner and which mode the barrier is in. */
typedef struct {
  UCHAR owner;
  UCHAR mode;
  UCHAR state;
} BARRIER_RECORD;

/* Diameter of the network in max number of hops */
int network_diameter;

BARRIER_RECORD barriers[NUM_BARRIERS];
UCHAR          barrier_count[NUM_BARRIER_MODES];

int enqueue_barrier(int channel);

#endif
```

api/iso_barrier/iso_barrier.c

```c
/*
 * ------------------------------------------------------------------------
 *
 * Isotach Module  : Iso Barrier Module
 * Isotach Layer   : API
 * Isotach Version : Version 3
 *
 * ------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/api/send/send.c,v $
 *       $Revision: 1.6 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:15 $
 *
 * ------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This module exports the API functions:
 *
 * iso_register_barrier()
 * iso_clear_barrier()
 * iso_barrier()
 *
 * See API Documentation for Details
 *
 * ------------------------------------------------------------------------
 *
 *
 * ------------------------------------------------------------------------
 */

#include <api/iso_barrier/locals.h>

/* Iso Barrier Module Initialization Function */
int iso_barrier_init(node_info *nst) {
  int i;
  int route_length = 0;
  network_diameter = 0;

  /* This sets the network diameter based upon the max number of hops in any
     route between two hosts */
  for (i = 0; i < number_of_hosts; i++) {
    if ((ntohl(nst[i].route) & FOURTH_BYTE) == 0) {
      route_length = 0;
    }
    else if ((ntohl(nst[i].route) & THIRD_BYTE) == 0) {
      route_length = 1;
    }
    else if ((ntohl(nst[i].route) & SECOND_BYTE) == 0) {
      route_length = 2;
    }
    else if ((ntohl(nst[i].route) & FIRST_BYTE) == 0) {
      route_length = 3;
    }
    else {
      route_length = 4;
    }

    if (route_length > network_diameter) {
      network_diameter = route_length;
    }
  }

  /* Initialize the barriers.  Barrier 0 is used only by the mLayer, and
     1 is usable by the application */
  barriers[0].owner = MLAYER;
  barriers[1].owner = UNCLAIMED;
```

```c
  for (i = 0; i < NUM_BARRIERS; i++) {
    barriers[i].mode  = MAX;
    barriers[i].state = SEND;
  }

  barriers[SHUTDOWN_BARRIER].state = HOLDING;
  barriers[SHUTDOWN_BARRIER].mode  = STRONG;

  /* Initialize each of the Barrier strengths.  WEAK and STRONG are identical
     for now, but can be made different. */
  /* NOTE: I am not sure why these are identical - Perry 3/17/2000 */
  barrier_count[WEAK]   = network_diameter + 1;
  barrier_count[STRONG] = barrier_count[WEAK];
  barrier_count[TICK]   = MIN_BARRIER_COUNT;
  barrier_count[MAX]    = MAX_BARRIER_COUNT;

  return SUCCESS;
}

/* Isotach Barrier Module Deinitialization Function */
int iso_barrier_deinit() {
  return SUCCESS;
}


/* This function enqueues a barrier for sending. */
int enqueue_barrier(int channel) {

  /* Removable Assertion that the barrier being enqueued is in a SEND state */
  if (barriers[channel].state != SEND) {
    printf("FAILURE: Barrier[%d] State is not SEND.  Aborting.\n", channel);
    shutdown(1);
  }

  /* Can only enqueue a barrier, if there is room left in the iso_send_buf */
  if (queue_full(iso_send_h, iso_send_t, ISO_SEND_BUF_SIZE)) {
    printf("FAILURE: Iso Send Buffer is full.\n");
    return FAILURE;
  }

  /* Go ahead and put the barrier on the buffer, and indicate that it is ok
     to send. */
  iso_send_t->prefix = BS_MARKER_MASK;
  iso_send_t->prefix |= (barrier_count[barriers[channel].mode] << 16);
  iso_send_t->prefix |= (1 << (channel + 8));

  inc_ptr(iso_send_t, ISO_SEND_BUF_SIZE, iso_send_buf);
  barriers[channel].state = NOT_HOLDING;

  return SUCCESS;
}

/* This function allows the application to register a barrier for
   use.  The application can only register barrier channel 1, as 0 is reserver
   for the mLayer. If the barrier is already registered, it must be cleared
   first using iso_clear_barrier() */
int iso_register_barrier(int channel, UCHAR bmode) {
  if ((channel < 0) || (channel >= NUM_BARRIERS)) {
    printf("FAILURE: Barrier Channel out of range.\n");
    return FAILURE;
  }

  if (barriers[channel].owner != UNCLAIMED) {
    printf("FAILURE: Barrier is already claimed.\n");
    return FAILURE;
  }

  if ((bmode != STRONG) && (bmode != WEAK)) {
    printf("FAILURE: Barrier Mode is neither Strong nor Weak.\n");
    return FAILURE;
  }

  /* Only the application can register barriers, so set owner to APPLICATION */
  barriers[channel].owner = APPLICATION;
```

```
  /* Set mode the the specified mode */
  barriers[channel].mode  = bmode;

  /* Removable assertion that the barrier state is either NOT_HOLDING or SEND*/
  if ((barriers[channel].state != NOT_HOLDING) &&
      (barriers[channel].state != SEND)) {
    printf("FAILURE: Barrier[%d] State is not SEND.  Aborting.\n", channel);
    shutdown(1);
  }

  /* Change barrier states.  See Ironman for State Diagram. */
  if (barriers[channel].state == NOT_HOLDING) {
    barriers[channel].state = TRANSITION;
  }
  else {
    barriers[channel].state = HOLDING;
  }

  return SUCCESS;
}

/* This function allows the application to clear a registered barrier for
   re-registration.  It is a NOP when used on an unregistered barrier
   channel. */
int iso_clear_barrier(int channel) {

  if ((channel < 0) || (channel >= NUM_BARRIERS)) {
    printf("FAILURE: Barrier Channel out of range.\n");
    return FAILURE;
  }

  if (barriers[channel].owner != APPLICATION) {
    return FAILURE;
  }

  /* Clear the owner, and reset the mode to MAX */
  barriers[channel].owner = UNCLAIMED;
  barriers[channel].mode  = MAX;

  /* Change the barrier state.  See State Diagram in Ironman */
  if ((barriers[channel].state == TRANSITION) ||
      (barriers[channel].state == READY)) {
    barriers[channel].state = NOT_HOLDING;
  }
  else if (barriers[channel].state == HOLDING) {
    barriers[channel].state = SEND;
  }

  if (barriers[channel].state == SEND) {
    enqueue_barrier(channel);
  }

  return SUCCESS;
}

/* Initiate an Isotach Barrier on the specified channel and with the specified
   mode.  The mode specified here must match the mode specified when the
   channel was registered. This function is only called by the Application */
int iso_barrier(int channel, UCHAR bmode) {
  if ((channel < 0) || (channel >= NUM_BARRIERS)) {
    printf("FAILURE: Barrier Channel out of range.\n");
    return FAILURE;
  }

  if (barriers[channel].owner != APPLICATION) {
    printf("FAILURE: Barrier is not registered to Application.\n");
    return FAILURE;
  }

  if (bmode != barriers[channel].mode) {
    printf("FAILURE: Barrier Mode does not match registered barrier mode\n");
    return FAILURE;
  }
```

```
  if (mid_net_isochron == TRUE) {
    printf("FAILURE: Cannot send a Barrier while in the middle of an Isochron.\n");
    return FAILURE;
  }

  /* More barrier state changes... See Ironman */
  if (barriers[channel].state == HOLDING) {
    /* If the barrier is in the Holding State, then send out the barrier by
       calling enqueue_barrier() and setting state to SEND */
    barriers[channel].state = SEND;
    enqueue_barrier(channel);
  }
  else if (barriers[channel].state == TRANSITION) {
    barriers[channel].state = READY;
  }
  else {
    return FAILURE;
  }

  return SUCCESS;
}


/* Initiate an Isotach Barrier on the specified channel and with the specified
   mode.  The mode specified here must match the mode specified when the
   channel was registered. This function is only called by the mLayer */
int iso_mLayer_barrier(int channel, UCHAR bmode) {

  if ((channel < 0) || (channel >= NUM_BARRIERS)) {
    printf("FAILURE: Barrier Channel out of range.\n");
    return FAILURE;
  }

  if (barriers[channel].owner != MLAYER) {
    printf("FAILURE: Barrier is not registered to mLayer.\n");
    return FAILURE;
  }

  if (bmode != barriers[channel].mode) {
    printf("FAILURE: Barrier Mode does not match registered barrier mode\n");
    return FAILURE;
  }

  if (mid_net_isochron == TRUE) {
    printf("FAILURE: Cannot send a Barrier while in the middle of an Isochron.\n");
    return FAILURE;
  }

  /* More barrier state changes... See Ironman */
  if (barriers[channel].state == HOLDING) {
    /* If the barrier is in the Holding State, then send out the barrier by
       calling enqueue_barrier() and setting state to SEND */
    barriers[channel].state = SEND;
    enqueue_barrier(channel);
  }
  else if (barriers[channel].state == TRANSITION) {
    barriers[channel].state = READY;
  }
  else {
    return FAILURE;
  }

  return SUCCESS;
}

/* Isotach Barrier Poll Function */
int iso_barrier_poll() {
  int i;

  /* Check each barrier channel.  If any barrier channel has state SEND, then
     try to enqueue the barrier for sending. */
  for (i = 0; i < NUM_BARRIERS; i++) {
    if (barriers[i].state == SEND) {
      enqueue_barrier(i);
```

```
    }
  }

  return SUCCESS;
}

/* Called by the IOM to notify the Isotach Barrier Module that barrier bits
   have arrived in a message. */
int iso_barrier_notify(UCHAR bits) {
  UCHAR app_bits = 0x0;
  int i;

  /* Check each of the barrier channels */
  for (i = 0; i < NUM_BARRIERS; i++) {
    if (bits & BITS[i]) {
      // Removable assertion that the barrier state is neither SEND nor HOLDING
      if ((barriers[i].state == SEND) ||
           (barriers[i].state == HOLDING)) {
          printf("FAILURE: Barrier[%d] State is SEND or HOLDING.  Aborting.\n", i);
          shutdown(1);
      }

      /* Barrier state transitions... See Ironman */
      if (barriers[i].state == TRANSITION) {
          barriers[i].state = HOLDING;
      }
      else if ((barriers[i].state == NOT_HOLDING)
                && (barriers[i].owner == APPLICATION)) {
          app_bits |= BITS[i];
          barriers[i].state = HOLDING;
      }
      else if ((barriers[i].state == READY) ||
                ((barriers[i].state == NOT_HOLDING) &&
                  (barriers[i].owner != APPLICATION))) {
          barriers[i].state = SEND;
          enqueue_barrier(i);
      }
    }
  }

  return app_bits;
}
```

api/iso_signal/exports.h

```
/*
 * -------------------------------------------------------------------------
 *
 * Isotach Module  : Signal Module Exported Functions/Variables
 * Isotach Layer   : API
 * Isotach Version : Version 3
 *
 * -------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/api/send/exports.h,v $
 *       $Revision: 1.6 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/02 01:24:00 $
 *
 * -------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This file contains all of the exported functions, variables and data
 * structures for the given module.  Other modules which need to access these
 * variables/functions should include this header file.
 *
 * -------------------------------------------------------------------------
 *
 * COMMENTS:
 * Included by the Hostman Module and the Flow Module
 *
 * -------------------------------------------------------------------------
 */

#ifndef ISO_SIGNAL_EXPORTS_H
#define ISO_SIGNAL_EXPORTS_H

/* This defines which bit (0-5) of the BS Marker should be interpreted as the
   reset signal bit.  This signal is owned by the mLayer, and should not be
   changed.  Used by both iso_signal and hostman */
#define RESET_SIGNAL 5

/* The number of resets that have happened yet.  Initialized to zero and
   incremented for every reset signal that is received. */
int reset_count;

int iso_register_signal(int channel);
int iso_clear_signal(int channel);
int iso_send_signal(int channel);
int iso_send_mLayer_signal(int channel);
int iso_signal_poll();
UCHAR iso_signal_notify(UCHAR bits);
int iso_signal_init();
int iso_signal_deinit();

#endif
```

api/iso_signal/locals.h

```
/*
 * ------------------------------------------------------------------------
 *
 * Isotach Module  : Signal Module Local Variable Header File
 * Isotach Layer   : API
 * Isotach Version : Version 3
 *
 * ------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/api/send/locals.h,v $
 *       $Revision: 1.5 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/02 01:24:00 $
 *
 * ------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This file contains all of the local variable and data structure definitions.
 * It also includes all other header files necessary for the module's function.
 * It also prototypes all functions internal to that module.
 *
 * ------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * ------------------------------------------------------------------------
 */

#ifndef ISO_SIGNAL_LOCALS_H
#define ISO_SIGNAL_LOCALS_H

#include <hostman/host_utils.h>
#include <niu_interface/iso_shipping/exports.h>
#include <api/iso_signal/exports.h>
#include <api/iso_deliver/exports.h>
#include <api/iso_send/exports.h>

/* The number of signals available to the MLayer and Application */
#define NUM_SIGNALS 6

/* These are the 'bits' that need to be sent out as part of the initial reset
   signal sent out by host number 0.  This is a combination of the reset signal
   and both barrier channels.  The barrier channels need to go out with the
   reset to prime the system (by giving the Token Manager initial barrier
   credits) for future barriers. */
#define INITIAL_RESET_SIGNAL 0x83

/* The initial ownership of each of the signal channels.  The Application
   can use signals 1-4, by calling the iso_register_signal function.  Signals
   0 and 5 are reserved for use by the MLayer. */
UCHAR signals[] = {MLAYER, UNCLAIMED, UNCLAIMED, UNCLAIMED, UNCLAIMED, MLAYER};

/* Each call to iso_send_signal sets a bit in the signal accumulator.  Normally
   one signal is sent on a given outgoing signal packet, but if for some reason
   that packet cannot be sent, the signal accumulator accumulates signals and
   sends them all out at once. */
UCHAR signal_accumulator;

int enqueue_signals();

#endif
```

api/iso_signal/iso_signal.c

```c
/*
 * -------------------------------------------------------------------------
 *
 * Isotach Module  : Signal Module
 * Isotach Layer   : API
 * Isotach Version : Version 3
 *
 * -------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/api/send/send.c,v $
 *       $Revision: 1.6 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:15 $
 *
 * -------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This module exports the API functions:
 *
 * iso_register_signal()
 * iso_clear_signal()
 * iso_send_signal()
 *
 * See API Documentation for Details
 *
 * -------------------------------------------------------------------------
 *
 *
 * -------------------------------------------------------------------------
 */

#include <api/iso_signal/locals.h>

/* Module Initialization Function */
int iso_signal_init() {
  signal_accumulator = 0;
  reset_count        = 0;

  return SUCCESS;
}

/* Module De-Initialization Function */
int iso_signal_deinit() {
  return SUCCESS;
}

/* This function enqueues the current signal_accumulator onto a packet on
   iso_send_t to be sent out on the network. */
int enqueue_signals() {
  /* If the queue is full, fail for now and try again later... */
  if (queue_full(iso_send_h, iso_send_t, ISO_SEND_BUF_SIZE)) {
    printf("FAILURE: iso_send_buf full.  Cannot send signal. \n");
    return FAILURE;
  }

  /* Create the packet in iso_send_t */
  iso_send_t->prefix = BS_MARKER_MASK;
  /* Or in the signals... */
  iso_send_t->prefix |= (signal_accumulator << 8);
  /* Reset the accumulator */
  signal_accumulator = 0;

  /* Clear the packet for flow control to handle... */
  inc_ptr(iso_send_t, ISO_SEND_BUF_SIZE, iso_send_buf);

  return SUCCESS;
}
```

```
/* This registers a signal channel to the application. The channel must not
   be registered already. */
int iso_register_signal(int channel) {
  if ((channel < 0) || (channel >= NUM_SIGNALS)) {
    printf("FAILURE: Signal requested is out of range.  Aborting.\n");
    return FAILURE;
  }

  if (signals[channel] != UNCLAIMED) {
    printf("FAILURE: Attempted to register an already claimed signal.\n");
    return FAILURE;
  }

  signals[channel] = APPLICATION;

  return SUCCESS;
}

/* This unregisters the signal at the specified channel.  The signal must be
   registered to the application in order for this function to work. */
int iso_clear_signal(int channel) {
  if ((channel < 0) || (channel >= NUM_SIGNALS)) {
    printf("FAILURE: Signal requested is out of range.  Aborting.\n");
    return FAILURE;
  }

  if (signals[channel] == APPLICATION)
    signals[channel] = UNCLAIMED;

  return SUCCESS;
}

/* This sends out a signal on the specified channel.  The signal channel must
   be registered to the application in order for this to work. */
int iso_send_signal(int channel) {
  if ((channel < 0) || (channel >= NUM_SIGNALS)) {
    printf("FAILURE: Signal requested is out of range.  Aborting.\n");
    return FAILURE;
  }

  if ((signals[channel] != APPLICATION) || (mid_net_isochron == TRUE)) {
    printf("FAILURE: Signal is not registered to application, or in middle of isochron.\n");
    return FAILURE;
  }

  /* Add this signal to the signal accumulator.  Normally, signals are sent
     out individually.  (i.e. each addition to the signal accumulator results
     in a packet sent out with that signal bit set).  However, if the iso_send
     buffer is full, the signal send may be delayed and the accumulator may
     gather more than one signal.  */

  /* ********************************************************************** */
  /* NOTE: What happens if two signals are sent out in a row with the same  */
  /*       channel?  Potentially, the first signal ends up on the           */
  /*       accululator w/o being sent, because of a full iso_send_buf.      */
  /*       The second signal would be or'd onto the signal accumulator, but */
  /*       the end result is that one signal is sent out on channel x, not  */
  /*       two, which is what the application requested. This needs to be   */
  /*       looked into.                                                     */
  /*       -Perry Myers (03/28/2000)                                        */
  /* ********************************************************************** */

  signal_accumulator |= (1 << (channel + 2));

  if (enqueue_signals() == FAILURE) {
    return FAILURE;
  }

  return SUCCESS;
}

/* This is the function that the mLayer calls to send out an mLayer signal.
   It is used by the mLayer currently to send out a reset signal at system
   initialization */
```

```
int iso_send_mLayer_signal(int channel) {
  if ((channel < 0) || (channel >= NUM_SIGNALS)) {
    printf("FAILURE: Signal requested is out of range.  Aborting.\n");
    return FAILURE;
  }

  if ((signals[channel] != MLAYER) || (mid_net_isochron == TRUE)) {
    printf("FAILURE: Signal is not registered to mLayer, or in middle of isochron.\n");
    return FAILURE;
  }

  if ((reset_count == 0) && (channel == RESET_SIGNAL)) {
    /* If this is the first reset signal sent, then send the
       INITIAL_RESET_SIGNAL, which includes the barrier bits and keep
       trying to send it until it goes out... */
    signal_accumulator = INITIAL_RESET_SIGNAL;
    while (enqueue_signals() == FAILURE);
  }
  else {
    /* If this is not the first reset signal (i.e. all other mLayer signals)
       then add the signal to the accumulator and attempt to send */
    signal_accumulator |= (1 << (channel + 2));
    if (enqueue_signals() == FAILURE) {
      return FAILURE;
    }
  }

  return SUCCESS;
}

int iso_signal_poll() {
  /* If there are signals in the accumulator, attempt to send. */
  if (signal_accumulator != 0) {
    enqueue_signals();
  }

  return SUCCESS;
}

/* IOM calls this when it recieves a packet with signal bits sent.
   It handles the processing of reset signals.  For the first reset signal
   (i.e. the initial reset signal) it simply increments the counter, and for
   all other resets it shuts the system down. All other signal bits are
   delivered to the application, by passing app_bits back to the IOM */
UCHAR iso_signal_notify(UCHAR bits) {
  UCHAR app_bits = 0x00;
  int i;

  /* Shift the bits right so that the barrier bits are shifted out... */
  bits = bits >> 2;

  /* If the RESET bit is set... */
  if (bits & BITS[RESET_SIGNAL]) {
    reset_count++;

    /* If this is not the initial reset signal, shut down the system */
    if (reset_count > 1) {
      printf("Received a reset signal.  Shutting Down System...\n");
      fprintf (stderr, "Loading the LANai with ./lcp\n");
      lanai_load_and_reset (0, "./lcp" , 0, 0, 0, callback);
      shutdown(1);
    }
  }

  /* For all of the other application signals set, place the bit into app_bits
   */
  for (i = 0; i < NUM_SIGNALS; i++) {
    if (bits & BITS[i]) {
      if (signals[i] == APPLICATION) {
          app_bits |= BITS[i];
      }
    }
  }
```

```
  /* Shift back 2 to the left, to leave room for the barrier bits. */
  app_bits = app_bits << 2;

  return app_bits;
}
```

api/iso_retrieve/exports.h

```
/*
 * -------------------------------------------------------------------------
 *
 * Isotach Module  : Iso-Retrieve Module Exported Functions/Variables
 * Isotach Layer   : API
 * Isotach Version : Version 3
 *
 * -------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/api/send/exports.h,v $
 *       $Revision: 1.6 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/02 01:24:00 $
 *
 * -------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This file contains all of the exported functions, variables and data
 * structures for the given module.  Other modules which need to access these
 * variables/functions should include this header file.
 *
 * -------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 *
 * -------------------------------------------------------------------------
 */

#ifndef ISO_RETRIEVE_EXPORTS_H
#define ISO_RETRIEVE_EXPORTS_H

//iso_getread()
//report_value()
//handle_read_response()
//getread_init()
//getread_deinit()
//getread_poll()          // callme function called by hostMan poll()

#endif
```

api/iso_retrieve/locals.h

```
/*
 * ------------------------------------------------------------------------
 *
 * Isotach Module  : Iso-Retreive Module Local Variable Header File
 * Isotach Layer   : API
 * Isotach Version : Version 3
 *
 * ------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/api/send/locals.h,v $
 *       $Revision: 1.5 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/02 01:24:00 $
 *
 * ------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This file contains all of the local variable and data structure definitions.
 * It also includes all other header files necessary for the module's function.
 * It also prototypes all functions internal to that module.
 *
 * ------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * ------------------------------------------------------------------------
 */

#ifndef ISO_RETRIEVE_LOCALS_H
#define ISO_RETRIEVE_LOCALS_H

#include <hostman/host_utils.h>
#include <niu_interface/receive/exports.h>
//#include <api/iso_retrieve/exports.h>
//#include <processing/shmem/exports.h>

#endif
```

api/iso_retrieve/iso_retrieve.c

```
/*
 * -------------------------------------------------------------------------
 *
 * Isotach Module  : Iso-Retrieve Module
 * Isotach Layer   : API
 * Isotach Version : Version 3
 *
 * -------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/api/send/send.c,v $
 *       $Revision: 1.6 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:15 $
 *
 * -------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This module exports the API functions:
 *
 *
 * See API Documentation for Details
 *
 * -------------------------------------------------------------------------
 *
 * COMMENTS: This module is not yet implemented yet.
 *
 * -------------------------------------------------------------------------
 */

#include <api/iso_retrieve/locals.h>
```

processing/flow/exports.h

```c
/*
 * ---------------------------------------------------------------------------
 *
 * Isotach Module  : Flow Module Exported Functions/Variables
 * Isotach Layer   : Processing
 * Isotach Version : Version 3
 *
 * ---------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/flow_control/flow/exports.h,v $
 *       $Revision: 1.6 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:15 $
 *
 * ---------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This file contains all of the exported functions, variables and data
 * structures for the given module.  Other modules which need to access these
 * variables/functions should include this header file.
 *
 * ---------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * Included by the Hostman, Shipping, Deliver, and Receive Modules
 *
 * ---------------------------------------------------------------------------
 */

#ifndef FLOW_EXPORTS_H
#define FLOW_EXPORTS_H

int flow_init(node_info *nst);
int flow_deinit();
int update_credit(int node_id, PACKET_PTR new_head);
int flow_poll();
int send_credit_packet(ULONG dest);
int delete_packet(ULONG sender, PACKET_PTR packet);

/* The size of each NONISO queue in pinned memory.  This size is in number of
   PACKETS */
ULONG REMOTE_NONISO_SIZE;

#endif
```

processing/flow/locals.h

```
/*
 * -------------------------------------------------------------------------
 *
 * Isotach Module  : Flow Module Local Variable Header File
 * Isotach Layer   : Processing
 * Isotach Version : Version 3
 *
 * -------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/flow_control/flow/locals.h,v $
 *       $Revision: 1.5 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:15 $
 *
 * -------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This file contains all of the local variable and data structure definitions.
 * It also includes all other header files necessary for the module's function.
 * It also prototypes all functions internal to that module.
 *
 * -------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * -------------------------------------------------------------------------
 */

#ifndef FLOW_LOCALS_H
#define FLOW_LOCALS_H

#include <hostman/host_utils.h>
#include <api/send/exports.h>
#include <processing/flow/exports.h>
#include <niu_interface/shipping/exports.h>
#include <niu_interface/receive/exports.h>

#define MAX_SEND_ATTEMPTS 500000

int credit_packet_threshold;

/* This is an array of pointers to the beginning of queues.  There is one for
   each other host in the network.  The remote_t pointer for a host is
   incremented each time a new packet is send out to that host.  The remote_q
   variable allows the base of the queue to be stored, so that queue wrap
   around works */
PACKET_PTR *remote_q;
PACKET_PTR *remote_h;

/* This is an array of tail pointers.  There is one for each other host in the
   network.  When new packets are constructed in the send module, the value of
   remote_t[target node] is written into the address feild.  This allows for
   specific sender based flow control.
   For remote_t[index], this tail pointer corresponds to node index's i-th
   receive buffer, where i is this host_node_id */
PACKET_PTR *remote_t;

/* These are the head and base pointers for the NONISO receive buffers.  These
   pointers refer to the local receive buffers that are used by other hosts in
   the network */
PACKET_PTR *receive_h;
PACKET_PTR *receive_q;
PACKET_PTR *receive_last_h;
PACKET_PTR *credit_packet_base;

PACKET_PTR credit_packet_slot;

/*
```

```
  This is a tunable variable that indicates at what point a sender should
  send an explicit credit packet.
*/
float CREDIT_PACKET_THRESHOLD_PERCENTAGE = 0.90;

#endif
```

processing/flow/flow.c

```c
/*
 * -------------------------------------------------------------------------
 *
 * Isotach Module  : Flow Module
 * Isotach Layer   : Processing
 * Isotach Version : Version 3
 *
 * -------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/flow_control/flow/flow.c,v $
 *       $Revision: 1.3 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:15 $
 *
 * -------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This module controls the flow of packets from non-iso send to the network.
 * The send_flow_ptr points to the next packet to be examined on the send_buf,
 * by the flow module.  As flow clears a packet for transmission by shipping
 * to the niu_send_buf, it increments send_flow_ptr.  Flow uses sender
 * specific flow control.  That is, each host maintains how many send credits
 * it has to send to a particular host.
 *
 * -------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * -------------------------------------------------------------------------
 */

#include <processing/flow/locals.h>


/*
   called by deliver once we know that the previous packet has been
   processed by the application.  Thus, we can now free up one
   send credit
*/
int delete_packet(ULONG sender, PACKET_PTR packet) {
  // After assertion below is removed, remove packet argument to this function

  // Free up another slot
  inc_ptr(receive_h[sender], REMOTE_NONISO_SIZE, receive_q[sender]);

  // Assertion: packet points to the packet at head of sender's receive
  //            buffer.  This should be removed later.
  //if ((ULONG)packet != (ULONG)receive_h[sender] + offset) {
  //  printf("ERROR: Failed assertion packet != receive_h[sender] in delete_packet.\n");
  //  exit(1);
  //}

  /*
     If we've freed up quite a few credits and not sent any credit
     information to the sender, send an explicit credit packet.
  */
  if (queue_size(receive_last_h[sender], receive_h[sender], REMOTE_NONISO_SIZE)
      > credit_packet_threshold) {
    send_credit_packet(sender);
  }

  return SUCCESS;
}


/* called by delete packet to send a credit packet */
int send_credit_packet(ULONG dest) {
  int i = 0;
```

```c
  /* The target is written into the route, so that shipping will know which
     entry off of routes[] to use */
  credit_packet_slot->route       = dest;
  credit_packet_slot->credit_info = (ULONG)receive_h[dest];
  credit_packet_slot->address     = (ULONG)htonl((ULONG)credit_packet_base[dest]);

  /*
     we use the shipping module's ship_packet function to bypass flow
     control to send the credit packet.  The only way this function can
     fail is if the Lanai's send queue is full.  Since every packet on
     that queue is able to be shipped, the queue will have a free slot
     shortly.
  */
  while (ship_packet(credit_packet_slot) == FAILURE) {
    ;
  }

  //printf("Sending Credit Packet to %d: %d slots freed\n", dest,
queue_size(receive_last_h[dest], receive_h[dest], REMOTE_NONISO_SIZE));

  // remember the credit information that we sent to the sender
  receive_last_h[dest] = receive_h[dest];
  return SUCCESS;
}

// called by Hostman once the application has finished.
int flow_deinit() {
  free(remote_q);
  free(remote_h);
  free(remote_t);
  free(receive_q);
  free(receive_h);
  free(receive_last_h);
  free(credit_packet_base);
  free(credit_packet_slot);
  return SUCCESS;
}

// called by Hostman to initialize all of the data structures contained in
// this module
int flow_init(node_info *nst) {
  int i;

  /* The arrays are initialized, and set the the DMA base of each of
     this host's remote noniso queues on every other host in the network */
  remote_q       = (PACKET_PTR *)malloc(number_of_hosts * sizeof(PACKET_PTR));
  remote_h       = (PACKET_PTR *)malloc(number_of_hosts * sizeof(PACKET_PTR));
  remote_t       = (PACKET_PTR *)malloc(number_of_hosts * sizeof(PACKET_PTR));
  receive_q      = (PACKET_PTR *)malloc(number_of_hosts * sizeof(PACKET_PTR));
  receive_h      = (PACKET_PTR *)malloc(number_of_hosts * sizeof(PACKET_PTR));
  receive_last_h = (PACKET_PTR *)malloc(number_of_hosts * sizeof(PACKET_PTR));
  credit_packet_base = (PACKET_PTR *)malloc(number_of_hosts * sizeof(PACKET_PTR));
  credit_packet_slot = (PACKET_PTR)malloc(sizeof(PACKET));

  for (i = 0; i < number_of_hosts; i++) {
    remote_q[i]           = nst[i].remote_noniso.base + CONTROL_SLOTS;
    remote_h[i]           = nst[i].remote_noniso.base + CONTROL_SLOTS;
    remote_t[i]           = nst[i].remote_noniso.base + CONTROL_SLOTS;
    receive_q[i]          = nst[i].local_noniso.base + CONTROL_SLOTS;
    receive_h[i]          = nst[i].local_noniso.base + CONTROL_SLOTS;
    receive_last_h[i]     = nst[i].local_noniso.base + CONTROL_SLOTS;
    credit_packet_base[i] = nst[i].remote_noniso.base;
  }


  // calculate how many noniso credits we have at each host
  REMOTE_NONISO_SIZE = (ULONG)(nst[host_node_id].remote_noniso.size - (CONTROL_SLOTS+1));

  // calculate at what point we need to send an explicit credit packet
  credit_packet_threshold = CREDIT_PACKET_THRESHOLD_PERCENTAGE *
    REMOTE_NONISO_SIZE;

  // fill in the predetermined fields of the credit packet
```

```
  credit_packet_slot->type           = htons(NONISO);
  credit_packet_slot->subtype        = CREDIT;
  credit_packet_slot->sender         = host_node_id;
  credit_packet_slot->payload_length = 0;

  return SUCCESS;
}


// called by receive when we receive a credit packet
int update_credit(int node_id, PACKET_PTR new_head) {
  remote_h[node_id] = new_head;
  return SUCCESS;
}


// polling function called by the main poll function in hostman
int flow_poll() {
  static int send_attempts = 0;
  static int last_send_attempts = 0;
  int out_of_credits = FALSE;
  int receiver;
  PACKET_PTR temp_address;
  PACKET credit_request;
  PACKET_PTR request_address;


  // while there are packets waiting to be cleared and we are not out
  // of credits on any of the receivers
  while ((!queue_empty(send_t, send_flow_ptr)) && (!out_of_credits)) {
    receiver = send_flow_ptr->route;


    // If we are out of credits at the destination
    if (queue_full(remote_h[receiver],remote_t[receiver],REMOTE_NONISO_SIZE)) {
      out_of_credits = TRUE;

      // wait a while before sending out a credit request.
      // credit request packets are used to prevent a particular
      // type of deadlock which Perry will fill in here
      if (++send_attempts == MAX_SEND_ATTEMPTS) {
          //printf("Sending out a Credit Request to host %d\n", receiver);
          credit_request.type = htons(NONISO);
          credit_request.subtype = REQUEST_CREDIT;
          credit_request.pad1 = 0;
          credit_request.sender = host_node_id;
          credit_request.payload_length = 0;
          credit_request.pad2 = 0;
          credit_request.credit_info = NULL_CREDIT;
          request_address = credit_packet_base[receiver] + 1;
          credit_request.address = (ULONG)htonl((ULONG)request_address);
          credit_request.route = receiver;
          ship_packet(&credit_request);
          last_send_attempts = send_attempts;
          send_attempts = 0;
      }
    }

    // otherwise we can pass the packet on to shipping
    else {
      last_send_attempts = send_attempts;
      send_attempts = 0;

      // check to see if there is credit information to piggy back
      // on the packet

      // if nothing has changed...
      if (receive_h[receiver] == receive_last_h[receiver]) {
          send_flow_ptr->credit_info = NULL_CREDIT;
      }
      // otherwise
      else {
          receive_last_h[receiver] = receive_h[receiver];
          send_flow_ptr->credit_info = (ULONG)receive_last_h[receiver];
```

```
      }

      /*
          Now calculate the address where this packet will be placed on
          the receiver.  The tail pointer into the target's non-iso queue,
          is the location  where this packet should be placed.  This
          address is written into the address field.
      */
      send_flow_ptr->address = (ULONG)htonl((ULONG)remote_t[receiver]);
      /* The tail pointer (remote_t[target]) is incremented, for the next call
       * to send with this target.  The tail wraps around to the beginning of
       * the queue, when necessary
       */
      inc_ptr(remote_t[receiver], REMOTE_NONISO_SIZE, remote_q[receiver]);

      // now clear the packet for shipping
      inc_ptr(send_flow_ptr, SEND_BUF_SIZE, send_buf);

    }
  }


  return SUCCESS;
}
```

processing/iso_flow/exports.h

```
/*
 * -------------------------------------------------------------------------
 *
 * Isotach Module  : Isotach Flow Module Exported Functions/Variables
 * Isotach Layer   : Processing
 * Isotach Version : Version 3
 *
 * -------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/flow_control/flow/exports.h,v $
 *       $Revision: 1.6 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:15 $
 *
 * -------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This file contains all of the exported functions, variables and data
 * structures for the given module.  Other modules which need to access these
 * variables/functions should include this header file.
 *
 * -------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * Included by the Hostman, Iso_Shipping, Iso_Deliver, and Receive Modules
 *
 * -------------------------------------------------------------------------
 */


#ifndef ISO_FLOW_EXPORTS_H
#define ISO_FLOW_EXPORTS_H

int iso_flow_init(node_info *nst);
int iso_flow_deinit();
int iso_update_credit(int node_id, ULONG credit_info);
int iso_delete_packet(int node_id);
int iso_flow_poll();


// number of ordered packets that a host can store in pinned memory
ULONG REMOTE_ORD_SIZE;

#endif
```

processing/iso_flow/locals.h

```c
/*
 * -------------------------------------------------------------------------
 *
 * Isotach Module  : Isotach Flow Module Local Variable Header File
 * Isotach Layer   : Processing
 * Isotach Version : Version 3
 *
 * -------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/flow_control/flow/locals.h,v $
 *       $Revision: 1.5 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:15 $
 *
 * -------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This file contains all of the local variable and data structure definitions.
 * It also includes all other header files necessary for the module's function.
 * It also prototypes all functions internal to that module.
 *
 * -------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * -------------------------------------------------------------------------
 */


#ifndef ISO_FLOW_LOCALS_H
#define ISO_FLOW_LOCALS_H

#include <hostman/host_utils.h>
#include <niu_interface/shipping/exports.h>
#include <api/iso_send/exports.h>
#include <processing/iso_flow/exports.h>
#include <processing/iom/exports.h>
#include <niu_interface/iso_shipping/exports.h>
#include <niu_interface/iso_receive/exports.h>


int iso_send_credit_packet(ULONG dest);

/*
   array of counters for the isotach credits
   used by the local node at remote nodes
*/
ULONG *my_credits_used;

/*
   array of counters for isotach credits restored
   to the local node at remote nodes
*/
ULONG *my_credits_freed;

/*
   the total number of isotach credits the local node has
   at each of the remote nodes
*/
ULONG *my_credits;

/* the number of isotach credits restored to remote nodes */
ULONG *your_credits_freed;

/*
   the number of isotach credits allocated on the local node
   for each of the remote nodes.
```

```c
*/
ULONG *your_credits;

/* the last credit information sent to each node */
ULONG *your_credits_freed_last;


/* the address in pinned memory where isotach credit packets are sent */
PACKET_PTR *iso_credit_packet_base;


/* the point at which a receiver needs to send explicit credit packets */
ULONG iso_credit_packet_threshold;
float ISO_CREDIT_PACKET_THRESHOLD_PERCENTAGE = 0.50;

/* pointer to the isotach credit packet that is sent out */
PACKET_PTR iso_credit_packet_slot;

/* starting addresses of ordered queues on the remote hosts */
PACKET_PTR *ord_remote_q;

/*
   tail pointers into those queues a.k.a. the address to which the ordered
   packet needs to be dma'd by the receiver.
*/
PACKET_PTR *ord_remote_t;

#endif
```

processing/iso_flow/iso_flow.c

```c
/*
 * -------------------------------------------------------------------------
 *
 * Isotach Module   : Isotach Flow Module
 * Isotach Layer    : Processing
 * Isotach Version : Version 3
 *
 * -------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *        $Source: /home2/isotach/cvsroot/v3/flow_control/flow/flow.c,v $
 *        $Revision: 1.3 $
 *        $Author: pnm2h $
 *        $Date: 1999/11/04 22:29:15 $
 *
 * -------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This module controls the flow of packets from non-iso send to the network.
 * The send_flow_ptr points to the next packet to be examined on the send_buf,
 * by the flow module.  As flow clears a packet for transmission by shipping
 * to the niu_send_buf, it increments send_flow_ptr.  Flow uses sender
 * specific flow control.  That is, each host maintains how many send credits
 * it has to send to a particular host.
 *
 * -------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * -------------------------------------------------------------------------
 */


#include <processing/iso_flow/locals.h>


// Called by hostman to initialize all local data structures
int iso_flow_init(node_info *nst) {

  int i;

  // all of these arrays need to have num_hosts slots
  my_credits_used         = (ULONG *)malloc(number_of_hosts * sizeof(ULONG));
  my_credits_freed        = (ULONG *)malloc(number_of_hosts * sizeof(ULONG));
  my_credits              = (ULONG *)malloc(number_of_hosts * sizeof(ULONG));
  your_credits_freed      = (ULONG *)malloc(number_of_hosts * sizeof(ULONG));
  your_credits            = (ULONG *)malloc(number_of_hosts * sizeof(ULONG));
  your_credits_freed_last = (ULONG *)malloc(number_of_hosts * sizeof(ULONG));
  iso_credit_packet_base  = (PACKET_PTR *)malloc(number_of_hosts *
                                                  sizeof(PACKET_PTR));

  ord_remote_q = (PACKET_PTR *)malloc(number_of_hosts * sizeof(PACKET_PTR));
  ord_remote_t = (PACKET_PTR *)malloc(number_of_hosts * sizeof(PACKET_PTR));

  // an isotach credit packet
  iso_credit_packet_slot = (PACKET_PTR)malloc(sizeof(PACKET));

  // the number of ordered packets a remote host can store
  REMOTE_ORD_SIZE = nst[host_node_id].remote_iso.size;

  // point at which to send out an explicit isotach credit packet
  iso_credit_packet_threshold = REMOTE_ORD_SIZE *
    ISO_CREDIT_PACKET_THRESHOLD_PERCENTAGE;


  // initialize arrays with information retrieved during synchronization
  for (i = 0; i < number_of_hosts; i++) {
    my_credits_used[i]     = 0;
    my_credits_freed[i]    = 0;
```

```
    my_credits[i]         = REMOTE_ORD_SIZE;
    your_credits_freed[i] = 0;
    your_credits[i]       = REMOTE_ORD_SIZE;

    iso_credit_packet_base[i] = nst[i].remote_noniso.base + 3;
    ord_remote_q[i]           = nst[i].remote_iso.base;
    ord_remote_t[i]           = nst[i].remote_iso.base;
  }


  // fill in pre-determined fields of the isotach credit packet
  iso_credit_packet_slot->type           = htons(NONISO);
  iso_credit_packet_slot->subtype        = ISO_CREDIT;
  iso_credit_packet_slot->sender         = host_node_id;
  iso_credit_packet_slot->payload_length = 0;


  return SUCCESS;
}


// called by hostman when the application closes the messaging layer
int iso_flow_deinit() {
  free(my_credits_used);
  free(my_credits_freed);
  free(my_credits);
  free(your_credits_freed);
  free(your_credits);
  free(your_credits_freed_last);
  free(iso_credit_packet_base);
  free(ord_remote_q);
  free(ord_remote_t);
  return SUCCESS;
}


// called by receive whenever it receieves an isotach credit packet
int iso_update_credit(int node_id, ULONG credit_info) {
  my_credits_freed[node_id] = credit_info;
  return SUCCESS;
}

// called by iso_delete_packet to send out an explicit credit packet
int iso_send_credit_packet(ULONG dest) {

  /* The target is written into the route, so that shipping will know which
     entry off of routes[] to use */
  iso_credit_packet_slot->route       = dest;
  iso_credit_packet_slot->credit_info = (ULONG)your_credits_freed[dest];
  iso_credit_packet_slot->address     = (ULONG)htonl((ULONG)iso_credit_packet_base[dest]);

  /*
     Uses shipping's ship_packet which only returns failure if the
     Lanai's send queue is full.  Since a packet will be put on the
     wire almost immediately, we sit and loop here.
  */
  while (ship_packet(iso_credit_packet_slot) == FAILURE) {
    ;
  }

  /* now we've notified the host about all slots freed */
  your_credits_freed_last[dest] = your_credits_freed[dest];

  return SUCCESS;

}


// called by iso_deliver once the application has processed a message
int iso_delete_packet(int node_id) {

  // free up a credit
  inc_idx(your_credits_freed[node_id], your_credits[node_id]);
```

```
  // if we haven't sent to the node in a while, send a credit packet
  if (queue_size(your_credits_freed_last[node_id], your_credits_freed[node_id],
                  your_credits[node_id]) >= iso_credit_packet_threshold) {
    iso_send_credit_packet(node_id);
  }

  return SUCCESS;
}


// called by Hostman's poll function
int iso_flow_poll() {
  int out_of_credits = FALSE;
  static int counter = 0;
  ULONG receiver;

  /*
     loop while there are packets to send and we are not out of credits
     at any receiver.
  */
  while (!queue_empty(iso_send_t, iso_send_flow_ptr) && !out_of_credits) {
    /*check to see if it is a BS Marker since they can always go out*/
    if (iso_send_flow_ptr->prefix & BS_MARKER_MASK) {
      inc_ptr(iso_send_flow_ptr, ISO_SEND_BUF_SIZE, iso_send_buf);
      printf("cleared a bsmarker...\n");
      continue;
    }

    /*otherwise we need to check if there is room at the receiving host*/
    receiver = iso_send_flow_ptr->route2;
    if (queue_full(my_credits_freed[receiver], my_credits_used[receiver],
                    my_credits[receiver])) {
      out_of_credits = TRUE;
    }

    /*if we can send the packet*/
    else {
      out_of_credits = FALSE;
      counter = 0;
      /*increment the number of credits used*/
      inc_idx(my_credits_used[receiver], my_credits[receiver]);


      /*check to see if there is credit info to piggy back*/
      if (your_credits_freed[receiver] == your_credits_freed_last[receiver]) {
          iso_send_flow_ptr->packet.credit_info = NULL_CREDIT;
      }
      else {
          your_credits_freed_last[receiver] = iso_send_flow_ptr->packet.credit_info =
your_credits_freed[receiver];
      }

      /*if its an iso pointer ... */
      if (iso_send_flow_ptr->packet.subtype == ISO_MBM) {

          /*set the dma address in the corresponding ord packet*/
          ord_flow_ptr->address = (ULONG)htonl((ULONG)ord_remote_t[receiver]);

          /*write the value into the pointer field of the iso_packet*/
          iso_send_flow_ptr->packet.body.pointer = (ULONG)ord_remote_t[receiver];

          /*increment our tail pointer into the remote ordered queue*/
          inc_ptr(ord_remote_t[receiver], REMOTE_ORD_SIZE,
                  ord_remote_q[receiver]);
          /*clear the ordered packet for shipping*/
          inc_ptr(ord_flow_ptr, ORD_SEND_BUF_SIZE, ord_send_buf);

      }

      /*regardless, clear the isotach packet for shipping*/
      inc_ptr(iso_send_flow_ptr, ISO_SEND_BUF_SIZE, iso_send_buf);

    }
```

```
  }

  return SUCCESS;
}
```

processing/iom/exports.h

```
/*
 * ---------------------------------------------------------------------------
 *
 * Isotach Module  : Isotach Ordering Module Exported Functions/Variables
 * Isotach Layer   : Procesing
 * Isotach Version : Version 3
 *
 * ---------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/flow_control/flow/exports.h,v $
 *       $Revision: 1.6 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:15 $
 *
 * ---------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This file contains all of the exported functions, variables and data
 * structures for the given module.  Other modules which need to access these
 * variables/functions should include this header file.
 *
 * ---------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * Included by the Hostman, iso_receive, and iso_send Modules
 *
 * ---------------------------------------------------------------------------
 */


#ifndef IOM_EXPORTS_H
#define IOM_EXPORTS_H

void bucketize(ULONG *pkt, UCHAR pkt_subtype);
void post_isochron(int net_isochron, ULONG self_count, int net_isochrons_sent);
int iom_init();
int iom_deinit();

#endif
```

processing/iom/locals.h

```
/*
 * ---------------------------------------------------------------------------
 *
 * Isotach Module  : Isotach Ordering Module Local Variable Header File
 * Isotach Layer   : Processing
 * Isotach Version : Version 3
 *
 * ---------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: $
 *       $Revision: 1.5 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:15 $
 *
 * ---------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This file contains all of the local variable and data structure definitions.
 * It also includes all other header files necessary for the module's function.
 * It also prototypes all functions internal to that module.
 *
 * ---------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * ---------------------------------------------------------------------------
 */

#ifndef IOM_LOCALS_H
#define IOM_LOCALS_H

#include <hostman/host_utils.h>
#include <processing/iom/exports.h>
#include <api/iso_deliver/exports.h>
#include <api/iso_send/exports.h>
#include <api/iso_signal/exports.h>
#include <api/iso_barrier/exports.h>
//#include <processing/shmem/exports.h>


// Masks used to determine if an EOP has barrier and/or signal bits set
#define BARRIER_MASK 0x03  // 000000xx
#define SIGNAL_MASK  0xFC  // xxxxxx00

// Size of a sort vector returned by the SIU
#define sort_vector_count 32


// a bucket type (really an arry or core 3 structs)
// bucket size is defined in constants.h
typedef PACKET_CORE_3 bucket[BUCKET_SIZE];

// the array of buckets.  bucket count is defined in constants.h
bucket buckets[BUCKET_COUNT];

// the array of tail pointers into the buckets
ULONG bucket_t[BUCKET_COUNT];


// used for self messages...  The hit_q is an array of core_2 isoslot structs
// we also have a head and tail pointer into this queue
#define HIT_Q_SIZE 8192
isoslot hit_q[HIT_Q_SIZE];
isoslot *hit_q_t;
isoslot *hit_q_h;

void print_bucket_entry(PACKET_CORE_3 *p, int num);
```

```
inline void process_bucket_entry(PACKET_CORE_3 *p);

#endif
```

processing/iom/iom.c

```c
/*
 * ---------------------------------------------------------------------------
 *
 * Isotach Module  : Isotach Ordering Module
 * Isotach Layer   : Processing
 * Isotach Version : Version 3
 *
 * ---------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/flow_control/flow/flow.c,v $
 *       $Revision: 1.3 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:15 $
 *
 * ---------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This module orders messages/accesses so they can be delivered in
 * isotach receive order.  Iso_receive passes in pointers to packets
 * so that they can be enqueued into the appropriate bucket or trigger
 * a draining of a bucket if it is an eop marker.  Additionally, when an
 * isochron that contains self messages/accesses is sent out, a marker is
 * stored here in the hit_q, so that when a pulse is executed, they can be
 * delivered to the application at the correct logical time.
 *
 * ---------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * ---------------------------------------------------------------------------
 */


#include <processing/iom/locals.h>


// Called by hostman to initialize all local data structures
int iom_init() {
  int i;


  /* Initialize head and tail pointers into the buckets */
  for (i=0; i<BUCKET_COUNT; i++) {
    bucket_t[i] = 0;
  }

  for (i=0; i < HIT_Q_SIZE; i++) {
    hit_q[i].subtype = ISO_SLOT;
  }

  hit_q_t = hit_q_h = hit_q;
  return SUCCESS;
}


// Also called by Hostman when application execution terminates
// Right now its a stub, but eventually who knows...
int iom_deinit() {

  return SUCCESS;
}



/*
   We currently support two different levels of ordering.
   MBM_VER 1 ignores eop markers, and deliveres messages
   to the application as soon as they are received by
```

```
   enqueueing them directly on the applications delivery queue
   Note:  this version does not support self messages.
*/
#if (MBM_VER==1)


// Called by iso_receive when an isotach packet is dma'd up to the
// host from the Lanai.
void bucketize(ULONG *pkt, UCHAR pkt_subtype) {
  eop_marker *eop_m;
  isochron_marker *iso_m;
  ISO_RECVFRAME *iso_p;

  // if it is an eop marker, check for signals but otherwise ignore
  if (pkt_subtype == EOP_MARKER) {
    eop_m = (eop_marker *)pkt;

    if (eop_m->bits & SIGNAL_MASK)
      iso_signal_notify(eop_m->bits);

  }
  // ignore isochron markers
  else if (pkt_subtype == ISO_MARKER) {
    ;
  }
  // if it is an isotach packet, enqueue it directly on the
  // isotach delivery queue
  else if (pkt_subtype == ISO_MBM) {
    // retrieve the address of the packet
    iso_p = (ISO_RECVFRAME *)pkt;

    // set the subtype on the delivery queue
    iso_delivery_t->subtype = iso_p->packet.subtype;

    // have the slot in the delivery queue point to the packet
    ((isopointer *)iso_delivery_t)->sender = iso_p->packet.sender;
    ((isopointer *)iso_delivery_t)->pointer = (PACKET_PTR)iso_p->packet.body.pointer;
    inc_ptr(iso_delivery_t, ISO_DELIVERY_SIZE, iso_delivery_q);
  }
  // missing a case for Shared Memory
  else {
    ;
  }
  return;
}
#endif


// used for debugging purposes
void print_bucket_entry(PACKET_CORE_3 *p, int num) {
  PACKET_PTR msg;
  if (p->subtype == ISO_MBM) {
    msg = (PACKET_PTR)((ULONG)((isopointer *)p)->pointer + offset);
    printf("Entry %d is pointing to %.20s\n",num,msg->data);
  }
  else if (p->subtype == ISO_MARKER) {
    printf("Entry %d has iso_id %d\n",num,((isomarker *)p)->iso_id);
  }
  else
    printf("Entry %d has subtype %d\n",num,p->subtype);
  return;
}


/*
 "Called" by MBM_VER 2 bucketize as a bucket is drained.
 It is separated out into an inline function merely for readability
*/
inline void process_bucket_entry(PACKET_CORE_3 *p) {

  // If it is a message, enqueue it on the isotach delivery queue
  if (p->subtype == ISO_MBM) {
    // only copy the first 2 words from the bucket entry since the
    // deliver queue consists of core_2 structures
```

```
    memcpy(iso_delivery_t, p, sizeof(PACKET_CORE_2));
    inc_ptr(iso_delivery_t, ISO_DELIVERY_SIZE, iso_delivery_q);
  }
  // If it is an isochron marker, check for self messages
  else if (p->subtype == ISO_MARKER) {

    // check that the isochron id of the marker equals the one at
    // the head of the hit_q
    if (((isomarker *)p)->iso_id != hit_q_h->iso_id) {
      printf("IOM: Error.  The ID of the isochron marker in ");
      printf("the bucket did not match up with the ID of the hit_q\n");
      printf("marker=%d, hit_q=%d\n",((isomarker *)p)->iso_id,hit_q_h->iso_id);
      shutdown(1);
    }

    // now copy the isochron slot into the delivery queue if there are
    // self messages to be delivered
    if (hit_q_h->self_count > 0) {
      memcpy(iso_delivery_t, (PACKET_CORE_2 *)hit_q_h, sizeof(PACKET_CORE_2));
      inc_ptr(iso_delivery_t, ISO_DELIVERY_SIZE, iso_delivery_q);
    }
    inc_ptr(hit_q_h, HIT_Q_SIZE, hit_q);
  }

  // sanity check since this should have been caught much further downstream
  else {
    printf("Found something unexpected in the bucket\n");
    shutdown(1);
  }

  return;
}


/*
  MBM_VER 2 supports full isotach ordering and self messages.  Note that
  the only difference between the two versions is the implementation of
  bucketize.
*/

#if (MBM_VER==2)

// Called by iso_receive when an isotach packet has been dma'd up to the
// host by the Lanai.
void bucketize(ULONG *pkt, UCHAR pkt_subtype) {
  eop_marker *eop_m;
  isochron_marker *iso_m;
  ISO_RECVFRAME *iso_p;
  UCHAR bits = 0x00;
  UCHAR TS;
  int i,j;

  if (pkt_subtype == EOP_MARKER) {
    eop_m = (eop_marker *)pkt;
    TS = eop_m->TS;

    /*check to ensure that the count equals the # of elts in the bucket*/
    if (eop_m->count != bucket_t[TS]) {
      printf("IOM: Received an incorrect sort vector from the SIU.  Aborting\n");
      shutdown(1);
    }

    /*
      If the SIU has sorted everything for us, we can process elements in
      the bucket according to the order provided in the sort vector
    */
    if (eop_m->count <= sort_vector_count) {
      for (i = 0; i < eop_m->count; i++) {
          process_bucket_entry(&buckets[TS][eop_m->sort_vector[i]]);
      }
    }

    /* Else there were too many isotach messages for this pulse*/
    /* Note: rather than sort the rest and merge, we will re-sort the entire
```

```
      bucket since for right now it is roughly equivalent and requires
      less copies
   */
   /* Second Note: since we currently have a small number of hosts, it seems
      that the most efficient sort is to simply loop through the bucket and
      and enqueue all items for this particular sender.  We know that they
      are in FIFO order for each sender and this will be a stable sort.
      This has complexity O(m*n), where m is the number of hosts, and n is
      the size of the bucket.  Since n is fairly small, most nlogn sorts have
      too much overhead, and since m is small, m*n will be n*n.  As the
      number of hosts in an isotach system grows, this sorting algorithm
      will need to be re-examined.
   */

   else {
     for (j = 0; j < number_of_hosts; j++) {
         for (i = 0; i < bucket_t[TS]; i++) {
           if (buckets[TS][i].sender == j) {
             process_bucket_entry(&buckets[TS][i]);
           }
         }
     }
   }

   /* reset bucket tail pointer */
   bucket_t[TS] = 0;

   bits = 0;

   /* now check for signals */
   if (eop_m->bits & SIGNAL_MASK)
     bits = iso_signal_notify(eop_m->bits);

   /* if there were any signals, deliver them to the application*/
   if (bits) {
     iso_delivery_t->subtype = BS_MARKER;
     ((bsnotice *)iso_delivery_t)->bits = bits;
     inc_ptr(iso_delivery_t, ISO_DELIVERY_SIZE, iso_delivery_q);
   }

   bits = 0;

   /* now check for barriers */
   if (eop_m->bits & BARRIER_MASK) {
     bits = iso_barrier_notify(eop_m->bits);
   }
   /* if there were any barriers, deliver them to the application*/
   if (bits) {
     iso_delivery_t->subtype = BS_MARKER;
     ((bsnotice *)iso_delivery_t)->bits = bits;
     inc_ptr(iso_delivery_t, ISO_DELIVERY_SIZE, iso_delivery_q);
   }
 }

 else if (pkt_subtype == ISO_MARKER) {
   iso_m = (isochron_marker *)pkt;
   TS = iso_m->TS;

   /* check to see if there is room in the bucket */
   if (bucket_t[TS] == BUCKET_SIZE) {
     printf("IOM: Bucket %d has overflowed.  Aborting.\n",(int)TS);
     for (i = 0; i< BUCKET_SIZE; i++) {
         print_bucket_entry(&buckets[TS][i], i);
     }
     shutdown(1);
   }

   /*
      Copy all pertinent information out of the marker and into the
      bucket entry
   */
   buckets[TS][bucket_t[TS]].subtype = pkt_subtype;
   ((isomarker *)&buckets[TS][bucket_t[TS]])->sender = ntohs(iso_m->source);
   ((isomarker *)&buckets[TS][bucket_t[TS]])->iso_id = iso_m->iso_id;
```

```
    bucket_t[TS]++;
  }
  else if (pkt_subtype == ISO_MBM) {
    iso_p = (ISO_RECVFRAME *)pkt;
    TS = iso_p->packet.TS;

    /* check to see if there is room in the bucket */
    if (bucket_t[TS] == BUCKET_SIZE) {
      printf("IOM: Bucket %d has overflowed.  Aborting.\n",(int)TS);
      for (i=0; i<BUCKET_SIZE; i++) {
          print_bucket_entry(&buckets[TS][i],i);
      }
      shutdown(1);
    }

    // copy the sender and address of the message into the bucket entry
    buckets[TS][bucket_t[TS]].subtype = pkt_subtype;
    ((isopointer *)&buckets[TS][bucket_t[TS]])->sender =
      (USHORT)iso_p->packet.sender;
    ((isopointer *)&buckets[TS][bucket_t[TS]])->pointer =
      (PACKET_PTR)iso_p->packet.body.pointer;
    bucket_t[TS]++;
  }
  // currently missing a case for Shared memory packets
  else {
    ;
  }
  return;
}
#endif



#if (MBM_VER == 1)
//version 1 does not support self messages so this is merely a stub
void post_isochron(int net_isochron, ULONG self_count, int net_isochrons_sent) {
  return;
}
#endif


#if (MBM_VER == 2)
// Called by iso_send whenever there are self messages
void post_isochron(int net_isochron, ULONG self_count, int net_isochrons_sent){
  isoslot *last_hit;
  PACKET_CORE_2 *last_delv;

  /*
     If the isochron containing the self messages had net components as well,
     we need to wait for the correct logical time before executing them.
     Thus, we enqueue an isoslot onto the hit_q
  */
  if (net_isochron == TRUE) {
    if (queue_full(hit_q_h,hit_q_t,HIT_Q_SIZE)) {
      printf("IOM: Error: the hit_q is full.  Re-adjust this parameter and recompile\n");
      shutdown(1);
    }
    hit_q_t->self_count = self_count;
    hit_q_t->iso_id     = net_isochrons_sent;
    inc_ptr(hit_q_t,HIT_Q_SIZE,hit_q);
  }

  /* otherwise, the isochron contained all self messsages */
  /*
     if the hit_q is empty, we can deliver them
     immediately to the application
  */
  else if (queue_empty(hit_q_h, hit_q_t)) {
    last_delv = q_last_item(iso_delivery_t, ISO_DELIVERY_SIZE, iso_delivery_q);
    /*
       if the last item of the delivery queue is an isoslot, add these
       messages to that isoslot.
    */
    if ((last_delv->subtype == ISO_SLOT) &&
```

```
        (!queue_empty(iso_delivery_h, iso_delivery_t))) {
      ((isoslot *)last_delv)->self_count += self_count;
    }

    /* otherwise, enqueue an isoslot onto the deliver queue */
    else {
      iso_delivery_t->subtype = ISO_SLOT;
      ((isoslot *)iso_delivery_t)->self_count = self_count;
      inc_ptr(iso_delivery_t, ISO_DELIVERY_SIZE, iso_delivery_q);
    }
  }

  /*
    if the hit_q wasn't empty, add these messages to
    the last isoslot on the hit_q
  */
  else {
    last_hit = q_last_item(hit_q_t,HIT_Q_SIZE,hit_q);
    last_hit->self_count += self_count;
  }
  return;
}
#endif
```

processing/shmem/exports.h

```
/*
 * -------------------------------------------------------------------------
 *
 * Isotach Module  : Isotach Shared Memory Module Exported Functions/Variables
 * Isotach Layer   : Procesing
 * Isotach Version : Version 3
 *
 * -------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/flow_control/flow/exports.h,v $
 *       $Revision: 1.6 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:15 $
 *
 * -------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This file contains all of the exported functions, variables and data
 * structures for the given module.  Other modules which need to access these
 * variables/functions should include this header file.
 *
 * -------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 *
 * -------------------------------------------------------------------------
 */

#ifndef SHMEM_EXPORTS_H
#define SHMEM_EXPORTS_H

//shmem_init()
//shmem_deinit()

#endif
```

processing/shmem/locals.h

```
/*
 * ------------------------------------------------------------------------
 *
 * Isotach Module  : Isotach Shared Memory Module Local Variable Header File
 * Isotach Layer   : Processing
 * Isotach Version : Version 3
 *
 * ------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: $
 *       $Revision: 1.5 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:15 $
 *
 * ------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This file contains all of the local variable and data structure definitions.
 * It also includes all other header files necessary for the module's function.
 * It also prototypes all functions internal to that module.
 *
 * ------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * ------------------------------------------------------------------------
 */

#ifndef SHMEM_LOCALS_H
#define SHMEM_LOCALS_H

#include <hostman/host_utils.h>
#include <processing/flow/exports.h>
//#include <api/getread/exports.h>
//#include <processing/iom/exports.h>
//#include <processing/shmem/exports.h>

#endif
```

processing/shmem/shmem.c

```
/*
 * -------------------------------------------------------------------------
 *
 * Isotach Module  : Isotach Shared Memory Module
 * Isotach Layer   : Processing
 * Isotach Version : Version 3
 *
 * -------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/flow_control/flow/flow.c,v $
 *       $Revision: 1.3 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:15 $
 *
 * -------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This is the Shared Memory Module which processes all Shared Memory
 * Requests, and maintains the shared memory.  This module executes the
 * Srefs on local portions of shared memory.
 *
 * -------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * Not Implemented Yet.
 *
 * -------------------------------------------------------------------------
 */

#include <processing/shmem/locals.h>
```

niu_interface/shipping/exports.h

```
/*
 * -------------------------------------------------------------------------
 *
 * Isotach Module  : Shipping Module Exported Functions/Variables
 * Isotach Layer   : NIU Interface
 * Isotach Version : Version 3
 *
 * -------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/niu_interface/shipping/exports.h,v $
 *       $Revision: 1.4 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:16 $
 *
 * -------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This file contains all of the exported functions, variables and data
 * structures for the given module.  Other modules which need to access these
 * variables/functions should include this header file.
 *
 * -------------------------------------------------------------------------
 *
 * COMMENTS:
 * Included by the Hostman Module, Flow Module and Send Module
 *
 * -------------------------------------------------------------------------
 */

#ifndef SHIPPING_EXPORTS_H
#define SHIPPING_EXPORTS_H

/* niu_send_buf is a buffer for outgoing messages that exists on the LANAI's
 * SRAM.  The following memory mapped variables point to the beginning of the
 * buffer, the head and the tail (respectively).  Exported to Hostman for
 * synchronization purposes.
 */
volatile PACKET *niu_send_buf;
volatile UCHAR *niu_send_h;
volatile UCHAR *niu_send_t;

int ship_packet(PACKET_PTR packet);

int shipping_init(node_info *nst);
int shipping_deinit();

int shipping_poll();

/* send_buf is a queue of PACKETS that need to be transferred to the
   niu_send_buf on the LANAI. */
PACKET send_buf[SEND_BUF_SIZE];

/* send_t points to the next slot in the queue in which the Send Module may
   create a packet to be sent */
PACKET_PTR send_t;

/* send_h points to the next element in the queue which the shipping module
   should take off and place on the niu_send_buf */
PACKET_PTR send_h;

/* send_flow_ptr points to the next element in the queue which the Flow Module
   needs to examine to determine if there are sufficient send credits to send
   the message */
PACKET_PTR send_flow_ptr;


/* these are for the ordered packets.  Each is similiar to the one above*/
PACKET ord_send_buf[ORD_SEND_BUF_SIZE];
```

```
PACKET_PTR ord_send_t;
PACKET_PTR ord_send_h;
PACKET_PTR ord_flow_ptr;



#endif
```

niu_interface/shipping/locals.h

```
/*
 * -------------------------------------------------------------------------
 *
 * Isotach Module  : Shipping Module Local Variable Header File
 * Isotach Layer   : NIU Interface
 * Isotach Version : Version 3
 *
 * -------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/niu_interface/shipping/locals.h,v $
 *       $Revision: 1.6 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:16 $
 *
 * -------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This file contains all of the local variable and data structure definitions.
 * It also includes all other header files necessary for the module's function.
 * It also prototypes all functions internal to that module.
 *
 * -------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * -------------------------------------------------------------------------
 */

#ifndef SHIPPING_LOCALS_H
#define SHIPPING_LOCALS_H

#include <hostman/host_utils.h>
#include <processing/flow/exports.h>
#include <niu_interface/shipping/exports.h>

/* An array of routes from THIS host to every other host in the network */
ULONG *routes;

#endif
```

niu_interface/shipping/shipping.c

```
/*
 * ------------------------------------------------------------------------
 *
 * Isotach Module  : Shipping Module
 * Isotach Layer   : NIU Interface
 * Isotach Version : Version 3
 *
 * ------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/niu_interface/shipping/shipping.c,v $
 *       $Revision: 1.5 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:16 $
 *
 * ------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This module takes packets from the send_buf, after they have been cleared
 * by the Flow Module and places them on the niu_send_buf for network
 * transmission.
 *
 * ------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * ------------------------------------------------------------------------
 */

#include <niu_interface/shipping/locals.h>


// used by flow and iso_flow to ship out credit packets and credit requests
// as well as the barrier module to send out barrier packets
int ship_packet(PACKET_PTR packet) {

  if (!queue_full(*niu_send_h, *niu_send_t, NIU_SEND_SIZE)) {

    packet->route = routes[packet->route];

    /* Copy the actual packet from the send_buf to the niu_send_buf */
    /* NOTE: this should eventually be optimized using write combining
       or some other sort of PCI bus magic*/
    memcpy((PACKET_PTR)&niu_send_buf[*niu_send_t], packet, sizeof(PACKET));

    /* Incremenet the niu_send_t to inform Netman that a new packet has been
       placed on its buffer */
    inc_idx(*niu_send_t, NIU_SEND_SIZE);

    return SUCCESS;
  }
  else {
    return FAILURE;
  }
}


// called by hostman when the messaging layer is terminated
int shipping_deinit() {
  free(routes);
  return SUCCESS;
}


// called by hostman when the messaging layer is initialized
int shipping_init(node_info *nst) {

  int i;
```

```
  routes = (ULONG *)malloc(number_of_hosts * sizeof(ULONG));

  // retrieve the routes from the network status table
  for (i = 0; i < number_of_hosts; i++) {
    routes[i] = nst[i].route;
  }

  // retrieve pointers into the send buffer from the Lanai
  niu_send_buf = (PACKET_PTR)get_lanai_sym("_niu_send_buf");
  niu_send_h   = (UCHAR *)get_lanai_sym("_niu_send_h");
  niu_send_t   = (UCHAR *)get_lanai_sym("_niu_send_t");


  // initialize all host send buffer pointers
  send_t     = send_h     = send_flow_ptr = send_buf;
  ord_send_t = ord_send_h = ord_flow_ptr  = ord_send_buf;


  // fill in pre-determined information in the ordered send buffer
  for (i = 0; i < ORD_SEND_BUF_SIZE; i++) {
    ord_send_buf[i].type        = (USHORT)htons(NONISO);
    ord_send_buf[i].subtype     = ORDERED;
    ord_send_buf[i].sender      = (ULONG)host_node_id;
    ord_send_buf[i].credit_info = NULL_CREDIT;
    ord_send_buf[i].pad1        = 0x00;
  }

  return SUCCESS;
}



/* Shipping Poll is called by the main poll function in hostman.  It checks
   to see if there is anything outstanding to send.  If there are any packets
   between send_h and send_flow_ptr, they are all put on the niu_send_queue
*/
int shipping_poll() {
  UCHAR local_niu_send_h;

  /* If there are no NONISO Packets to send, return now. */
  if (queue_empty(send_h, send_flow_ptr) && queue_empty(ord_send_h,
                                                        ord_flow_ptr))
    return FAILURE;

  // take a snapshot of the head pointer
  local_niu_send_h = *niu_send_h;

  /* if there are noniso and ordered packets to send out,
     alternate sending them */

  while (!queue_empty(send_h, send_flow_ptr) &&
           !queue_empty(ord_send_h, ord_flow_ptr)) {

    /*
      if the Lanai's send buffer seems to be full, first take
      a new snapshot of the head pointer.  Then if it is still full,
      return failure.
    */
    if (queue_full(local_niu_send_h, *niu_send_t, NIU_SEND_SIZE)) {
      local_niu_send_h = *niu_send_h;
      if (queue_full(local_niu_send_h, *niu_send_t, NIU_SEND_SIZE)) {
          return FAILURE;
      }
    }


    /*send the nonisotach packet out*/

    /* Write the route from the routing table into the route field in the out-
       going packet.  The target host number is stored in the route field, and
       is overwritten with the routing bytes */
    send_h->route = routes[send_h->route];

    /* Copy the actual packet from the send_buf to the niu_send_buf */
```

```
    /* Need to copy the entire size of the packet structure since the
       route is at the end */
    /* Note: this needs to be optimized with write combining or some other
       form of PCI bus magic */
    memcpy((PACKET_PTR)&niu_send_buf[*niu_send_t], send_h, sizeof(PACKET));

    /* Incremenet the niu_send_t to inform Netman that a new packet has been
       placed on its buffer */
    inc_idx(*niu_send_t, NIU_SEND_SIZE);

    /* Free up the slot used by this packet on the send_buf */
    inc_ptr(send_h, SEND_BUF_SIZE, send_buf);


    /*recheck to see if the queue is full*/
    if (queue_full(local_niu_send_h, *niu_send_t, NIU_SEND_SIZE)) {
      local_niu_send_h = *niu_send_h;
      if (queue_full(local_niu_send_h, *niu_send_t, NIU_SEND_SIZE)) {
          return FAILURE;
      }
    }


    /*send the ordered packet out*/

    /* Write the route from the routing table into the route field in the out-
       going packet.  The target host number is stored in the route field, and
       is overwritten with the routing bytes */
    ord_send_h->route = routes[ord_send_h->route];

    /* Copy the actual packet from the send_buf to the niu_send_buf */
    /* Note: this needs to be optimized */
    memcpy((PACKET_PTR)&niu_send_buf[*niu_send_t], ord_send_h, sizeof(PACKET));

    /* Incremenet the niu_send_t to inform Netman that a new packet has been
       placed on its buffer */
    inc_idx(*niu_send_t, NIU_SEND_SIZE);

    /* Free up the slot used by this packet on the send_buf */
    inc_ptr(send_h, ORD_SEND_BUF_SIZE, ord_send_buf);
  }


  /*note that at most one of the following two loops will be entered*/

  /*now send out any extraneous nonisotach packets*/
  while (!queue_empty(send_h, send_flow_ptr)) {

    /* check to see if the send buffer on the Lanai is full */
    if (queue_full(local_niu_send_h, *niu_send_t, NIU_SEND_SIZE)) {
      local_niu_send_h = *niu_send_h;
      if (queue_full(local_niu_send_h, *niu_send_t, NIU_SEND_SIZE)) {
          return FAILURE;
      }
    }

    /* Write the route from the routing table into the route field in the out-
       going packet.  The target host number is stored in the route field, and
       is overwritten with the routing bytes */
    send_h->route = routes[send_h->route];

    /* Copy the actual packet from the send_buf to the niu_send_buf */
    /* Note: this needs to be optimized */
    memcpy((PACKET_PTR)&niu_send_buf[*niu_send_t], send_h, sizeof(PACKET));

    /* Incremenet the niu_send_t to inform Netman that a new packet has been
       placed on its buffer */
    inc_idx(*niu_send_t, NIU_SEND_SIZE);

    /* Free up the slot used by this packet on the send_buf */
    inc_ptr(send_h, SEND_BUF_SIZE, send_buf);
  }
```

```
  /*now send out any extraneous ordered packets*/
  while (!queue_empty(ord_send_h, ord_flow_ptr)) {

    /* check to see if the Lanai's send buffer is full */
    if (queue_full(local_niu_send_h, *niu_send_t, NIU_SEND_SIZE)) {
      local_niu_send_h = *niu_send_h;
      if (queue_full(local_niu_send_h, *niu_send_t, NIU_SEND_SIZE)) {
          return FAILURE;
      }
    }

    /* Write the route from the routing table into the route field in the out-
       going packet.  The target host number is stored in the route field, and
       is overwritten with the routing bytes */
    ord_send_h->route = routes[ord_send_h->route];


    /* Copy the actual packet from the send_buf to the niu_send_buf */
    /* Note: this needs to be optimized */
    memcpy((PACKET_PTR)&niu_send_buf[*niu_send_t], ord_send_h, sizeof(PACKET));

    /* Incremenet the niu_send_t to inform Netman that a new packet has been
       placed on its buffer */
    inc_idx(*niu_send_t, NIU_SEND_SIZE);

    /* Free up the slot used by this packet on the send_buf */
    inc_ptr(ord_send_h, ORD_SEND_BUF_SIZE, ord_send_buf);
  }

  return SUCCESS;
}
```

niu_interface/receive/exports.h

```
/*
 * ---------------------------------------------------------------------------
 *
 * Isotach Module  : Receive Module Exported Functions/Variables
 * Isotach Layer   : NIU Interface
 * Isotach Version : Version 3
 *
 * ---------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/niu_interface/receive/exports.h,v $
 *       $Revision: 1.3 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:16 $
 *
 * ---------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This file contains all of the exported functions, variables and data
 * structures for the given module.  Other modules which need to access these
 * variables/functions should include this header file.
 *
 * ---------------------------------------------------------------------------
 *
 * COMMENTS:
 * Included by the Hostman Module
 *
 * ---------------------------------------------------------------------------
 */

#ifndef RECEIVE_EXPORTS_H
#define RECEIVE_EXPORTS_H

int receive_init(node_info *nst);
int receive_deinit();
int receive_poll();


#endif
```

niu_interface/receive/locals.h

```
/*
 * ---------------------------------------------------------------------------
 *
 * Isotach Module  : Receive Module Local Variable Header File
 * Isotach Layer   : NIU Interface
 * Isotach Version : Version 3
 *
 * ---------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/niu_interface/receive/locals.h,v $
 *       $Revision: 1.3 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:16 $
 *
 * ---------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This file contains all of the local variable and data structure definitions.
 * It also includes all other header files necessary for the module's function.
 * It also prototypes all functions internal to that module.
 *
 * ---------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * ---------------------------------------------------------------------------
 */

#ifndef RECEIVE_LOCALS_H
#define RECEIVE_LOCALS_H

#include <hostman/host_utils.h>
#include <api/deliver/exports.h>
#include <api/iso_deliver/exports.h>
#include <processing/flow/exports.h>
#include <niu_interface/receive/exports.h>

// pointers to the delivery queue on the lanai
volatile PACKET_PTR *niu_delivery_q;
volatile ULONG *niu_delivery_h;
volatile ULONG *niu_delivery_t;

// array of pointers for each node's ordered packet receive buffer
PACKET_PTR *ord_receive_buf;

// array of sizes for each node's ordered packet receive buffer
ULONG      *ord_receive_buf_limit;


#endif
```

niu_interface/receive/receive.c

```
/*
 * --------------------------------------------------------------------------
 *
 * Isotach Module  : Receive Module
 * Isotach Layer   : NIU Interface
 * Isotach Version : Version 3
 *
 * --------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/niu_interface/receive/receive.c,v $
 *       $Revision: 1.2 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:16 $
 *
 * --------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 *
 * --------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * --------------------------------------------------------------------------
 */

#include <niu_interface/receive/locals.h>


// called by hostman when the application closes the messaging layer
int receive_deinit() {
  free(ord_receive_buf);
  free(ord_receive_buf_limit);
  free(ord_receive_t);
  return SUCCESS;
}


// called by hostman when the messaging layer is initialized
int receive_init(node_info *nst) {
  int i;

  // retrieve the values from the Lanai
  niu_delivery_q  = (PACKET_PTR *)get_lanai_sym("_niu_delivery_q");
  niu_delivery_h  = (ULONG *)get_lanai_sym("_niu_delivery_h");
  niu_delivery_t  = (ULONG *)get_lanai_sym("_niu_delivery_t");

  ord_receive_buf = (PACKET_PTR *)malloc(number_of_hosts * sizeof(PACKET_PTR));
  ord_receive_buf_limit = (ULONG *)malloc(number_of_hosts * sizeof(ULONG));
  ord_receive_t   = (PACKET_PTR *)malloc(number_of_hosts * sizeof(PACKET_PTR));

  for (i = 0; i < number_of_hosts; i++) {
    ord_receive_buf[i] = (PACKET_PTR)((ULONG)nst[i].local_iso.base + offset);
    ord_receive_t[i] = ord_receive_buf[i];
    ord_receive_buf_limit[i] = nst[i].local_iso.size;
  }

  /* this is our sanity check...  To make sure that we aren't fooling
     ourselves, we zero out pinned memory.  A VERY useful debugging tool
     when things are foobar'd */
  /*
    bzero((void *)ord_receive_buf[0],
    nst[host_node_id].local_iso.size*sizeof(PACKET));
    bzero((void *)ord_receive_buf[1],
    nst[host_node_id].local_iso.size*sizeof(PACKET));
  */

  return SUCCESS;
}
```

```
// called by hostman's poll function
int receive_poll() {
  ULONG current_niu_delivery_t;
  ULONG temp_niu_delivery_h;
  PACKET_PTR this_packet;

  /* Find out where the tail on the niu_delivery queue is at the current
     time */
  current_niu_delivery_t = ntohl(*niu_delivery_t);

  /* Process all packets on the niu_delivery_q up until the value of
     current_niu_delivery_t.  This means that receive_poll() should always
     handle as many packets as it thought needed to be handled at the beginning
     of the function call.  If new packets are queued up by the LANAI, these
     should be handled on the next call to receive_poll() */

  while(!queue_empty(ntohl(*niu_delivery_h), current_niu_delivery_t)) {
    /* Read the address in pinned memory of the next packet to be handled
       off of the niu_delivery_q */
    this_packet = (PACKET_PTR)(ntohl((ULONG)niu_delivery_q[ntohl(*niu_delivery_h)]) + offset);

    /* Once this address has been read in, increment the niu_delivery_h to let
       the LANAI know that space has been freed.  We need a temporary copy
       since the head needs to be switched.
    */

    temp_niu_delivery_h = ntohl(*niu_delivery_h);
    inc_idx(temp_niu_delivery_h, NIU_DELV_SIZE);
    *niu_delivery_h = htonl(temp_niu_delivery_h);

    /* Check to see if the CRC Field is NON-zero.  If it is NON-zero, the
       packet has been corrupted and must be handled */
    /* The CRC byte is found by looking into the payload of the packet.  When
       the packet is received off of the network, the crc byte is always
       present in the 1st byte of the word immediatly following the packet
       payload.  This is because the packet is word aligned when it is sent
       out by the sending host. */
    /*
      if ((UCHAR)(this_packet->data[ntohs(this_packet->pad2) -
      PACKET_HEADER_SIZE - 4]) != 0) {
      printf("ERROR: Bad CRC Received.  Aborting.\n");
      print_out_packet(this_packet);
      printf("Packets Received = %d\n", pkt_counter);

      shutdown(1);
      }
    */

    // if it is an isotach credit packet, update credit info
    if (this_packet->subtype == ISO_CREDIT)
      iso_update_credit(this_packet->sender,
                        (PACKET_PTR)this_packet->credit_info);

    // if there is credit info piggy backed, update credit info
    else if (this_packet->credit_info != NULL_CREDIT)
      update_credit(this_packet->sender, (PACKET_PTR)this_packet->credit_info);

    /* If the incoming packet is a NONISO Message, send it to the Delivery
       module */
    if (this_packet->subtype == NONISO_MBM) {
      if (!queue_full(delivery_h, delivery_t, DELIVERY_SIZE)) {
          delivery_q[delivery_t] = this_packet;
          inc_idx(delivery_t, DELIVERY_SIZE);
      }
      else {
          printf("ERROR: Delivery Queue is Full!  Aborting.\n");
          shutdown(1);
      }
    }
    else if (this_packet->subtype == ORDERED) {
      /* tell iso_deliver that the ord packet is here */
      inc_ptr(ord_receive_t[this_packet->sender],
```

```
              ord_receive_buf_limit[this_packet->sender],
              ord_receive_buf[this_packet->sender]);
    }

    /* remember that all credit information was extracted previously */
    else if (this_packet->subtype == CREDIT) {
      //printf("Received a credit packet from %d w/ new head at %lu\n", this_packet->sender,
(ULONG)this_packet->credit_info);
      fflush(stdout);
    }
    else if (this_packet->subtype == ISO_CREDIT) {
      //printf("Received an iso_credit packet from %d w/ new head at %lu\n", this_packet->sender,
(ULONG)this_packet->credit_info);
      fflush(stdout);
    }
    /* if it is a noniso barrier, notify the barrier module */
    else if (this_packet->subtype == BARRIER) {
      // printf("Received a barrier packet from %d\n", this_packet->sender);
      fflush(stdout);
      process_barrier(this_packet->sender);
    }
    /* if it is a credit request, send out a credit packet */
    else if (this_packet->subtype == REQUEST_CREDIT) {
      //printf("received a credit request from host %d\n",this_packet->sender);
      fflush(stdout);
      send_credit_packet(this_packet->sender);
    }
    /* this should hopefully never happed... */
    else {
      printf("Received something we are really confused about... bailing\n");
      print_out_packet(this_packet);
      shutdown(1);
    }
  }

  return SUCCESS;
}
```

niu_interface/iso_shipping/exports.h

```
/*
 * -------------------------------------------------------------------------
 *
 * Isotach Module  : Iso_Shipping Module Exported Functions/Variables
 * Isotach Layer   : NIU Interface
 * Isotach Version : Version 3
 *
 * -------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/niu_interface/shipping/exports.h,v $
 *       $Revision: 1.4 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:16 $
 *
 * -------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This file contains all of the exported functions, variables and data
 * structures for the given module.  Other modules which need to access these
 * variables/functions should include this header file.
 *
 * -------------------------------------------------------------------------
 *
 * COMMENTS:
 * Included by the Hostman Module, iso_Flow Module and iso_Send Module
 *
 * -------------------------------------------------------------------------
 */

#ifndef ISO_SHIPPING_EXPORTS_H
#define ISO_SHIPPING_EXPORTS_H

/* iso_niu_send_buf is a buffer for outgoing messages that exists
 * on the LANAI's SRAM.  The following memory mapped variables point
 * to the beginning of the buffer, the head and the tail (respectively).
 */
volatile ISO_SENDFRAME *iso_niu_send_buf;
volatile UCHAR *iso_niu_send_h;
volatile UCHAR *iso_niu_send_t;


int iso_shipping_init(node_info *nst);
int iso_shipping_deinit();
int iso_shipping_poll();

/* iso_send_buf is a queue of ISO_SENDFRAMES that need to be transferred
   to the niu_send_buf on the LANAI. */
ISO_SENDFRAME iso_send_buf[ISO_SEND_BUF_SIZE];

/* iso_send_t points to the next slot in the queue in which the
   Isotach Send Module may create a packet to be sent */
ISO_SENDFRAME *iso_send_t;

/* iso_send_h points to the next element in the queue which the
   isotach shipping module should take off and place on the niu_send_buf */
ISO_SENDFRAME *iso_send_h;

/* iso_send_flow_ptr points to the next element in the queue
   which the Isotach Flow Module needs to examine to determine
   if there are sufficient send credits to send the message */
ISO_SENDFRAME *iso_send_flow_ptr;

#endif
```

niu_interface/iso_shipping/locals.h

```
/*
 * -------------------------------------------------------------------------
 *
 * Isotach Module  : Iso_Shipping Module Local Variable Header File
 * Isotach Layer   : NIU Interface
 * Isotach Version : Version 3
 *
 * -------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/niu_interface/shipping/locals.h,v $
 *       $Revision: 1.6 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:16 $
 *
 * -------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This file contains all of the local variable and data structure definitions.
 * It also includes all other header files necessary for the module's function.
 * It also prototypes all functions internal to that module.
 *
 * -------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * -------------------------------------------------------------------------
 */

#ifndef ISO_SHIPPING_LOCALS_H
#define ISO_SHIPPING_LOCALS_H

#include <hostman/host_utils.h>
#include <processing/iso_flow/exports.h>
#include <niu_interface/iso_shipping/exports.h>

/* The SIU hardware expects a route to be at least 6 bytes long with a pad */
typedef struct {
  ULONG route2;
  USHORT route1;
} iso_route;

/* An array of routes from THIS host to every other host in the network */
iso_route *iso_routes;

#endif
```

niu_interface/iso_shipping/iso_shipping.c

```
/*
 * -------------------------------------------------------------------------
 *
 * Isotach Module  : ISO_SHIPPING Module
 * Isotach Layer   : niu_inteface
 * Isotach Version : Version 3
 *
 * -------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/niu_interface/shipping/shipping.c,v $
 *       $Revision: 1.5 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:16 $
 *
 * -------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This module takes packets from the iso_send_buf, after they have been
 * cleared  by the Flow Module and places them on the niu_send_buf for network
 * transmission.
 *
 * -------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * -------------------------------------------------------------------------
 */

#include <niu_interface/iso_shipping/locals.h>

int iso_shipping_deinit() {
  free(iso_routes);
  return SUCCESS;
}


// called by hostman when the messaging layer is initialized
int iso_shipping_init(node_info *nst) {

  int i;
  ULONG route;
  iso_routes   = (iso_route *)malloc(number_of_hosts * sizeof(iso_route));

  /*initialize all routing bytes to zero*/
  memset(iso_routes, 0x00, number_of_hosts * sizeof(iso_route));


  /* The SIU expects the route to range from 2 - 6 bytes,
     depending on the number of routing bytes.  Since
     nonisotach supports at most 4 routing bytes, we are
     limiting isotach to that as well.  However, to accomodate
     the way in which routes are used by the SIU, we need to split
     it across two different fields in the ISO_SENDFRAME structure.
     Below is a list of the four cases, and how the fields need to
     be populated for each.  The fields are listed in network order,
     so byte ordering will have to be switched:

     1 byte route:  route1 -> 00R1
                    route2 -> 00000000

     2 byte route:  route1 -> R1R2
                    route2 -> 00000000

     3 byte route:  route1 -> 0000
                    route2 -> 00R1R2R3

     4 byte route:  route1 -> 0000
                    route2 -> R1R2R3R4
```

```
  */

  /* the following algorithm could be accomplished with 2 cases,
     however for readability, we explicitly list all 4 cases */

  for (i = 0; i < number_of_hosts; i++) {
    /* first put the route back in host ordering */
    route = (ULONG)ntohl(nst[i].route);

    /* if the third byte is zero, we have a one byte route */
    if ( (route & THIRD_BYTE) == 0) {
      iso_routes[i].route1 = (USHORT)htons((USHORT)route);
      iso_routes[i].route2 = 0;
    }
    /* if the second byte is zero, we have a two byte route */
    else if ( (route & SECOND_BYTE) == 0) {
      iso_routes[i].route1 = (USHORT)htons((USHORT)route);
      iso_routes[i].route2 = 0;
    }
    /* if the first byte is zero, we have a three byte route */
    else if ( (route & FIRST_BYTE) == 0) {
      iso_routes[i].route1 = 0;
      iso_routes[i].route2 = (ULONG)htonl(route);
    }
    /* otherwise it is a four byte route */
    else {
      iso_routes[i].route1 = 0;
      iso_routes[i].route2 = (ULONG)htonl(route);
    }
  }


  /* set the type field to ISO */
  for (i = 0; i < ISO_SEND_BUF_SIZE; i++) {
    iso_send_buf[i].packet.type = htons(ISO);
  }


  /* retrieve pointers into the isotach send buffer on the Lanai */
  iso_niu_send_buf = (ISO_SENDFRAME *)get_lanai_sym("_iso_niu_send_buf");
  iso_niu_send_h   = (UCHAR *)get_lanai_sym("_iso_niu_send_h");
  iso_niu_send_t   = (UCHAR *)get_lanai_sym("_iso_niu_send_t");

  /* initialize the queue pointers for the isotach send buffer on the host */
  iso_send_t = iso_send_h = iso_send_flow_ptr = iso_send_buf;

  return SUCCESS;
}


/* iso_Shipping Poll is called by the main poll function in hostman.
   It checks to see if there is anything outstanding to send.
   If there are any packets between iso_send_h and iso_send_flow_ptr,
   they are all put on the iso_niu_send_queue */
int iso_shipping_poll() {
  UCHAR local_iso_niu_send_h;

  /* If there are no NONISO Packets to send, return now. */
  if (queue_empty(iso_send_h, iso_send_flow_ptr))
    return FAILURE;

  // take a snapshot of the head pointer
  local_iso_niu_send_h = *iso_niu_send_h;

  // while there are packets to ship
  while (!queue_empty(iso_send_h, iso_send_flow_ptr)) {

    // if the queue appears full, first take a new snapshot of the head
    // if it is still full then return failure
    if (queue_full(local_iso_niu_send_h, *iso_niu_send_t, ISO_NIU_SEND_SIZE)) {
      local_iso_niu_send_h = *iso_niu_send_h;

      if (queue_full(local_iso_niu_send_h, *iso_niu_send_t,
                     ISO_NIU_SEND_SIZE)) {
```

```
        //printf("\t\tISO_NIU_SEND_Q Full!!!  Leaving ISO_Shipping!!!\n");
        return FAILURE;
      }
    }

    /* look up the route in the table and write it into the outgoing packet */
    iso_send_h->route1 = iso_routes[iso_send_h->route2].route1;
    iso_send_h->route2 = iso_routes[iso_send_h->route2].route2;

    /* Switch the byte ordering of the sender field */
    iso_send_h->packet.sender = (USHORT)htons(iso_send_h->packet.sender);

    /* Switch the byte ordering of the prefix */
    iso_send_h->prefix = (ULONG)htonl(iso_send_h->prefix);

    /* Copy the actual packet from the send_buf to the niu_send_buf */
    /* Note: this needs to be optimized using write combining or some
       other form of PCI bus magic */
    memcpy((ISO_SENDFRAME *)&iso_niu_send_buf[*iso_niu_send_t],
           iso_send_h, sizeof(ISO_SENDFRAME));

    /* Incremenet the iso_niu_send_t to inform Netman that a
       new packet has been placed on its buffer */
    inc_idx(*iso_niu_send_t, ISO_NIU_SEND_SIZE);

    /* Free up the slot used by this packet on the iso_send_buf */
    inc_ptr(iso_send_h, ISO_SEND_BUF_SIZE, iso_send_buf);
  }

  return SUCCESS;
}
```

niu_interface/iso_receive/exports.h

```
/*
 * ------------------------------------------------------------------------
 *
 * Isotach Module  : Iso Receive Module Exported Functions/Variables
 * Isotach Layer   : NIU Interface
 * Isotach Version : Version 3
 *
 * ------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source:$
 *       $Revision: 1.3 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:16 $
 *
 * ------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This file contains all of the exported functions, variables and data
 * structures for the given module.  Other modules which need to access these
 * variables/functions should include this header file.
 *
 * ------------------------------------------------------------------------
 *
 * COMMENTS:
 * Included by the Hostman Module
 *
 * ------------------------------------------------------------------------
 */

#ifndef ISO_RECEIVE_EXPORTS_H
#define ISO_RECEIVE_EXPORTS_H

int iso_receive_init(ULONG *iso_base_ptr);
int iso_receive_deinit();
int iso_receive_poll();

#endif
```

niu_interface/iso_receive/locals.h

```
/*
 * --------------------------------------------------------------------------
 *
 * Isotach Module  : Iso Receive Module Local Variable Header File
 * Isotach Layer   : NIU Interface
 * Isotach Version : Version 3
 *
 * --------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source:$
 *       $Revision: 1.3 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:16 $
 *
 * --------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This file contains all of the local variable and data structure definitions.
 * It also includes all other header files necessary for the module's function.
 * It also prototypes all functions internal to that module.
 *
 * --------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * --------------------------------------------------------------------------
 */

#ifndef ISO_RECEIVE_LOCALS_H
#define ISO_RECEIVE_LOCALS_H

#include <hostman/host_utils.h>
#include <processing/iom/exports.h>
#include <processing/iso_flow/exports.h>
#include <niu_interface/iso_receive/exports.h>
#include <api/iso_signal/exports.h>


/* The iso_recv buffer is a buffer with elements of word length.  The normal
   queue manipulation functions do not apply to this queue because pointer
   increments are not of uniform size.  They depend on the size of the
   packet stored (i.e. iso_pointer, iso_sref, EOP or isochron marker)
   Both of these pointers are to virtual addresses because they are only
   used on the host.*/
ULONG *iso_recv_buf;
ULONG *iso_recv_h;

/* These pointers are pointers into the iso_recv_buf, but because they are
   modified on the LANai, they need to exist on the LANai's SRAM
   These addresses are physical addresses into pinned memory.
   iso_receive_buf_start is only used on the LANAI, so physical to virtual
   address calculations never need to be done on it.  iso_recv_t is looked
   at on the host (to determine if the queue is empty) and the offset must
   be added to it to get the virtual address of the iso_recv_t */
volatile ULONG *iso_recv_t;
volatile ULONG *iso_recv_start;
volatile ULONG *iso_recv_size;

#endif
```

niu_interface/iso_receive/iso_receive.c

```
/*
 * -------------------------------------------------------------------------
 *
 * Isotach Module  : Iso Receive Module
 * Isotach Layer   : NIU Interface
 * Isotach Version : Version 3
 *
 * -------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source:$
 *       $Revision: 1.2 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:16 $
 *
 * -------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 *
 * -------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * -------------------------------------------------------------------------
 */

#include <niu_interface/iso_receive/locals.h>


// called by hostman when the application shuts down the messaging layer
// right now it is a stub since there is nothing to do
int iso_receive_deinit() {
  return SUCCESS;
}

// called by hostman when the messaging layer is initialized
int iso_receive_init(ULONG *iso_base_ptr) {

  iso_recv_t      = (ULONG *)get_lanai_sym("_iso_recv_t");
  iso_recv_start  = (ULONG *)get_lanai_sym("_iso_recv_start");
  iso_recv_size   = (ULONG *)get_lanai_sym("_iso_recv_size");

  iso_recv_buf    = (ULONG *)((ULONG)iso_base_ptr + offset);

  *iso_recv_t     = (ULONG)htonl((ULONG)iso_base_ptr);
  *iso_recv_start = (ULONG)htonl((ULONG)iso_base_ptr);
  *iso_recv_size  = (ULONG)htonl(ISO_RECV_SIZE);

  iso_recv_h      = iso_recv_buf;

  return SUCCESS;
}

// called by hostman's poll function
int iso_receive_poll() {
  ULONG *current_iso_recv_t;
  UCHAR pkt_subtype;
  eop_marker *eop_m;
  isochron_marker *iso_m;
  ISO_RECVFRAME *iso_p;

  /* take a snapshot of the tail pointer and add the offset */
  current_iso_recv_t = (ULONG *)((ULONG)ntohl(*iso_recv_t) + offset);

  /* loop until the head equals the snapshot of the tail */
  while (!queue_empty(iso_recv_h, current_iso_recv_t)) {
    /* if we have received a stop packet from the Lanai,
       reset the head pointer back to the beginning */
    if (*iso_recv_h == STOP_PACKET) {
```

```
    iso_recv_h = iso_recv_buf;
    continue;
  }

  /*The packet subtype is the third byte of the first word of the packet */
  pkt_subtype = ((ISO_RECVFRAME *)iso_recv_h)->packet.subtype;

  /*We have different tasks, depending on what we receive*/

  if (pkt_subtype == EOP_MARKER) {
    eop_m = (eop_marker *)iso_recv_h;

    /* Check the CRC */
    if (eop_m->crc) {
        printf("ISOTACH RECEIVING: CRC Error on EOP!\n");
        shutdown(1);
    }


    /* Insert the EOP marker into a bucket */
    bucketize(iso_recv_h, pkt_subtype);

    /* Increment the head pointer */
    iso_recv_h += EOP_MARKER_SIZE;
  }

  else if (pkt_subtype == ISO_MARKER) {
    iso_m = (isochron_marker *)iso_recv_h;

    /* Check the CRC */
    if (iso_m->crc) {
        printf("ISOTACH RECEIVING: CRC Error on ISO_MARKER!\n");
        print_out_iso_marker(iso_m);
        shutdown(1);
    }

    /* Insert the Isochron marker into a bucket */
    bucketize(iso_recv_h, pkt_subtype);

    /* Increment the head pointer */
    iso_recv_h += ISO_MARKER_SIZE;
  }

  else if ( (pkt_subtype == ISO_MBM)    ||
            (pkt_subtype == ISO_READ)   ||
            (pkt_subtype == ISO_WRITE)  ||
            (pkt_subtype == ISO_ASSIGN) ||
            (pkt_subtype == ISO_SCHED)  ) {

    iso_p = (ISO_RECVFRAME *)iso_recv_h;


    /* Check the CRC */
    if (iso_p->crc) {
        printf("ISOTACH RECEIVING: CRC Error! on ISO_PACKET\n");
        shutdown(1);
    }

    /* switch the sender back to host ordering */
    iso_p->packet.sender = (USHORT)ntohs(iso_p->packet.sender);

    /* Extract credit information */
    if (iso_p->packet.credit_info != NULL_CREDIT) {
        iso_update_credit((int)iso_p->packet.sender,iso_p->packet.credit_info);
    }


    /* Insert the Isotach packet into a bucket */
    bucketize(iso_recv_h, pkt_subtype);

    /* Increment the head pointer */
    iso_recv_h += ISO_RECVFRAME_SIZE;
  }
```

```
    else {
      printf("ISOTACH RECEIVING: did not receive an isotach packet!\n");
      print_out_iso_marker((isochron_marker *)iso_recv_h);
      print_out_iso_packet(&((ISO_RECVFRAME *)iso_recv_h)->packet);
      shutdown(1);
    }
  }
  return SUCCESS;
}
```

netman/net_utils.h

```
/*
 * ---------------------------------------------------------------------------
 *
 * Isotach Module  : Network Utility Functions
 * Isotach Layer   : Netman
 * Isotach Version : Version 3
 *
 * ---------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *        $Source: /home2/isotach/cvsroot/v3/netman/net_utils.h,v $
 *        $Revision: 1.4 $
 *        $Author: pnm2h $
 *        $Date: 1999/11/02 01:25:37 $
 *
 * ---------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This file contains some useful macros which can be used in the Netman
 * module, but which are not applicable to the rest of the messaging layer.
 * These were copied from the previous version of Isotach.  No author was
 * indicated for these macros, so credit will be given to anonymous.
 *
 * ---------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 *
 * ---------------------------------------------------------------------------
 */

#ifndef NET_UTILS_H
#define NET_UTILS_H

#include <hostman/utils.h>

/* The following is strictly for debugging the LANai */
ULONG D_iso_receives = 0;
ULONG D_receives     = 0;
ULONG D_iso_sends    = 0;
ULONG D_sends        = 0;
ULONG D_iso_dma_s    = 0;
ULONG D_iso_dma_e    = 0;
ULONG D_dma_s        = 0;
ULONG D_dma_e        = 0;
ULONG D_error_state  = 0;
ULONG D_mbm_receives = 0;
ULONG D_eop_receives = 0;
ULONG D_chr_receives = 0;
ULONG D_bsmark       = 0;
ULONG D_forced       = 0;
ULONG D_idle         = 0;
ULONG D_dropped_packets = 0;
ULONG D_dropped_words   = 0;

/* End Debugging Variables */

/* This enumerated type represents the different states the receive
   code can be in while dma'ing packets up to the host
*/
enum dma_state {
  IDLE = 0x0, DMA_NONISO, DMA_ISO
};

/* Return the max/min of two given numbers */
#define max(a,b) ((a) >= (b) ? (a) : (b))
#define min(a,b) ((a) < (b) ? (a) : (b))
```

```
/* The following are macros used to access the Lanai's status register */

/*
 * REMINDER (from page 15 of the LANai 4.x docs):
 *
 * after a write to RML:
 *     recvready() is undefined for 2 instructions
 *     recvdone() is undefined for 1 instruction
 *
 * after a write to SML or SMLT:
 *     sendready() is undefined for 2 instructions
 *     senddone() is undefined for 1 instruction
 *
 * after a read from RB, RH, RW:
 *     recvready() is undefined for 3 instructions
 *
 * after a write to SB, SH, SW, ST:
 *     sendready() is undefined for 3 instructions
 *
 * after a write to DMA_CTR:
 *     dmadone() is undefined for 1 instruction
 *
 */

/* often status bits are undefined for a certain number of
   instructions.  The following nop is used to wait until
   status bits are valid.
*/
#define nop() asm volatile ("nop")


//currently not used
#define wait_senddone() \
{ nop(); \
  nop(); \
  while(!(ISR & SEND_INT_BIT)); }

//an inline function taken from the previous version of the code
//that returns the status of a dma transfer out onto the wire
//note that you must have at least 2 instructions between the
//write to SMLT (initiating the transfer), and this check
static inline int senddone(void)
{
  return (ISR & SEND_INT_BIT);
}

//wait after a simple write onto the network (ie SB,SH,SW)
#define wait_sendready() \
{ nop(); \
  nop(); \
  nop(); \
  while(!(ISR & SEND_RDY_BIT)); }

//the check after a simple write onto the network
#define sendready() (ISR & SEND_RDY_BIT)

//a macro that will wait until we are finished a simple receive
//(ie. RB, RH, RW)
#define wait_recvdone() \
{ nop(); \
  nop(); \
  while(!(ISR & RECV_INT_BIT));}


//wait for a receive dma to finish
#define wait_recvbufdone() \
{ nop(); \
  nop(); \
  while(!(ISR & (RECV_INT_BIT | BUFF_INT_BIT)));}

//check to see if there is data waiting to be received
#define recvready() (ISR & BYTE_RDY_BIT)

//check to see if a host dma has finished
```

```
//note that there needs to be at least 2 instructions between
//the write to DMA_CTR (initiating the dma) and this check
#define dmadone() (ISR & DMA_INT_BIT)

//explicitly wait for a host dma to finish
#define wait_dmadone() \
{ nop(); \
  while(!(ISR & DMA_INT_BIT));}


//check to see if we have overrun our receive buffer.  If we have
//something is wrong so just drop the packet
#define buffer_overrun() ((ISR & BUFF_INT_BIT) ? drop() : FALSE)

/*
 * was the message we just received aligned on a word boundary?
 *
 * that is: are both overrun bits set, meaning that there were
 * three overrun bytes since the CRC should have landed on the
 * first byte of the word following the message
 */

static inline int aligned (void) {
  register int orun;
  orun = (ISR & (ORUN1_INT_BIT | ORUN2_INT_BIT));
  return (orun == (ORUN1_INT_BIT | ORUN2_INT_BIT));
}

/*
 * drop an entire packet on the floor
 */

static inline int drop (void) {
  register unsigned char bit_bucket;
  nop();
  nop();
  nop();

  while (!(ISR & TAIL_INT_BIT) && (ISR & BYTE_RDY_BIT)) {
    nop();
    nop();
    nop();

    bit_bucket = RB;
    D_dropped_words++;
  }

  return TRUE;
}

/*
 * Has the token timeout timer expired?
 */
//don't think we use this for anything right now, but could
//be useful for an all software SIU
static inline int token_timeout (void) {
  return (ISR & TIME_INT_BIT);
}


#endif
```

netman/locals.h

```
/*
 * -------------------------------------------------------------------------
 *
 * Isotach Module  : Netman Local Variable Header File
 * Isotach Layer   : Netman
 * Isotach Version : Version 3
 *
 * -------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/netman/locals.h,v $
 *       $Revision: 1.9 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:16 $
 *
 * -------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This file contains all of the local variable and data structure definitions.
 * It also includes all other header files necessary for the module's function.
 * It also prototypes all functions internal to that module.
 *
 * -------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 * -------------------------------------------------------------------------
 */

#ifndef NETMAN_LOCALS_H
#define NETMAN_LOCALS_H

#include <lanai4_def.h>
#include <netman/net_utils.h>


/* -------------------------------------------------------------------------
 * Local Variable that are memory mapped to the host
 * -------------------------------------------------------------------------
 */

volatile UCHAR init_stage;
volatile ULONG netman_hostbase;
volatile ULONG netman_maxlen;
volatile ULONG niu_iso_recv_base;
volatile UCHAR NIU_SYS_TYPE;

/* niu_send_buf is a buffer for outgoing messages that exists on the LANAI's
 * SRAM.  The following variables point to the beginning of the
 * buffer, the head and the tail (respectively).
 */

volatile PACKET niu_send_buf[NIU_SEND_SIZE];
volatile UCHAR niu_send_t;
volatile UCHAR niu_send_h;

/* niu_delivery_q is a buffer for pointers to incoming messages.  It exists
 * on the LANAI's SRAM.  The following variables point to the beginning of
 * the buffer, the head and the tail (respectively)
 */

volatile PACKET_PTR niu_delivery_q[NIU_DELV_SIZE];
volatile ULONG niu_delivery_t;
volatile ULONG niu_delivery_h;

/* Keeps a count of the number of packets received by the LANAI.  The host
   checks this against its own count of how many it has received to determine
   whether or not messages from the LANAI need to be processed.  This is
```

```
   only used during synchronization.
*/
volatile ULONG lanai_received;

/* isotach memory mapped variables */

// functions similarily to the niu_send_buf described above
volatile ISO_SENDFRAME iso_niu_send_buf[ISO_NIU_SEND_SIZE];
volatile UCHAR iso_niu_send_t;
volatile UCHAR iso_niu_send_h;

/* the address of iso_receive_buf in pinned memory*/
volatile ULONG iso_recv_start;
/* the tail pointer into the isotach receive buffer */
volatile ULONG iso_recv_t;
/* the size of the isotach receive buffer */
volatile ULONG iso_recv_size;

/* ----------------------------------------------------------------------
 * Local Variable that are NOT memory mapped to the host
 * ----------------------------------------------------------------------
 */

/* we need to calculate where the next tail pointer could possibly
   be after we DMA an entire isotach receive buffer to determine
   if we will need to wrap around back to the beginning on the
   next transfer.  This variable stores what the tail will be
   once the DMA has finished.
   Please see the Ironman documentation for a full description of
   the Isotach receive path from the LANAI to the host.
*/
volatile ULONG *new_iso_recv_t;

// used during the initialization of the dma engine
ULONG dma_sts;


// the receive buffer on the LANAI for nonisotach packets
// they are queued up here and then dma'd up to the host
// as soon as the engine is free
PACKET niu_receive_buf[NIU_RECV_BUF_SIZE];
PACKET_PTR niu_recv_t;
PACKET_PTR niu_recv_h;

/*isotach receive data structures*/

/* This is the 'high water mark' for filling up the iso_recv_buf.  If the
   current DMA will place any words beyond this address, then a STOP word must
   be placed as the first word following the last DMA'd packet.  This indicates
   to the host, that the next packet will be at the beginning of the
   iso_recv_buf */
ULONG *iso_recv_buf_thresh;

/* These are for receiving from the network into LANAI SRAM
   We have two different buffers that we use for pipelining.
   Thus, when we receive an EOP marker (a buffer fills up), we
   can DMA that buffer up to the host, and still keep receiving
   from the network at the same time by utilizing the second buffer.
 */

// the two aforementioned buffers
ULONG niu_iso_recv_buf0[ISO_NIU_RECV_SIZE + 1];
ULONG niu_iso_recv_buf1[ISO_NIU_RECV_SIZE + 1];

// a threshhold value for each buffer.  Once we hit it, we know that
// we have no more room for isotach packets inside this buffer.
// Thus once we hit this threshold, we dma that buffer, and switch to
// the next.
ULONG *niu_iso_thresh[2];

// this stores the starting address of each of the isotach receive buffers
ULONG *niu_iso_buffer[2];

// the tail pointer into whichever buffer we are currently using
```

```
ULONG *niu_iso_tail_ptr;

// stores which number buffer we are using.  This is used to toggle between
// the two buffers
UCHAR cur_niu_iso_buf;

// this stores the state of the DMA engine
// its potential values are IDLE, DMA_NONISO, DMA_ISO
char state;

// used to store the first word of any received packet so
// that we can determine what its type is
ULONG buffer;

/* Local Function Prototypes */
inline void send_word(ULONG word);
inline void send_packet(UCHAR *data, UCHAR length);
inline void receive_packets();
inline void synchronize();
inline void send_packets();
inline char check_dma();
inline void iso_dma();
inline void noniso_dma();
inline void force_noniso_dma();

#endif
```

netman/netman.c

```
/*
 * ---------------------------------------------------------------------------
 *
 * Isotach Module  : Netman
 * Isotach Layer   : Netman
 * Isotach Version : Version 3
 *
 * ---------------------------------------------------------------------------
 *
 * REVISION INFORMATION:
 *
 *       $Source: /home2/isotach/cvsroot/v3/netman/netman.c,v $
 *       $Revision: 1.11 $
 *       $Author: pnm2h $
 *       $Date: 1999/11/04 22:29:16 $
 *
 * ---------------------------------------------------------------------------
 *
 * DESCRIPTION:
 *
 * This is the main module of the program which gets loaded onto the Myricom
 * Interface board.  The, along with send_side and receieve_side comprise
 * the lanai control program.  The module contains the main function and
 * synchronization.
 *
 * ---------------------------------------------------------------------------
 *
 * COMMENTS:
 *
 *
 * ---------------------------------------------------------------------------
 */

#include <netman/locals.h>

int main() {

  /* ------------------------------------------------------------------------
   * Initialization of the LANAI Begins HERE
   * ------------------------------------------------------------------------
   */

  int i;
  ULONG tmpbuf[8];

  MYRINET = CRC_ENABLE_BIT;
  VERSION = 0;
  TIMEOUT = 3;
  WRITE_ENABLE = 0x80000000;

  /* Spinning until host tells us to continue... */

  while (init_stage != START_LANAI) {}

  /* Section 1: Initialization of a small portion of SRAM which sets value
   *            for DMA test.
   */

  for (i = 0; i < 8; i++) {
    tmpbuf[i] = 0x11223344;
  }

  /* Section 2: DMA from SRAM to pinned memory. */

  DMA_STS = dma_sts;
  DMA_DIR = 0;
  LAR = (void *) tmpbuf;
  EAR = (void *) netman_hostbase;
  DMA_CTR = 8 * sizeof(int);
  wait_dmadone();
```

```
init_stage = INIT_DMA_TEST;

/* Spinning until host has initialized */

while (init_stage != CHECK_DMA_TEST) {}

/* Section 3: Initialize all necessary variables in SRAM */

/* Initialize Lanai Buffers, Head, & Tail Pointers*/
niu_send_t     = 0;
niu_send_h     = 0;
niu_delivery_h = 0;
niu_delivery_t = 0;

iso_niu_send_t = 0;
iso_niu_send_h = 0;


niu_recv_t = niu_receive_buf;
niu_recv_h = niu_receive_buf;


cur_niu_iso_buf = 0;

niu_iso_buffer[0] = niu_iso_recv_buf0;
niu_iso_buffer[1] = niu_iso_recv_buf1;

niu_iso_thresh[0] = (ULONG *)((niu_iso_buffer[0] + ISO_NIU_RECV_SIZE) -
                              (sizeof(eop_marker) / 4));
niu_iso_thresh[1] = (ULONG *)((niu_iso_buffer[1] + ISO_NIU_RECV_SIZE) -
                              (sizeof(eop_marker) / 4));

niu_iso_tail_ptr = niu_iso_buffer[cur_niu_iso_buf];

// we currently aren't dma'ing anything up to the host
state = IDLE;


/* Spinning until host maps additional variables and reads in configuration
 * file
 */
while (init_stage != SYNCHRONIZE) {}


/* Section 4: Enter synchronization routine */
synchronize();


/* kind of a kludge here...  if we have an isotach system, the host is
   finished once the isotach receive buffers are mapped.  If its a
   nonisotach system, it goes to a finished stage.
*/
if (NIU_SYS_TYPE != NONISOTACH) {
  while ((iso_recv_start == 0) || (iso_recv_size == 0));

  iso_recv_buf_thresh = (ULONG *)((iso_recv_start + iso_recv_size) - ISO_NIU_RECV_SIZE);
}
else {
    while (init_stage != FINISHED) {}
}


/* ------------------------------------------------------------------------
 * Initialization COMPLETED.  The next section of the main function is the
 * main event loop
 * ------------------------------------------------------------------------
 */

while(1) {
  // one of our main design goals is to keep the dma engine as
  // busy as possible.  Thus, every pass through this loop we
  // check to see if the engine is free and if so, dma up any
  // nonisotach packets that are sitting around
  if (state != IDLE)
```

```
      check_dma();
    if (!queue_empty(niu_recv_h,niu_recv_t))
      noniso_dma();

    /* call receive_packets */
    receive_packets();

    /* call send_packets */
    send_packets();
  }

  return 0;
}

// a debugging routine that we use to send a single word of
// data to a separate host on the switch that is running our
// send/receive program.  Kind of our version of printf() debugging
inline void send_word(ULONG word) {
  /* change this before recompiling to the debug host route */
  char route = 0xBE;

  wait_sendready();
  SB = route;
  wait_sendready();
  SW = word;
  wait_sendready();
  ST = 0;
  wait_sendready();
}

// a debugging routine that we use to send a block of
// data to a separate host on the switch that is running our
// send/receive program.  Kind of our version of printf() debugging
inline void send_packet(UCHAR *data, UCHAR length) {
  int i;
  wait_sendready();
  SB = 0x83;
  wait_sendready();

  for (i=0; i< length; i++) {
    SB = data[i];
    wait_sendready();
  }
  ST = 0;
  wait_sendready();
}


/* ***************************************************************************
 * Receiving Support Functions
 * ***************************************************************************
 */


// this function checks the status of the dma engine and returns
// whatever it was previously doing.
// it also performs all necessary operations once a particular
// type of dma has finished
inline char check_dma() {
  char prev_state = state;

  // if the dma engine has finished sending
  if (dmadone()) {
    // if we were dma'ing a nonisotach packet
    if (state == DMA_NONISO) {
      // enqueue a pointer onto the niu_delivery queue
      niu_delivery_q[niu_delivery_t] = (PACKET_PTR)niu_recv_h->address;

      // increment the tail pointer of the deliverq queue
      inc_idx(niu_delivery_t, NIU_DELV_SIZE);

      // advance the head pointer of the niu receive buffer
      inc_ptr(niu_recv_h, NIU_RECV_BUF_SIZE, niu_receive_buf);
```

```
      // set our state to idle
      state = IDLE;

      //D_dma_e++;
    }
    // if we were dma'ing the isotach receive buffer
    else if (state == DMA_ISO) {
      // our new tail pointer is what we calculated previously
      iso_recv_t = (ULONG)new_iso_recv_t;

      // set our state to idle
      state = IDLE;

      //D_iso_dma_e++;
    }
  }
  return prev_state;
}

// a function that tries to dma a nonisotach packet up to the host
inline void noniso_dma() {
  // if the engine is free
  if (state == IDLE) {
    // we are taking it off the head of the recieve buffer
    LAR = (void *)(niu_recv_h);

    // and putting it wherever the packet says to
    EAR = (void *)(niu_recv_h->address);

    // we need to be careful though...
    // if the destination address is out of bounds, we bail and sit
    // in an infinte loop.  Otherwise, we could potentially overwrite
    // part of the kernel or some other important portion of memory
    // and hang the machine.
    if ((niu_recv_h->address >
           (netman_hostbase + netman_maxlen - niu_recv_h->pad2 - 100))
          || (niu_recv_h->address < (netman_hostbase))) {
      D_error_state = 1;
      while(1);
    }

    /* The size of the packet was stored in the 2nd pad*/
    DMA_CTR = niu_recv_h->pad2;

    // set our state to reflect the fact that we are transfering a
    // nonisotach packet
    state = DMA_NONISO;

    //D_dma_s++;
  }
  return;
}

// sometimes, if our receive buffer is full, we need to free up a
// slot before we can receive off of the network.  The previous
// function only dma's if the engine is idle.
// this function enforces that a dma transfer occurs
inline void force_noniso_dma() {

  // wait for the dma engine to finish what it was doing
  wait_dmadone();

  // if we were previously dma'ing an isotach buffer, we need
  // to force a noniso dma.
  if (check_dma() == DMA_ISO) {
    noniso_dma();
    wait_dmadone();
    check_dma();
  }

  // otherwise, we know that we've finished a nonisotach dma and
  // a slot has freed up.  So lets start another and return.
  noniso_dma();
  return;
```

```
}

// this function dma's an entire isotach receive buffer
// it is forced in the sense that we need to start the
// transfer now.  It can finish whenever.
// It also determines whether we need to wrap around in the
// hosts isotach receive buffer on the next dma, as well as
// switches the niu isotach receive buffer so we can keep
// receiving isotach packets
// for a detailed description of the isotach receive path, consult
// the ironman design document
inline void iso_dma() {

  // if the engine isn't free, wait for it to finish
  if (state != IDLE) {
    wait_dmadone();
    check_dma();
  }

  /* Calculate where the new iso_recv_t would be after the transfer */
  new_iso_recv_t = (ULONG *)iso_recv_t + (niu_iso_tail_ptr -
                                          niu_iso_buffer[cur_niu_iso_buf]);


  /* If another niu_iso_recv_buf will not fit in the iso_recv_buf, then
     reset the tail pointer to the beginning of the iso_recv_buf */
  if (new_iso_recv_t >= iso_recv_buf_thresh) {
    // write STOP_PACKET into the last word of the buffer before
    // beginning the dma.  This will tell the host to wrap around
    *niu_iso_tail_ptr = STOP_PACKET;
    niu_iso_tail_ptr++;

    // set the new tail pointer to the beginning of the buffer
    new_iso_recv_t = (ULONG *)iso_recv_start;
  }

  // we are transfering data from the current buffer
  LAR = (void *)niu_iso_buffer[cur_niu_iso_buf];

  // our destination is the tail pointer into the host's isotach
  // receive buffer
  EAR = (void *)iso_recv_t;

  // the number of bytes to transfer is the tail - the start of the niu
  // isotach recieve buffer
  DMA_CTR = (void *)niu_iso_tail_ptr - LAR;

  // set our state accordingly
  state = DMA_ISO;

  // now toggle niu isotach receive buffers and set the new tail
  cur_niu_iso_buf = (cur_niu_iso_buf + 1) % 2;
  niu_iso_tail_ptr = niu_iso_buffer[cur_niu_iso_buf];
  return;
}


// this function receives packets off of the network
inline void receive_packets() {
  int receives = 0;
  USHORT iso_pkt_size; //can't put size into pad2 of an isotach packet

  /* while there is something to receive... */
  /* we also want to set a cap on the number of packets received in
     a single function call, since receiving has the potential of
     starving sending */
  while ((recvready()) && (receives < RECV_LIMIT)) {
    // always check the status of the dma engine
    check_dma();

    /* Reads word off to determine type before DMA */
    buffer = RW;
    wait_recvdone();
```

```
    /* Check the header to see if it is a noniso packet */
    if ((USHORT)(buffer >> 16) == NONISO) {

      // if the receive buffer is full, force a noniso dma
      if (lanai_queue_full((ULONG)niu_recv_h, (ULONG)niu_recv_t,
                             NIU_RECV_BUF_SIZE*sizeof(PACKET))) {
          //D_forced++;
          force_noniso_dma();
      }

      // write the first word of the packet into the receive buffer
      *(ULONG *)niu_recv_t = buffer;

      // dma the rest of the packet off of the network
      RMP = (void *)niu_recv_t + 4;
      RML = (void *)niu_recv_t + sizeof(PACKET) - 4;
      wait_recvbufdone();
      receives++;
      D_receives++;

      // calculate the size of the packet and store it in the pad2 field
      niu_recv_t->pad2 = (USHORT)((void *)RMP - (void *)niu_recv_t);

      // increment our tail pointer
      inc_ptr(niu_recv_t, NIU_RECV_BUF_SIZE, niu_receive_buf);
      receives++;

      // try to dma the packet up to the host
      noniso_dma();
    }
    /* Otherwise it is an Isotach Packet */
    else {
      // write the first word into the isotach receive buffer
      *niu_iso_tail_ptr = buffer;

      // we need to check the packet type at this point
      // some of the hardware SIU's send back garbage before they are reset
      if ((((PACKET *)niu_iso_tail_ptr)->type != EOP) && (((PACKET *)niu_iso_tail_ptr)->type !=
ISO) ) {
          drop();
          D_dropped_packets++;
          continue;
      }

      // dma the rest of the packet into the niu isotach receive buffer
      RMP = (void *)niu_iso_tail_ptr + 4;
      RML = (void *)niu_iso_tail_ptr + sizeof(eop_marker);

      // wait for the dma to finish
      wait_recvbufdone();
      receives++;
      //D_iso_receives++;

      // calculate the size of the packet
      iso_pkt_size = (USHORT)((void *)RMP - (void *)niu_iso_tail_ptr);


      // if we are supporting a hardware SIU...
#if (SIU == 1)
      /* Check to see if it is an isochron_marker */
      if (iso_pkt_size == sizeof(isochron_marker)) {

          // or the isochron crc in with the packets crc
          ((isochron_marker *)niu_iso_tail_ptr)->crc |=
            ((isochron_marker *)niu_iso_tail_ptr)->subtype;

          // set the subtype for identification purposes on the host
          ((isochron_marker *)niu_iso_tail_ptr)->subtype = ISO_MARKER;

          // increment the tail pointer
          niu_iso_tail_ptr += ISO_MARKER_SIZE;

          //D_chr_receives++;
      }
```

```
      /* Is the incoming packet an EOP Marker. */
      else if (((PACKET *)niu_iso_tail_ptr)->type == EOP) {
          /* Use the 1st byte after  the type (sequence number) to
             store the subtype for ID purposes on the host */
          ((eop_marker *)niu_iso_tail_ptr)->subtype = EOP_MARKER;

          // find the crc byte and store it in the crc field in the structure
          ((eop_marker *)niu_iso_tail_ptr)->crc =
            ((ULONG)*((UCHAR *)RMP - 4) & FIRST_BYTE) >> 24;

          // increment the tail pointer
          niu_iso_tail_ptr += EOP_MARKER_SIZE;

          //D_eop_receives++;
      }
      /* otherwise it was a plain isotach packet */
      else {
          // increment the tail pointer
          niu_iso_tail_ptr += ISO_RECVFRAME_SIZE;

          //D_mbm_receives++;
      }

      // if the current isotach receive buffer has grown too large
      // or the packet just received was an EOP, dma the buffer up
      // to the host
      if ((niu_iso_tail_ptr >= niu_iso_thresh[cur_niu_iso_buf]) ||
            (((buffer & (0xFFFF0000)) >> 16) == EOP)) {
          iso_dma();
      }

      // if we are not supporting a hardware SIU, we only receive isotach
      // packets
#else
      // increment the tail pointer
      niu_iso_tail_ptr += ISO_RECVFRAME_SIZE;

      //D_mbm_receives++;
      // dma the packet up to the host
      iso_dma();
#endif
    } // this ends the isotach packet handling

    // regardless of what we received, check the dma engine and
    // try to dma any outstanding nonisotach packets
    if (state != IDLE)
      check_dma();
    if (!queue_empty(niu_recv_h,niu_recv_t))
      noniso_dma();
  }
}

/* ***********************************************************************
 * End Receiving Support Functions
 * ***********************************************************************
 */


/* ************************************************************************
 * Sending Support Functions
 * ************************************************************************
 */

// sends packets out one at a time
/* Currently, send_packets sends out one packet from the niu_send_buf and
 * one packet from niu_iso_send_buf.
 * It may be better to record the number of packets in niu_send_buf upon
 * entering send, and then send this number of packets out before giving
 * control back to main
 */
// this is debatable, since we need to bias the system heavily towards
// receiving
inline void send_packets() {
```

```
  // if we are supporting a hardware siu
#if (SIU==1)
  // keep track of the number of system resets
  static int reset_count = 0;
#endif

  // if we have a nonisotach packet to send
  if (!queue_empty(niu_send_h, niu_send_t)) {
    /* Send out the Routing Bytes */

    wait_sendready();

    /*examining the route from left to right, we have 4 possibilities:
      1 byte route -> 000000R1
      2 byte route -> 0000R1R2
      3 byte route -> 00R1R2R3
      4 byte route -> R1R2R3R4
      where the byte ordering is the order in which they are specified
      in the network configuration file.
    */

    /* If the third byte in the routing word is 0, then we only need
       to send out a 1 byte route, which is done by casting the ULONG
       to a single byte.*/
    if ((niu_send_buf[niu_send_h].route & THIRD_BYTE) == 0)
      SB = niu_send_buf[niu_send_h].route;

    /* If the second byte in the routing word is 0, then we only need
       to send out a half-word route which is done by casting the ULONG
       to a half word.*/
    else if ((niu_send_buf[niu_send_h].route & SECOND_BYTE) == 0)
      SH = niu_send_buf[niu_send_h].route;

    /* If the first byte in the routing word is 0, then we need to send out
       a single byte followed by a half word. */
    else if ((niu_send_buf[niu_send_h].route & FIRST_BYTE) == 0) {
      SB = (niu_send_buf[niu_send_h].route & SECOND_BYTE) >> 16;
      wait_sendready();
      SH = (niu_send_buf[niu_send_h].route & (THIRD_BYTE | FOURTH_BYTE));
    }
    /* Finally, if all 4 bytes are used, send out a 4 byte route with SW */
    else
      SW = niu_send_buf[niu_send_h].route;


    /* Send out the actual packet, up to HEADER_SIZE + payload_length */
    wait_sendready();

    SMP = (void *)&niu_send_buf[niu_send_h];

    /* smlt needs to point to the beginning of the last word of the packet.
       however, if our data is not word aligned, since the last 2 bits of
       smlt are hardwired to zero, it is possible that we could lose data.
       Therefore, we add 3 bytes to our payload length to guarantee that a
       partial word will go out, but then we need to subtract 4 to make it
       point to the beginning of the last word.  Thus, we ultimately
       subtract 1 byte.
    */

    SMLT = (char *)&niu_send_buf[niu_send_h] + PACKET_HEADER_SIZE +
      niu_send_buf[niu_send_h].payload_length - 1;

    // while we are waiting for the packet to go out, try to receive
    // some more packets
    while (!senddone()) {
      receive_packets();
    }

    D_sends++;
    /* Increment head pointer */
    inc_idx(niu_send_h, NIU_SEND_SIZE);
  }
```

```
  /* send out an isotach packet if we can */
  if (!queue_empty(iso_niu_send_h, iso_niu_send_t)) {
    wait_sendready();

    /* if it is a BS marker, we don't use the DMA engine since we are
       only sending out three bytes.  We only want to send them out
       if we are supporting a hardware SIU
    */

    if (iso_niu_send_buf[iso_niu_send_h].prefix & BS_MARKER_MASK) {
#if (SIU==1)
      SB = (iso_niu_send_buf[iso_niu_send_h].prefix & FIRST_BYTE) >> 24;
      wait_sendready();
      SH = (iso_niu_send_buf[iso_niu_send_h].prefix & (SECOND_BYTE |
                                                        THIRD_BYTE)) >> 8;
      wait_sendready();
      ST = 0;
      wait_sendready();
      reset_count++;
#endif
    }

    /* otherwise, send the packet */
    else {
      /* first, send the prefix.  The prefix was switched on the host
           before it was shipped down to the lanai, so we can simply send
           the whole word.  We only send it if we are supporting a
           hardware SIU.
      */
#if (SIU==1)
      SW = iso_niu_send_buf[iso_niu_send_h].prefix;
      wait_sendready();
#endif

      /* Second send the route.  Unfortunately, the SIU expects the route
           to range from 2 - 6 bytes, depending on the number of routing
           bytes.  Since nonisotach supports at most 4 routing bytes, we are
           limiting isotach to that as well.  As before in nonisotach, we have
           4 cases, however, the route is split across two different fields
           in the ISO_SENDFRAME structure:

           1 byte route:  route1 -> 00R1
                          route2 -> 00000000

           2 byte route:  route1 -> R1R2
                          route2 -> 00000000

           3 byte route:  route1 -> 0000
                          route2 -> 00R1R2R3
           4 byte route:  route1 -> 0000
                          route2 -> R1R2R3R4

           Thus, the algorithm to send out the route is as follows:
      */

      /* always send the half word route1 */

      // quick note...  We have conditional compilation designed to
      // specify whether we are supporting a hardware SIU.
      // if we are not, we only support 1 byte routes for isotach packets

#if (SIU==1)
      SH = iso_niu_send_buf[iso_niu_send_h].route1;
#else
      SB = iso_niu_send_buf[iso_niu_send_h].route1;
#endif

      wait_sendready();
      /* if those routing bytes were all zeros, then send out the full
           word route2, which contains the rest of the route
      */
      if (iso_niu_send_buf[iso_niu_send_h].route1 == 0) {
          wait_sendready();
          SW = iso_niu_send_buf[iso_niu_send_h].route2;
```

```
      }

      wait_sendready();

      /* now send the rest of the packet onto the wire using the DMA engine */
      SMP = (void *)&iso_niu_send_buf[iso_niu_send_h].packet;

      /* SMLT points to the last word that needs to be sent out*/
      SMLT = (void *)&iso_niu_send_buf[iso_niu_send_h].packet + sizeof(ISO_PACKET) - 4;

      // while we are waiting for the send to finish, receive what we can
      while (!senddone()) {
        receive_packets();
      }
    }

    //increment the head pointer
    inc_idx(iso_niu_send_h, ISO_NIU_SEND_SIZE);

    //D_iso_sends++;
  }

  return;
}

/* ************************************************************************
 * End Sending Support Functions
 * ************************************************************************
 */


// this is the function used to synchronize all of the hosts
// all it primarily does is send and receive packets
inline void synchronize() {
  UCHAR crc;

  int receives = 0;
  int sends    = 0;

  int sending   = FALSE;
  int receiving = FALSE;

  /* In order to support a flood of synchronization packets that can
     occur when synchronizing many hosts, we utilize a pipelining
     scheme in which a packet can be received from the network
     while the previous one is being dma'd up to the host */

  // array of two packets
  PACKET packet[2];

  // indices into that array
  int from_net = 0;
  int to_host  = 0;

  // pointer into pinned memory
  void *next_slot;

  // number of packets received by the lanai
  lanai_received = 0;

  /* Initialize the beginining of the DMA area to the beginning of host's
     pinned memory. */
  next_slot = (void *)netman_hostbase;


  /* Send and receive until the host says it has finished synchronization */
  while (init_stage != FINISHED) {

    /* If there is a Byte to read off the network, then receive it. */
    if (recvready()) {
      // dma the packet into the free packet slot
      RMP = &packet[from_net];
      RML = (char *)&packet[from_net] + sizeof(PACKET) + 4;
      wait_recvbufdone();
```

```
  // if the packet was ok....
  if (!buffer_overrun()) {
      /* Check the CRC */
      /* Sync and Sync Ack packets are always word aligned, so
         CRC byte is the first byte in the first word past the packet */

      crc = *((UCHAR *)RMP - 4);
      packet[from_net].pad2 = (USHORT)((void *)RMP -
                                       (void *)&packet[from_net]);

      /* If there is a bad CRC, notify the host and return */

      // we've noticed that sometimes we receive truncated packets
      // during synchronization if we have too many hosts trying to
      // synchronize at once, even with our pipelining scheme
      // thus we drop these packets and contine
      if ((packet[from_net].subtype == SYNC) &&
          (packet[from_net].pad2 < PACKET_HEADER_SIZE)) {
        continue;
      }

      // otherwise there was an actual CRC error
      if (crc) {
        LAR = (void *)&packet[from_net];
        EAR = (void *)netman_hostbase;
        DMA_CTR = (packet[from_net].payload_length + PACKET_HEADER_SIZE);
        wait_dmadone();
        init_stage = BAD_CRC;
        return;
      }

      /* Move the from_net pointer to the next slot, so the NIU interface can
         receive another packet */
      from_net = (from_net + 1) % 2;
      receives++;

      /* DMA the newly received packet into host's pinned memory */
      /* Only start a new DMA to the host if the previous one has finished */
      wait_dmadone();

      /* If we have received a packet last iteration, move the receiving
         slot and set Receiving to false. */
      if (receiving == TRUE) {
        /* Increment the 'tail pointer' on the Pinned Memory Queue to
           point to the next available slot for DMA. */
        next_slot += sizeof(PACKET);
        // toggle the to_host pointer
        to_host    = (to_host + 1) % 2;
        receiving  = FALSE;
        lanai_received++;
      }

      // set the destination
      EAR = next_slot;

      // we are dma'ing the previously received packet
      LAR = (void *)&packet[to_host];

      // here is a check to ensure that we are not dma'ing into
      // regions of memory that we shouldn't.  This check should
      // be able to be removed, but...
      if ((ULONG)next_slot > (netman_hostbase + netman_maxlen -
                              2 * sizeof(PACKET))) {
        while(1);
      }

      DMA_CTR    = (packet[to_host].payload_length + PACKET_HEADER_SIZE);
      receiving = TRUE;
  }
}

nop();
if ((receiving == TRUE) && (dmadone())) {
```

```
      /* Increment the 'tail pointer' on the Pinned Memory Queue to
           point to the next available slot for DMA. */
      next_slot += sizeof(PACKET);
      to_host    = (to_host + 1) % 2;
      receiving  = FALSE;
      lanai_received++;
    }

    nop();
    nop();
    nop();

    if ((sending == TRUE) && (sendready())) {
      sending = FALSE;
      /* Increment head pointer */
      inc_idx(niu_send_h, NIU_SEND_SIZE);
    }

    if ((!queue_empty(niu_send_h, niu_send_t)) && (sending == FALSE)) {
      /* Send out the Routing Bytes */

      /* See the notes in send_packets concerning sending out of routing
           bytes */
      if ((niu_send_buf[niu_send_h].route & SECOND_BYTE) == 0)
          SB = niu_send_buf[niu_send_h].route;
      else if ((niu_send_buf[niu_send_h].route & THIRD_BYTE) == 0)
          SH = niu_send_buf[niu_send_h].route;
      else if ((niu_send_buf[niu_send_h].route & FOURTH_BYTE) == 0) {
          SH = niu_send_buf[niu_send_h].route;
          wait_sendready();
          SB = *((char *)(&niu_send_buf[niu_send_h].route) + 2);
      }
      else
          SW = niu_send_buf[niu_send_h].route;

      /* Send out the actual packet, up to HEADER_SIZE + payload_length */
      wait_sendready();
      SMP = (void *)&niu_send_buf[niu_send_h];

      /* Since Sync packets are word aligned, we don't have to worry
           about any extraneous bytes in the "next" word */
      SMLT = (char *)&niu_send_buf[niu_send_h] + PACKET_HEADER_SIZE +
niu_send_buf[niu_send_h].payload_length - 4;
      sends++;
      sending = TRUE;
    }
  }

  /* reset the Lanai Send Buffer queue */
  niu_send_t = niu_send_h = 0;

  lanai_received = 0;

  return;
}
```