Testing the Isotach Prototype Hardware Switch Interface Unit

Doug Szajda

June 4, 1999

Contents

1 Introduction

Although Isotach systems can be implemented entirely in software using commercial off-the-shelf hardware, they achieve greatest efficiency with the help of special hardware components, one of which is the switch interface unit (SIU). Located in-link between the network interface boards and switches to which these boards are connected, the SIU is responsible for the maintenance of logical time, for timestamping outgoing isotach messages, and for exchanging tokens with another custom hardware device, the token manager (TM).

Testing the SIU requires the modification of the isotach prototype code base to create two new system versions, Version 1.75 which emulates the functionality of the SIU and is intended to run in systems in which other nodes are connected to hardware SIUs, and Version 2 which runs on hosts connected directly to hardware SIUs. In addition to these prototypes, a suite of test programs has been designed to aid in lower level debugging of the device, and to verify that design requirements have been met.

This paper discusses the development of these prototype versions, the current state of the SIU debugging process, and the design of the test suite. In addition, five appendices are included as a guide for those who will continue the SIU testing efforts. These discuss various implementation specific data structures and routines, offer tips for programming the LANai, provide pointers to the code for the various prototype versions, give values for some implementation specific constants, and provide pointers to additional information.

1.1 Isotach Systems

This section provides a brief introduction to the theory of Isotach systems, and describes the various versions of the system prototype implemented by the Isotach Group at the University of Virginia. This description is included merely for the sake of completeness, as it is assumed that the reader has a working knowledge of both Isotach networks and of the high level design of the prototype. Detailed discussion of these issues may be found in [6], [5], [8], and [9].

Isotach systems are parallel and/or distributed systems designed to provide synchronization mechanisms with substantially reduced overhead. This is accomplished by providing strong guarantees regarding message delivery order [6] [1]. These guarantees provide sequential consistency and atomicity over the operations performed by parallel programs without the need for more expensive traditional synchronization methods such as locks and barriers.

Isotach systems achieve event ordering through an extension of Lamport's [3] notion of logical time: Isotach logical time is represented as an ordered n-tuple of nonnegative integers. Message logical delivery times satisfy the *isotach invariant*, which states that a message sent at time (i, j, k) will be delivered at time $(i + \delta, j, k)$, where δ is the logical distance between the source and destination. Since this invariant guarantees that messages travel through the network at a constant logical rate, a process can control the receive time of a message by knowing the logical distances to destinations, and controlling when messages are emitted into the network. Logical time is advanced, and the network kept loosely synchronized (in physical time), through a token passing mechanism.

1.2 Prototype Design

An initial prototype system (dubbed V1 and described in [5]) has been built under a DARPA contract by the Isotach Group at the University of Virginia. In this version of the prototype, the three principle system components, the switch interface unit (SIU), token manager (TM), and shared memory manager (SMM) modules, are implemented in software using a Myrinet [2] network running Illinois Fast Messages v1.1 [4], and 180 and 200 MHz Pentium Pro and 166 MHz Pentium machines running Red Hat Linux. In V1, the SIU code is split between the processors on the Myrinet network interface boards (*LANais*) and the corresponding hosts, the TM code runs on LANai boards attached to hosts whose only purpose is to support the TMs, and the SMM code runs on each (non TM) host. In the V2 prototype, presently under construction, the SIUs and TMs are special hardware devices, with the SIUs located in-link between the network interface boards and the switches to which these boards are connected, and the TMs attached directly to a port on each network switch.

Two intermediate prototype versions, V1.5 and V1.75, have also been completed. These are intended primarily as test vehicles, and employ various combinations of software and custom hardware components. Specifically, V1.5 code is capable of performing with or without hardware TMs in the system. The V1.5 TM code emulates the functionality of a hardware TM to such a degree that communication with this module is indistinguishable (aside from the expected performance decrease) from communication with a hardware TM. Similarly, the V1.75 prototype is designed to perform correctly with a hardware SIU in the system, and to exactly emulate (up to performance speeds) the functionality and communication protocols of a hardware SIU. The advantages of these intermediate versions should be clear—provided the code itself is correctly implemented, a hardware device can be tested by observing its interactions with the corresponding software module.

The V2 code base is designed to run on the host and LANai processors of a node that is directly attached to a hardware SIU. Specifically, V2 code performs none of the SIU functionality required (to various degrees) of V1, V1.5, or V1.75. For this reason, testing and debugging of much of the V2 specific code has been concurrent with testing of the hardware SIU.

2 Modifying code to create V1.75/V2

Several changes were required to the V1.5 code base to allow interaction with hardware SIUs. In order to appreciate these changes, a short description of V1 and it's evolution to V1.5 are in order.

The V1 prototype was designed to provide (relatively quickly) a working prototype with which one could validate hardware specifications, pre-test some system software components (some of the software components that would remain even in V2), and as a platform for application development. As such it provides the same API as the V2 prototype, and code that *simulates* the functionality of the custom hardware TMs and SIUs. V1.5, on the other hand, provided the same simulation of overall functionality, but needed to *emulate* the functionality of the hardware token managers. The difference lies in the communication mechanisms. In V1, messages containing token information could be passed between nodes, but the specific form and content of these messages was entirely up to the implementer. In

V1.5, however, that information had to be exchanged with the exact protocols and packet formats required by the isotach hardware design specifications. The change from V1 to V1.5 then involved changes to data structures and packet formats in order to guarantee that communication with a software token manager was (almost) indistinguishable from communication with the hardware device. The V1.75 prototype took this a step further. Here the requirement was that interaction with a node running V1.75 looked exactly like communication with a node connected directly to a hardware SIU. Although this might appear to be nothing more than a cosmetic change to packets, because of the use of the Fast Messages messaging software, and the need for isotach specific information at levels both "above" and "below" the FM layer, the changes here were somewhat involved.

The changes required to V2 were the most extensive. To begin with, all of the SIU functionality had to be removed from both the LANai and host code. And where the goal of V1.75 is to present an accurate interface to the network, V2 needs to present an accurate interface to the SIU. This means, for example, that information that can be passed "internally" (i.e. from host to NIU or from host to itself) in the earlier prototype versions now has to be passed from network interface to the in-link SIU, and possibly back again. Thus new packet types such as isochron markers, EOP markers, and barrier/signal markers need to be both generated by and accommodated by the V2 code. Moreover, tokens, which are of fixed length in the earlier prototypes, have varying lengths in V2 because of the insertion of sort vectors by the SIU. Finally, the message sorting mechanism needed to be changed to accommodate the possibility that messages could be sorted entirely on the SIU, entirely in the host, or partially in each and then merged on the host.

2.1 Packet Formats

The use of the Fast Messages software package served to aid in the relatively quick development of the V1 prototype. Although even in this domain significant modifications to the FM source code were required to handle certain isotach specific primitives, for the most part one could encapsulate isotach messages, and particularly the struct net_sref, in the payload of an FM packet and allow FM to handle issues arising from the movement of the packet through the network (these issues include flow control, packet lengths, and routing, among others). Unfortunately, the packet formats required by the hardware SIU specifications precluded the use of this mechanism as it stood. This was unavoidable—the hardware SIUs could not possibly be placed "above" the FM layer (part of which runs on the host), and they need to be able to quickly distinguish between isotach and non-isotach traffic. One possible solution would have been to have the SIU store and forward all packets traveling to the network, examining the payload, and releasing non-isotach packets as quickly as possible. This would have unnecessarily delayed non-isotach traffic as well as tied the prototype to the FM messaging layer (the SIU would have needed information on where the FM payload resides in a packet, for example). The bottom line is that the best place for the isotach specific information, in terms of the SIU, is at the front of a packet, and this is exactly what the hardware specification mandates. From the standpoint of the V1.75 (or V2) developer, then, the only possible choices are scrapping the messaging layer, and with it the services provided by FM, or accepting a less than ideal layering model with isotach sitting both above and below the FM layer. In the interest of time (and with the knowledge that the prototype would likely be completely redesigned in the near future) I opted for the latter.

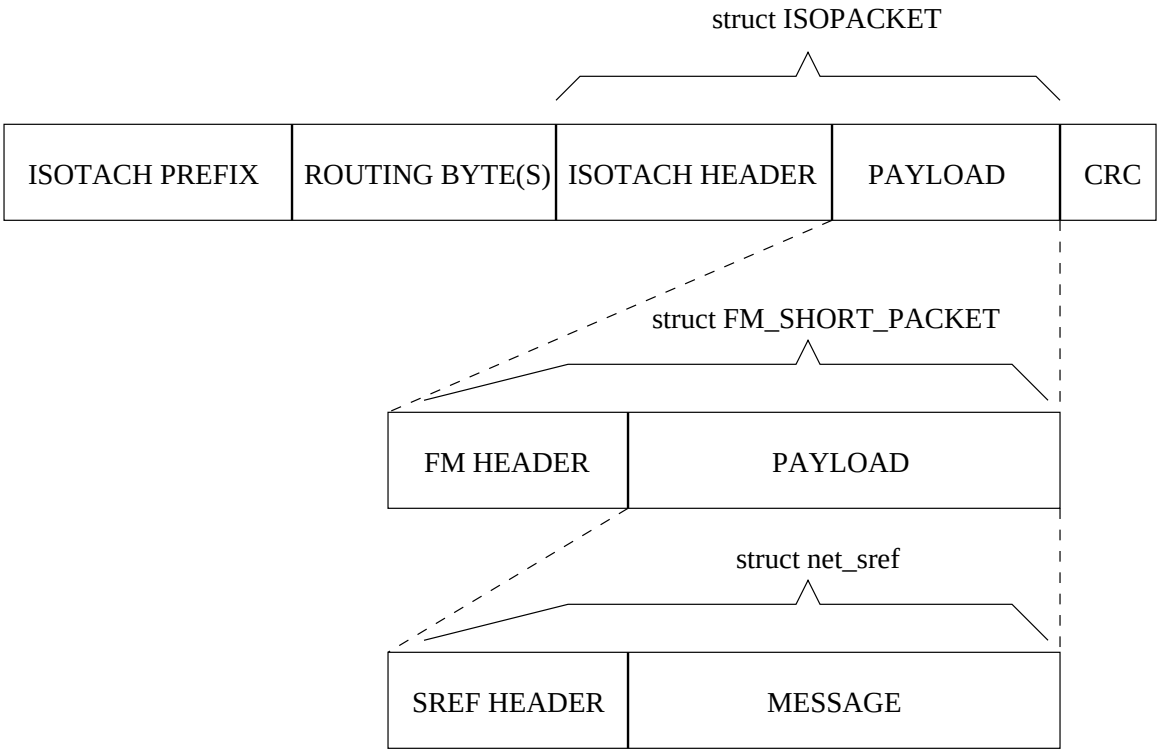

Figure 1: Structure of a V2 Isotach Packet

In both V1.75 and V2, the isotach packet passing mechanism works as follows. When an sref is generated for execution on a remote node, the relevant information is placed into a struct net_sref as in the earlier prototypes, which in turn is placed into the payload field of a modified FM packet (see Figure 1). The modified FM packet, a struct FM_SHORT_PACKET, is as it's name implies shorter than a regular FM packet, with the difference in length stemming entirely from reducing the size of the payload field by 18 bytes. Aside from the reduced payload field, an FM_SHORT_PACKET is indistinguishable from a regular FM packet. This modified FM packet is then placed in the payload field of a struct ISOPACKET, which itself consists of the isotach header fields followed by the FM_SHORT_PACKET payload. This entire assembly was designed so that the size of an ISOPACKET is identical to that of a regular FM packet (so that changes to LANai queue management mechanisms was minimal), while at the same time the FM packet header information retains the same format as in the unmodified FM packet. In addition, both FM packets and ISOPACKETS have leading two byte packet type fields. In V1.75, when the network interface prepares to send an ISOPACKET, it prepends the appropriate routing bytes. In V2, the isotach prefix is in turn prepended to these routing bytes. What ends up at the receiving node in either version (and with either FM or ISOPACKETS) is a packet of a fixed length, with a leading packet type field. If the packet type indicates an FM packet, then the data is DMAed "as is" to the host level FM receive queue. If, however, the packet type indicates an ISOPACKET, the network interface code removes the isotach specific information from the isotach header, loads it into the corresponding fields of the

struct net_sref in the payload of the embedded FM_SHORT_PACKET, and then DMAs the FM_SHORT_PACKET into the FM receive queue on the host. The host level FM code requires no modification because the usual FM headers have been preserved. This mechanism, while somewhat inelegant, leveraged heavily the existing network interface and host level code, and thus allowed for a relatively rapid V2 development.

A final packet format modification specific to V2 involved developing network interface code capable of handling token packets (technically end-of-pulse (EOP) markers) of various size, a necessity in this version because of the generation of variable length "sort vectors" by the H/W SIU.

2.2 V2 Sorting Mechanisms

The other significant code modification involved changing the V1.5 sorting mechanisms to accommodate the variety of "sort type" possibilities unique to V2. In the earlier prototype versions, all sorting of srefs in pulse buckets are performed completely in software. The addition of the H/W SIU, however, along with the above mentioned generation of sort vectors, means that in many instances there is no need for a software sort. Rather, the sort vector is passed to the host level code responsible for "draining" the pulse buckets and the srefs are executed in the order specified by this vector. Although this is the expected mode of operation in the vast majority of EOP/drain bucket "cycles", there are nevertheless situations in which the V2 prototype will be expected to execute either an all software sort, or a hybrid sort. The former is required during host mode operation when the first "in sequence" token the host receives for a particular pulse has the repeat bit set. This will occur only if a previous copy of the same token was transfered by the SIU, yet not received by the host. In this case, the SIU sets the repeat bit in the corresponding EOP marker, warning the host that an all software sort may be required. A hybrid sort is required when there are more srefs scheduled to be executed during a logical time pulse than there are slots in a sort vector (in our present prototype, the maximum number of entries that can be handled in a single sort vector is 32). When this occurs, the SIU places the correct number of srefs to be executed in the corresponding EOP marker, along with a maximum size sort vector. The host code, seeing that the EOP count field is greater than MAX_SORT_VECTOR_LENGTH, performs a software sort on the "excess" srefs and then merges this list with the sorted srefs represented by the sort vector.

Because in the present prototype buckets are not drained immediately upon receipt of an EOP marker, there must be a mechanism for retaining the "sort type" information with the sort vector. In V2, this is handled by the use of special "marker bytes". Specifically, when the sort vector is temporarily stored (in a queue of sort vectors corresponding to the particular bucket under consideration), it is prepended with either a "NO_EXTENDED_SORT byte, indicating that the sort was performed entirely in hardware, an "EXTENDED_SORT" byte, indicating the need for a hybrid sort, or an "ALL_SOFTWARE_SORT" byte. In addition, an "EOS_MARKER" byte ("end of sort") is appended to the sort vector to allow the code to locate the end of one sort vector and the beginning of the next (although technically an explicit end of sort indicator is not necessary, I chose to use one for the sake of consistency, as this makes the code for handling sort vectors mimic the code for draining and maintaining buckets).

3 Testing the SIU

The testing of the hardware SIU can be roughly divided into two phases, the debugging phase, and the "high-level" testing phase. In the first, various tests are performed and modifications made in order to move the prototype (both H/W and V2 software) to a state in which the system appears to be functioning properly. At this point, the second phase begins, with higher level testing used to verify that each individual SIU design requirement (as specified in [8]) is being met. This section is similarly divided into two subsections, each corresponding to a testing phase.

3.1 Simultaneously Debugging the SIU and V2 Code Base

One of the difficulties with debugging the SIU is that both the hardware SIU and the V2 software with which it interacts have to be debugged simultaneously. This situation arises in the present effort for two reasons. First, it would be inefficient to implement a software module to accurately test the V2 code since such a module would inevitably have to be debugged simultaneously with the V2 code in the same manner as is being done with the SIU. More important, a software module would be unlikely to accurately recreate the exact timing of events as they occurs on the SIU, and in many cases it is timing discrepancies that cause the bugs we are attempting to eliminate.

Our debugging philosophy has been to attempt to run a relatively complex application, "dining philosophers",and see where it breaks. An alternative approach would have involved sending various test vectors through the SIU in an effort to verify individual performance requirements (which is essentially what will happen after the initial debugging phase is completed). It was felt that such a systematic approach, however, would have been difficult in the initial testing phases because of the relatively large number of possible interactions that can occur in an isotach system. Using the "track down the bugs as they occur" approach has allowed us to isolate specific problem areas and then target them for the "vector approach".

Every test that has been run thus far has involved one of the connection schemes shown in Figures 2 and 3. Setup "A" is typically used for running dining philosophers. Setup "B" is typically used for analyzing and recreating specific events once a system breakdown has occurred. For the remainder of this paper, the terms "V2 host" and "V1.75 host" will be used in describing hosts as depicted in Figures 2 and 3 regardless of whether those nodes are actually running the V2 or V1.75 system code. That is, the V2 host is always the host directly connected to the "to host" side of the SIU, while the V1.75 host is the host directly attached to the switch if in setup A, or attached to the "to net" side of the SIU if in setup B. Also, to be technically correct it should be noted that there is typically a switch between the V2 and V1.75 hosts in setup B, although whether or not such a switch is used in practice has little effect on the individual tests aside from the necessity of adding a routing byte to messages. In addition, the TM module in setup A can be either a software TM emulator, or a hardware TM.

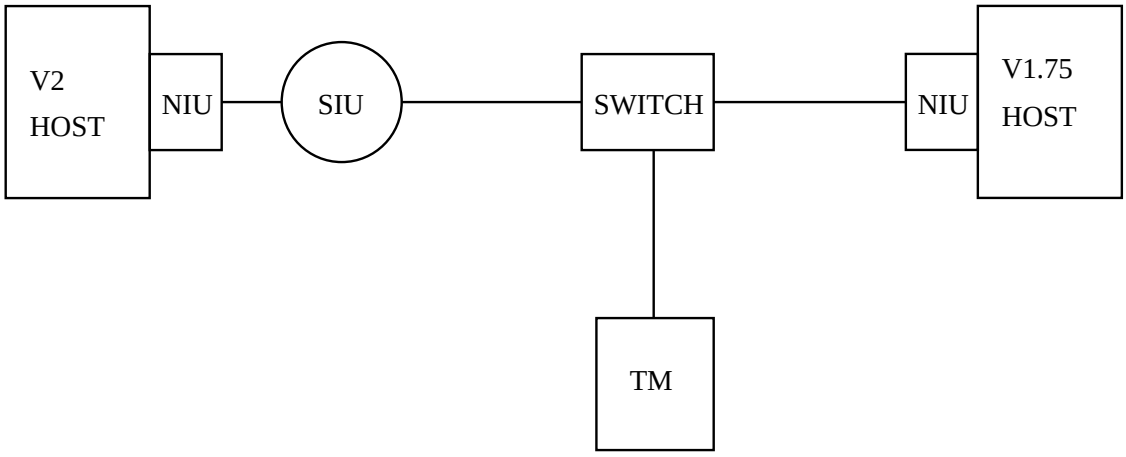

Figure 2: System setup "A" for SIU testing

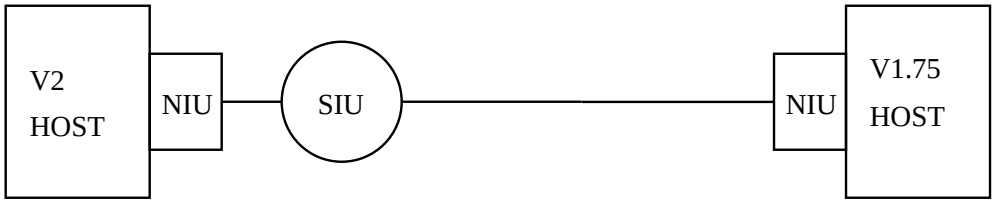

Figure 3: System setup "B" for SIU testing

3.1.1 Tools and Tests (Both Existing and "To Be Developed")

As mentioned above, one of our primary tests involves running dining philosophers in setup A, usually with system code that has been instrumented to reflect a particular mechanism. Since a knowledge of the system code is assumed, the present section focuses instead on the lower level tools that have evolved during the debugging process. Some of these have been implemented and are currently in use. Others are in development, the result of knowledge gained only recently. (The purpose here is to provide both a guide to what exists, and a description of what would be handy for those who will continue the testing effort). Finally, although there have been more than a dozen different tools used in debugging, only those that have proven to be helpful general purpose tools are described here (a tool that extracts only EOP markers with bad CRCs, for example, was useful to us, but only for analyzing a specific problem).

By far the two most useful tools are a byte generator, which sends bytes, and thus arbitrary packets, out to the network, and a receiver, which detects bytes on the wire and prints them to standard out (of course, the tools mentioned here always consist of a host code, LANai code pair—in the case of the receiver, the LANai code detects the incoming bytes and transfers them to the host code, which performs the output). The byte generator program has been enhanced in various ways (to allow the user to resend the previous packet, for example), but even in its simplest form, it allows the user to create any packet, append a tail flit, and then send it. Piping input from a file allows the sending of whole sequences of packets, useful for simulating the sending of an entire isochron. By using setup B, and

running the byte generator on the V1.75 host and the receiver on the V2 host, one can control what the SIU receives, and view what is passed through to its host. At present, each node runs either the byte generator or the receiver, but not both. A hybrid tool that allows a node to both send and receive should be available in the very near future.

A tool that would be most useful for performing several of the higher level tests discussed in the following sections is the "token bouncer". This program would be capable of both sending and receiving tokens with correct sequence numbers (signal and barrier bits could be modified to match specific test requirements), and of implementing the lost token algorithm. In addition, the user should be able to send arbitrary packets (which should be sent in their entirety between tokens) and have the option of recording either all received packets, or only non-token packets. Such a tool would allow the performance of tests for which the continued progress of logical time at machine speeds is essential. Such tests are necessary due to timing considerations—it is sometimes the case that an experiment run at "human speed" will indicate that the SIU is performing correctly, while at machines speeds it indicates a problem. Several of these situations have already been encountered during the debugging phase. (The SIU was originally unable to distinguish the end of an isotach message when it was followed immediately by a token (at machine speeds), for example, and had difficulty creating sort vectors at machine speeds.) In addition to allowing these machine speed tests, such a tool could also be used to run simulations in which it is necessary to create a temporarily "pause" in logical time.

Of course the most involved tool (by far) that has been implemented is the version 1.75 prototype code discussed earlier. By emulating the presence of a hardware SIU, it allows for the execution of isotach application code (such as dining philosophers) in a system containing only a single hardware SIU. This is more than simply a convenience—verifying individual requirements provides no guarantee that the device will be able to integrate the individual functional mechanisms into a correctly performing whole, especially when the timing of events plays such an important role in the intended operating environment.

3.2 "Higher Level" Tests

Each of the SIU requirements are listed below, followed by a test to verify that the criteria is being met. Although as mentioned above the intent was to first debug and then perform the higher level tests, in certain cases a side effect of the debugging process is that SIU behavior has shown to meet a specific requirement. In these cases, the specific context in which the required performance was verified have been described.

3.2.1 Tracking Logical Time

1) In SIU mode, each SIU initially sends and receives a token from (to) the neighboring TM with sequence number start. The value of start does not matter as long as all SIUs and TMs use the same starting sequence number. (In fact, both use start=01.)

Use setup B and run the token bouncer tool on the V1.75 host. If the token bouncer is run in a mode that has it print all incoming bytes, then it should be easy to check that the SIU has sent a token with the correct starting sequence number. To show that the SIU has received a token with the correct starting sequence number, simply look for the subsequent

sequence number in the list of token sequence numbers received by the token bouncer. In order for the TM to have received a START + 1 (mod 4) token from the SIU, the SIU must have received a START token from the TM.

Of course, the latter part of this test assumes that the SIU is meeting requirement (2), so I recommend checking requirement 2 before requirement (1).

2) In SIU mode, when an SIU receives a normal token with sequence number receiveseq, it can send a token to its neighboring TM with sequence number (receiveseq+1) mod 4.

Run the same test in the same setup as recommended for requirement (1). Examining the output of the token bouncer should provide verification.

3) In SIU mode, subject to the other requirements, an SIU attempts to send as many (non-duplicate) tokens as possible, i.e., the SIU does not unnecessarily hold up logical time.

This requirement is somewhat difficult to verify.

4) In SIU mode, an SIU increments its sendclock when it sends a non-duplicate token.

The key here is in noting that by controlling the sendclock on the neighboring TM, one automatically controls the clock value on the SIU. So, use setup B, and run a modified version of the token bouncer on the V1.75 host and a straight byte generator on the V2 host. The modification to the token bouncer should be that after x tokens have been sent (say 10), the token bouncer temporarily stops sending tokens. This will stop the sendclock on the SIU (as well as generating a number of repeat tokens). During this pause, generate a single sref isochron and send it from the V2 host through the SIU. The SIU will timestamp it and pass it to the V1.75 host. When this isochron is received, have the token bouncer send an additional token to the SIU. After this, send another single sref isochron identical to the first from V2 host through the SIU. If the SIU has correctly incremented its sendclock, the timestamp on this second sref should be one greater than the timestamp on the first isochron.

5) In host mode, an SIU increments its sendclock when it sends a normal token to the network.

This test is easier than the previous one because in host mode the V2 host can exert complete control over the timing of token sends. So, once again use setup B, only this time run the token bouncer on both the V2 and V1.75 hosts. Modify the V2 token bouncer slightly so that between the sending of any token, it sends out a single sref isochron (the same single sref isochron). Since these isochrons are going to the same host, they all have the same isotach distance. Because of the interleaving of tokens and isochrons, the timestamps on the

isochrons should form an increasing sequence with difference one between each subsequent message. Reading timestamp values on the srefs received at the V1.75 host will determine whether this is in fact the case.

6) In SIU mode, an SIU assigns a value to the source port field of each token it sends that will identify it to the receiving TM as the sender.

Use setup B and run the token bouncer on the V1.75 host. By looking at the values of the source port field in tokens returned to the token bouncer, one can verify that the SIU is correctly loading the source port bits.

7) The SIU checks the CRC of tokens (including tokens received from the host in host mode) and drops any token with a bad (non-zero) CRC.

This would be relatively easy to test if we could somehow generate tokens with bad CRCs. I'm not quite sure how to do this just yet.

8) In host mode, if the SIU receives an early token from the host it drops the token (or corrupts the CRC).

Use test setup B with a simple byte generator running on the V2 host and a token bouncer on the V1.75 host. Using the byte generator, respond (by hand) to the tokens received at the V2 host. Every so often send an early token and observe whether the token is received (or has a bad CRC) at the token bouncer on the V1.75 machine.

9) Each SIU has a token timer with which it is capable of timing events at approximately the token timeout interval (see initialization section) granularity. In SIU mode, the SIU resets the timer whenever it sends a token. If the timer elapses, an SIU running in SIU mode resends its last two tokens.

Another relatively simple test. Use setup B, run a modified token bouncer on the V1.75 host. Have the V1.75 host pause long enough to generate repeat tokens, and then see what the SIU sends.

3.2.2 Notifying the Host of Significant Pulses

10) In SIU mode, the SIU sends a pulse i EOP marker iff pulse i is a significant pulse. Pulse i is significant if an isotach message or isochron marker was sent to the host with TS i, if token i+1 (the token that ends pulse i) completes a barrier, contains a reset signal, or is the end of epoch token and a signal bit is set.

Use setup B, run a simple receiver on the V2 host and a token bouncer on the V1.75 host. Have the V1.75 host generate srefs and send them to the V2 host. We can control the timestamps generated on the V1.75 host, and ensure that they have been received correctly on the V2 host. The only thing left to check is that the SIU has sent EOP markers only for those pulses that are significant (we can create significant pulses by adjusting the signal and barrier bits of the tokens generated by the token bouncer).

11) In host mode, the SIU sends an EOP marker to the host for each normal or repeat token it receives from the network.

This has been verified during debugging. In setup B, we sent a stream of messages and tokens from the V1.75 host to a V2 host running a simple receiver. The behavior was also demonstrated during partial runs of dining philosophers in host mode.

12) In host mode, the SIU drops or corrupts the CRC of every early token it receives from the network or host.

Once again use setup B with token bouncers on each host. To test the tokens coming from the network, modify the token bouncer on the V1.75 machine to occasionally (say every fifth token) send an early token. By reading the bytes received at the V2 host, we can verify that either the token is dropped or discarded. To test the tokens coming from the host, simply switch the roles of the two token bouncers.

13) The SIU sends the pulse i EOP marker only after sending the host all isotach messages with TS i destined for the host. The SIU can and should assume that it has received all such messages upon receiving token i+1.

Use setup B with a token bouncer on the V1.75 host and a simple receiver on the V2 host. Have the token bouncer send isochrons to the V2 host and use the receiver to verify that in each case the EOP marker for pulse i is received only after all of the messages with timestamp i have been received.

14) The SIU sends the pulse i EOP marker only after sending the host all isochron markers with TS i.

Caveat: this requirement may be difficult to meet because it involves both sides of the SIU.

Once again use setup B with a token bouncer on the V1.75 host and a combined byte generator/receiver on the V2 host. Have the V2 host generate a number of isochrons. By reading what is returned from the host, we should be able to verify that we are receiving the pulse i EOP marker only after receiving all of the isochron markers with timestamp i.

15) If the SIU sends a pulse i EOP marker, it must send the marker before

sending the host any isotach traffic from the next congruent pulse modulo the timestamp range.

This requirement is difficult to test and probably requires more thought. The issue is that the SIU should not be holding EOP marker "too long". So, to ensure that it isn't, I would present the SIU with a sort of worst case scenario. The worst case scenario here is a steady stream of messages received between each token, with each message in the group assigned the same timestamp. This is worst case because it ensures that the SIU is heavily loaded and has to do the worst case sort and create the longest possible sort vector. So, use (as always) test setup B with a token bouncer on the V1.75 host and a simple receiver on the V2 host. Modify the token bouncer so that between each token it sends 32 isotach messages, each assigned the same timestamp, perhaps the current value of sendclock (but stored as a 32 bit quantity). This sendclock value should be loaded into the payload portion of the net_sref (so that there is a 32 bit record of it when the sref is received at the V2 host). One could then either examine the packets received at the V2 host by hand (difficult) to guarantee that the EOP markers were arriving "in time", or one could modify the receiver code to check for this and inform the user if an EOP marker violated the requirement.

How would one make this modification? Well, keep an array of counters on the V2 host, one for each bucket ON THE SIU (not the same as the buckets in the host code). So, suppose there were 8 buckets on the SIU. Then we would have 8 counters. Whenever an EOP marker arrived with a timestamp that was equivalent to 0 mod 8, then counter[0] would be incremented. By checking the value of this counter against the value of the 32 bit timestamp in any sref with a timestamp congruent to 0 mod 8, we can determine whether the EOP marker has arrived late. I.e. if we received an sref with 32 bit timestamp value 16 when the value of counter[0] was 1, then we'd have a problem, because counter[0] would have been incremented to 1 when the EOP marker for timestamp 0 arrived, and since a timestamp 16 sref should not be delivered to the host before the second EOP marker corresponding to a timestamp congruent to 0 mod 8 has been delivered to the host.

16) The SIU copies the packet type and CRC fields from the i+1st token into the corresponding EOP marker.

Use setup B and run a simple receiver on the V2 host and a token bouncer on the V1.75 host. By modifying the token bouncer to send single sref isochrons between each token, and printing both the tokens generated on the V1.75 host and the EOP markers received at the V2 host, this requirement may be verified.

17) In SIU mode, the SIU writes i into the TS field of the pulse i EOP marker.

This requirement can be tested by using the same setup as for requirement (16) and checking the timestamp fields in the EOP markers received at the V2 host.

18) In host mode, the SIU sets the repeat bit in an EOP marker iff the

corresponding token is a repeat token. The remaining bits of the TS are X's (don't cares) in host mode.

Again we can use the same test and setup as in (16), but with the slight modification that every so often the token bouncer on the V1.75 host generates a repeat token (perhaps every third token or something similar).

19) The pulse i EOP marker contains the total count of isotach messages and isochron markers sent to the host with TS i (even if the count is greater than bucket_size), except in host mode when the corresponding token is a repeat token.

Again, the same setup can be used as in (16), with the modification this time that the token bouncer, instead of generating single sref isochrons between each token, now generates isochrons with a random number of srefs between each token. By recording the number of srefs sent during each pulse, and examining the count field of the EOP markers received at the V2 host, we can verify that we are receiving an accurate count.

3.2.3 Separating Isotach from Non-Isotach Traffic

20) An SIU does not unnecessarily delay non-isotach traffic.

This is a tricky requirement to verify. The difficulty lies in the term "unnecessarily delay". Perhaps a better way of stating this is to say that the increased delay experienced by non-isotach traffic should be reasonable. Given this, one could perform the following comparison. Use setup B. Run a program on the V2 host that generates a large number of non-isotach messages and a program on the V1.75 host that simple bounces any non-isotach message back to the sender. By instrumenting these appropriately, we can get an idea of the average latency for these messages when the system is not performing any isotach specific operations.

Next, replace the non-isotach message bouncer with a token bouncer modified to reflect any non-isotach messages it receives and (for good measure) to send random isochrons between each token. By having the V2 host generate the same stream of non-isotach messages as before and recording the average latency, we should be able to understand the delay introduced by the SIU.

For good measure, we could also perform the first step of the test with the SIU completely removed from the system, thus getting three latency readings: one with the SIU removed, one with the SIU connected but not performing any isotach functions, and one with the SIU connected and processing isotach messages and tokens.

3.2.4 Maintaining FIFO guarantees

21) The SIU sends isotach messages in FIFO order, i.e., for any two isotach messages m1 and m2 traveling through the SIU in the same direction (both are host-to-net or both are net-to-host) if m1 arrives at the SIU before m2, the SIU sends m1 before m2.

Again use setup B, this time with combined byte generator/receivers running on both hosts. To test the net-to-host traffic, have the V1.75 host send a large number of isotach messages to the V2 host, each message containing a sequence number so that we could tell which messages were issued in which order. Testing the host-to-net traffic requires having the V2 host, rather than the V1.75 host, generate the messages.

22) The SIU sends non-isotach messages in FIFO order.

Similar test as in (21). Have the V2 host send a bunch of non-isotach messages containing sequence numbers, etc.

23) In host mode, the SIU sends isotach packets (tokens and messages) to the network in the order in which they are received from the host.

Similar test to (21) and (22), except that we may have to use the signal bits as a counter to store the sequence numbers used in the experiments.

3.2.5 Enforcing isochronicity and sequential consistency

24) All messages in same isochron must be assigned the same timestamp.

We can test this by sending a number of isochrons from the V2 host to the V1.75 host and checking the timestamps on the incoming srefs. To keep track of the srefs, they might all be isotach messages containing nothing but identification marks indicating the isochron to which they belong.

As always, use setup B with a byte generator set up on the V2 host machine and a token bouncer on the V1.75 host. Have the V2 host feed the SIU a string of isochrons and read the timestamps on the messages received at the V1.75 host.

Again, we can only verify this requirement through empirical observation. To achieve the most complete coverage of the possible problem space, we would have to send a large enough number of isochrons (i.e. run the test enough times) so that we can be reasonably confident that in all instances this requirement is met. For example, use many different isochrons, and have the various srefs within a single isochron go to a variety of destinations (note that we really don't need to have four or five physical receive hosts—since we're running this in a controlled environment, we can simply feed packets that *indicate* that they are going various logical distances to various hosts (by adding extra routing bytes), but actually have all the packets go to the same host).

25) The TS assigned each isochron must be no less than the sendclock at the time the TS is computed plus the isochron distance plus the send delta.

Note that technically, this is impossible, since we have a limited number of bits with which the sendclock is implemented, so wraparound is inevitable for some isochrons.

A difficulty here is determining exactly what the sendclock value is on the SIU. The best way to do this (short of actually using a digital analyzer to probe some register on the SIU) is to use setup B, and run a modified token bouncer on the V1.75 host. The modification will be that at a specified logical time (i.e. number of tokens received) the token bouncer stops bouncing tokens. The SIU will then keep sending repeat tokens, but at least we'll know what its sendclock value is. At this point, we send isochrons as in (24) with srefs that indicate they are going to various machines. Reading the timestamps in the receive host does the trick.

26) The TS assigned each isochron must be no less than the sendclock at the time the last flit in the isochron is sent plus the isochron distance.

This requirement implies coordination between the TS computation and the sending of tokens. The sendclock can be incremented while an isochron is being sent, but only if the sendclock plus the isochron distance remains less than or equal to the TS of the isochron.

We can extend the tests for (24) and (25) to help us here. Start with the test for (25). When we stop the receive token bouncer, we know what the sendclock value is. At this point, send the start and first few srefs of an isochron (but NOT the end of the isochron). Then we let the token bouncer send one more token, (or perhaps five or ten or however more) and stop it again. This will have caused the sendclock to increment however many times. Now send the rest of the isochron. By doing this enough times, we can be reasonably sure that this requirement is being met.

27) The TS of a sequenced isochron from channel x and seqcon set y is no less than the TS of the isochron with the latest TS among the logged isochrons from the same channel x and the other set NOT y. The channel and seqcon set of an isochron is determined by the value of the isochron's SOI channel field and seqcon set field respectively. The TS of an isochron figures into the TS computation for future isochrons only if the message is a logged message.

The SIU tracks the TS of logged isochrons and ensures that future isochrons (of the same channel and different concurrency set) are assigned a TS no smaller than that of the current isochron. Tracking the TS requires care to avoid problems with TS and clock wrap. The SIU is required (due to requirement 28) to represent the latest TS for each channel and set as a difference: define gap[x][y] as the difference between the latest TS for channel x and set y and the sendclock if the result is positive and as 0 otherwise. (Operationally, each time a logged isochron from channel x, set y is sent, gap[x][y] is set to the maximum of 1) its current value, and 2) the TS of the isochron minus the sendclock. When the sendclock is incremented, gap[x][y] is decremented, until the value reaches 0.)

Using setup B with token bouncers on both hosts, one would need to generate an appropriate set of isochrons (i.e. all isochrons from the same channel but from a number of

different concurrency sets—this can be controlled via the channel and seqcon set bit fields in the isotach prefix of the start of isochron (SOI) packet).

It should be noted that at present there is no support in any version of the prototype code for channels or concurrency sets (as is clear from above, however, this should not preclude effective testing of this requirement).

28) For each logged host-timestamped isochron sent on channel set x, set y, the SIU writes the value in the GAP field of the SOI message into gap[x][y].

The same setup and test as for requirement (27), though now it is necessary to check the gap field in the isochrons received at the V1.75 host.

29) The SIU assigns each isochron the earliest possible TS which satisfies the requirements given above.

This can be verified using the same tests as recommended for requirements (27) and (28).

3.2.6 Sending Flow Control and CRC Updates to the Host

30) The SIU must send all isochron markers it sends to the host in FIFO order, i.e., in the order in which it received the corresponding isochrons (even if the TSs it assigns the isochrons are not increasing).

We can test this by generating a number of isochrons on the V2 host, sending them to through the SIU, and then reading the corresponding isochron markers at the host. Specifically, the SIU is supposed to copy the iso_id byte from the SOI message into the corresponding isochron marker. By generating isochrons with strictly increasing iso_ids, we'll be able to know which isochron markers go with them by looking at their iso_ids.

31) The SIU sends the isochron marker to the host after it has removed the last byte of the last message in the isochron from the on-board buffer for holding out-going isochrons.

I'm not sure that this can be verified without careful low level hardware testing. This may be one that has to be verified by the hardware guys.

32) The isochron CRC field in the isochron marker the SIU creates for an isochron contains the result of the non-exclusive OR'ing of the CRC fields of all the messages in the isochron.

To test this, we need only send an isochron, read the values of the CRCs of the srefs

that we get at the V1.75 host, and then compute the non-exclusive OR of these CRCs. If this matches the isochron CRC field in the corresponding isochron marker, we're golden. We'll just have to be sure we run this test with enough different isochrons to be reasonably sure that the SIU satisfies this requirement.

3.2.7 Determining Delivery Order Among Packets with the Same TS

33) Except in host mode when the corresponding token is a repeat token, the SIU sends the sort vector for pulse t in the pulse t EOP marker.

This can be verified simply by running philo. If we don't get the sort vector for pulse t in the pulse t EOP marker, then the code breaks. We can also test this directly with the digital analyzer (as we have been doing during the debugging process).

34) The sort vector for pulse t contains an element for each isotach message or isochron marker with TS t that the SIU sent the host, up to bucket size elements.

Again, this has been tested through the use of the digital analyzer. As a redundant check, we have instrumented the V2 code to print any sort vectors it receives.

35) Element i of the sort vector for pulse t is s if s is the seqnum for the ith packet in the sorted order of packets with TS t. The sorted order among packets with the same TS is increasing order first by source and then (among packets with the same TS and source) by seqnum. If more than bucket size packets with TS t are received, the sort vector represents the result of sorting the first bucket size packets received.

Again, this is being verified during debugging.

3.2.8 Handling Signals and Barriers

36) The SIU can send a token reflecting the receipt from the host of a barrier/signal marker only after it has sent or started sending all isochrons sent by the host before that marker.

Note that the SIU can send a token reflecting the marker before it has completely sent the previous isochron (assuming the isochron has a send delta greater than zero).

Again, a hard one to test. Timing here is going to be tricky.

37) In SIU mode, the SIU zeros out the signal field in each EOP marker except in the following cases: 1) if the corresponding token is the end of epoch token, the SIU copies the signal bits from the token; and 2) if the

reset signal bit in the corresponding token is set, the SIU sets the reset signal in the EOP marker.

Ok, run a receive byte program on the V2 host with the SIU in SIU mode. Run our token bouncing program on V1.75, but modify it so that between each normal token, it sends a single sref isochron to the V2 host. This should stimulate the SIU to produce a continuous stream of EOP markers. By viewing these EOP markers on the V2 host, we can see whether or not the signal field has been zeroed, and whether the signal bits are correctly propagated if the token is an end of epoch token and if it's a reset token.

38) In SIU mode, the SIU zeros out the barrier field in each EOP marker except in the following case: if the corresponding token completes a barrier(s), the SIU sets the bit for that barrier(s) in the EOP marker.

Testing this is similar to testing (37). The only difference is in examining the barrier bits instead of the signal bits, and in checking that the barrier bits are NOT zeroed out only in the EOP marker corresponding to the pulse in which the barrier has completed.

39) In host mode, the SIU copies the signal and barrier fields from each normal and repeat token it receives from the network into the corresponding EOP marker.

Using the same setup as in the tests for (37) and (38), simply have the V1.75 host pump tokens with various signal and barrier bits to the V2 host. (Be sure, of course, that the SIU is in host mode). Reading the various signal and barrier bit fields of the token should provide confidence that this requirement is being met. Note also, that unlike the tests for (37) and (38), there is no need to send isochrons to initiate the generation of EOP markers, because for this test, we are in host mode.

4 Conclusions

The goal of this project was to design and prepare tests and tools to be used to verify that the SIU meets performance requirements as stated in the design document. In addition, what testing and debugging was possible was to be performed subject to project time constraints (at the time the project was initiated, it was not known whether an SIU would be ready for testing prior to a June 1999 graduation date). These goals have been met:

• Tests for each SIU requirement in the isotach design document have been described above, along with descriptions of the tools necessary to perform these tests.

• More than a dozen test drivers have been implemented and used during the debugging process. In particular, the byte generator and receiver tools have been used extensively. (In addition, at this writing the combined byte/generator receiver has been successfully implemented by my isotach colleagues, and the token bouncer is expected to be completed

shortly).

• Two new versions of the prototype have been implemented, V1.75 which emulates a hardware SIU and is intended to run in system with SIUs attached to other hosts, and V2 which is the code base that allows a host to interact when connected directly to a hardware SIU. The former has been tested (by running it against itself and with both software and hardware TMs) and performs well. The latter can only be tested and debugged in conjunction with the SIU. With both prototype versions performing in the same system, it is possible to run isotach applications with a system containing only a single hardware SIU, allowing us to test complex device functionality in relative isolation.

• The debugging process has been ongoing since the SIU was delivered in April. Significant progress has been made and several mechanism have been demonstrated to be performing correctly. Although neither the SIU nor the V2 code appear to be bug free at this time, the steady progress that has already occurred indicates that both modules should be ready for individual "requirement" testing in the very near future.

A Comments on the Isotach Prototype Code

This file is an attempt to make some sense of all of the individual pieces of Isotach system code. The first thing it does is list all of the routines in each of the main code modules (lcp.c, smm.c, fastmsgs.c, and initialize.c) along with the routines called by each. Using this, one can hopefully get a feel for code flow in the system. One note: there are a lot of preprocessor directives, so sometimes routines are named differently depending on the mode of the system (i.e. Host-to-host or just regular SIU mode) and/or whether a node is a host or a TM.

A.1 lcp.c

The file lcp.c is the primary source code for the executable that runs on the LANai. Its primary routines are described below.

A.1.1 send_path()

This routine takes a destination node as input and places the appropriate routing bytes on the wire.

A.1.2 synchronize()

This implements an n^2 algorithm to synchronize all of the nodes. HELLO packets are send to all other nodes. If we receive a HELLO packet from another node, we send a HELLO_ACK packet. We expect these back from nodes to which we have sent HELLO packets.

A.1.3 main()

Main first checks to see if a reset needs to be done. If so, it does one. main() then does a sort of initialization "dance" with the function FM_initialize(). main() spins while

FM_initialize() does the following:

1. calls lanai_read_symbol_table(), reading the symbol table of lcp.c

2. calls open_lanai_copy_block()

3. grabs pointers to variables and data structures in the LANai using calls to get_lanai_sym(). get_lanai_sym returns pointers to the LANai code variables named as input parameter (this allows the two code bases to "talk" with one another.

4. Begins a sort of DMA engine test. FM_initialize() places values into the UBLOCK location. UBLOCK is the user accessible area on the host into which the LANai DMAs data. It then calls lanai_load_and_reset(), sets the value of a LANai pointer into that memory (which is also known as DBLOCK when being accessed by the kernel, but it's the same memory. DBLOCK gives the kernel memory address, UBLOCK the user space address).

FM_initialize then spins waiting for main() to do the following:

1. Set up and execute a DMA transfer from the LANai SRAM into UBLOCK on the host, effectively resetting the values there.

main() then spins again, while FM_initialize does:

1. Checks the values just written into UBLOCK to verify that they match what they are supposed to be. If not, the program halts.

2. Initializes remaining data structures and pointers into them. These include hrecvptr, stailptr, *hostbase and counters of the number of DMAs performed, etc.

3. Calls configure(), see description in the initialize.c descriptions below.

4. Calls long_init(), again see descriptions below.

5. Allocates (initializes) send and receive credit. These are initialized to their max values.

FM_initialize() then spins waiting for main() to

1. call synchronize() and thus do the HELLO thing.

2. Initialize various LANai code data structures such as recvqueue[i].checksum, recvqueue[i].valid, in_iso_traffic[i], LAR (LANai starting address for DMAs), EAR (host starting address for DMAs), and various other counters.

3. reset the real time clock (variable used is RTC—see LANai users manual).

main() then tells FM_initialize that it has completed its initialization, and both routines then continue concurrently. FM_initialize sets the value of message handlers (calling FM_set_handler() three times), and then completes. main() continues by initializing a few more variables and then entering the main event loop. The main event loop is different for TMs and non-TMs. Accordingly, there are two versions of receive_messages() and two of send_token() as described below.

So, if one is a TM, then the main event loop is this:

1. call receive_messages()

2. update the recv_vec array. recv_vec[i] tracks the logical time (i.e. the number of tokens received) at port i on the switch to which the TM is attached.

3. if it's time to send a token, call send_token().

If one is not a TM and we are in HH mode, then the loop looks like:

1. call receive_messages()

2. if what the host has "sent" is not equal to what the LANai has sent (i.e. if there is something in the host send queue that has not been placed on the wire, checked by looking at the condition hostsent != LANaisent), call send_messages().

If one is not a TM and we are not in HH mode, then the loop looks like:

1. call receive_messages()

2. if recv_clock ¿ send_clock, then call send_token.

3. If hostsent != LANaisent, and we're in an isochron, call send_messages(). (i.e. keep calling this until we are no longer in an isochron (i.e. in_isochron = 0)).

A.1.4 recv_messages() — TM version

While there's a byte available at the LANai packet interface, DMA a packet into SRAM, check it's length, checksum, whether we received the whole thing, and if it's a TOKEN. If all checks are ok, increment the appropriate host's (really should be port's) clock (i.e. token counter). This is done with recv_vec[invhosts[host]]++ If any of the checks fail, drop the packet.

A.1.5 recv_messages() — Non-TM version

Some things to know here. Particularly, the uses of variables.

hostspace: the number of RECVFRAME size slots left in the host receive queue.

lcpspace: same as above but counts frames left in LANai receive queue.

iterations: initialized to lcpspace, but really its value is the minimum of hostspace and lcpspace (so that we are guaranteed to have at least "iteration" spaces left in both queues should we have to do a DMA from LANai to host).

newreceives: incremented by one every time we receive a new packet. This number is then added to nextframesdmaed, and finally nextframesdmaed is used to increment framesdmaed, which counts the number of frames dmaed. This is really not nearly the most efficient way to do this. One could instead just increment framesdmaed by the appropriate number of frames after each LANai-host DMA.

diddma: set to one immediately after we've DMAed something to host. Initialized to zero, and set to zero every time we update framesdmaed with nextframesdmaed.

in_iso_traffic[NUM_BUCKETS]: in_iso_traffic[i] keeps track of whether or not we have sent an isotach message to the host in pulse i (recall that bucket numbers refer to time pulses). Value is 1 if we have sent something, zero else.

recv_messages() first updates (if appropriate) framesdmaed with nextframesdmaed. It then

begins its main loop:

> While there is something at the packet interface and we have room to DMA stuff from LANai to host:
>
> 1. DMA a packet into SRAM. do the usual checks and drop it if any of them come up wrong.
>
> 2. switch on the size of the packet:
>
>> case: size of a TOKEN (verify that it is in fact a token – these are the only packet types we should get of length 5 bytes). If HH mode, increment eop_marks. Else, increment recv_clock, and check to see whether we need to send a TAG_ISO_EOP to the host (the check involves looking at in_iso_traffic[]).
>>
>> case: size of a packet. Increment newreceives and recvtail. Also, if it's of type TAG_ISO_MSG, then set the appropriate array element of in_iso_traffic to 1.
>
> After this main loop is exited, then recv_messages(), and if a message has been received (newreceives!=0):
>
>> 1. update lcpspace
>>
>> 2. Take everything we just DMAed into SRAM in above loop and DMA it into host, carefully keeping track of space available in each of host and LANai receive queues. We know what we DMAed into SRAM in the main event loop above by keeping track with recvtail. In fact, DMA is initiated with the assignment DMA_CTR = ((ULONG) recvtail - (ULONG) LAR);

A.1.6 send_token()

As mentioned above, there are two versions of this routine. If we are in HH mode, then neither is present, (excluded by preprocessor directive). Assuming then that we are not in HH mode, there are both TM and non-TM versions. Also, there are some issues here with the constant NO_SEND_ALIGN that are not yet completely understood. In the code version I have, this has value 1. It has something to do with the way the routing bytes are placed on the wire.

> send_token() – TM version
>
>> 1. Increment send_clock
>>
>> 2. send a token to appropriate destinations (i.e. call send_path() and DMA packet onto wire multiple times).
>
> send_token() – Non-TM version
>
>> 1. Same thing as above, only just send it to my TM. Why would we do this?

A.1.7 send_messages()

This routine puts messages on the wire.

Do the following only if we're not in HH mode and the packet we are about to send has tag STRUCT_ISO_MSG:

1. If we're not in an isochron, set in_isochron to 1, set send_mark to 1, and update last_receive_time. It's instructive to know exactly what send_mark is used for. It is initialized to 0. When we first enter an isochron, send_mark is set to 1. This value is later checked to see whether we should send a TAG_ISO_CHR (an isochron marker) back to the host. This is only done once for each isochron, and the TAG_ISO_CHR packet contains the logical receive timestamp assigned to the isochron. Once this packet has been sent to the host, send_mark is set back to zero.

2. if the head.flags field of our packet is equal to 0x2000, set in isochron to 0. Now, the reason for this: the messages in an isochron are actually sent as a block during a call to iso_end() (whose code is in smm.c). When the last sref in an isochron is reached, iso_end() sets the flag field of that sref to 0x2000, so that testing against 0x2000 is a test to see if we have the last sref in an isochron.

3. Set head.ts value to last receive time.

Regardless of whether we are in HH mode or not, the routine always executes these steps next:

1. place the routing bytes on wire with call to send_path()

2. DMA packet onto wire.

3. increment sendptr

4. If we're not in HH mode and the value of send_mark is 1, send a TAG_ISO_CHR packet to the host, with the calculated receive timestamp of the isochron (see comments above).

5. While waiting for our send to complete, call receive_messages()

6. send is now done, update pointers and counters.

A.1.8 Comments

There really is no receive queue on the LANai. Instead, as stuff comes in on the wire it is DMAed directly into the host receive queue. There is, however, a send queue on the LANai (but not one in the host, as I recall—stuff is written directly from the host into the LANai send queue).

A.2 initialize.c

It helps here to understand what the calling sequence is for the routines here. There are two FM API functions here, FM_set_parameters(), and FM_initilialize(). FM_set_parameters() must be called before FM_initialialize(). The exact sequence of events in FM_initialize is detailed in the above description of lcp.c routines. This is necessary because FM_initialize() and the main() routine of lcp.c are coroutines. In any case, the calling sequence here is as

follows:

> FM_initialize() calls configure(), which in turn calls both readconfiguration() and then findroutes(). Upon return to FM_initialize(), the routine long_init() (code is in fastmsgs.c — all long_init() does is allocate memory for a host reassembly buffer (purpose is described in section below on fastmsgs.c)) is called by FM_initialize().

Now for the detailed descriptions of these routines.

A.2.1 findroutes()

Before describing this, one needs to know the way routing bytes work in a Myrinet. Recall that at each switch in the Myrinet, a single routing flit is consumed. This routing flit specifies an offset of sorts from the incoming port on the switch to the outgoing port. Specifically, ports on an eight port Myrinet switch are numbered 0 through 8. The two highest order bits are 10, the one acting as a flag bit. The low order six bits are obtained as follows: If the destination port number is less than the source port number, then the routing byte should be the positive difference between the two (i.e. you give the positive offset). If the destination port number is less than the source port number, then the low order six bits should be a negative offset (here negatives are computed as six bit twos complement numbers). This, going from port four to port 6, the routing byte should be

10000010 (i.e. 0x82)

Going from port 4 to port 1, however, the routing byte should be

10111101 (i.e. 0xBD)

See Myrinet documentation for more details.

Now that we know how routing works, describing what findroutes() does is easy. It is a recursive function that determines the single source shortest path from the present node to each of the other nodes (using Dijkstra's algorithm). Once the routes are determined, it records the appropriate routing bytes in the fields

hostlist[targetnodenumber]routing.path[]

That is, the path to get from the present node to the node "targetnodenumber" is specified in the above field. The length of this route (i.e. number of hops) is recorded in the field

hostlist[targetnodenumber]routing.pathlen

A key observation: findroutes() PLACES THE ROUTING BYTES INTO THE PATH ARRAY IN THE CORRECT ORDER. THAT IS, WHEN findroutes() IS DONE, path[0] CONTAINS THE ROUTING BYTE THAT SHOULD GO ON THE WIRE FIRST, path[1] CONTAINS THE BYTE THAT SHOULD GO ON THE WIRE SECOND, ETC.

A code note: as things are set up here, switch names are numbers, while host names are alphabetic (i.e. grover, bigbird). Thus, the atoi conditional is really testing whether the next node we are considering is a switch or whether it is a host.

A.2.2 configure()

This general purpose of this routine is to load node and path information from a static configuration file and set up pointers to the "path" and "skipbytes" variables in the LANai code. The steps it takes are the following:

1. Sets the pointers into the LANai code variables

2. calls readconfiguration(), which as mentioned below merely reads and parses a static configuration file telling which device is connected to which switch, and which devices are TMs.

3. calls findroutes(). (See description above).

4. sets more pointers into the LANai

5. using the routing information obtained from the call to findroutes(), load the routing bytes into the "path" array (path is an array in lcp.c). Of note here is that the byte ordering is switched "manually" as opposed to using htonl(). See my comments in the code to determine just how often byte ordering is switched around here.

6. Determines the network diameter using some convoluted method that relies on counting the number of skipbytes in a route. This could be done much more efficiently and "cleanly" by simply going linearly through the pathlen field of the routing information for each node. (i.e. hostlist[i].routing.pathlen)

7. initializes the three arrays

hh_source_ports_send[]

hh_source_ports_recv[]

hh_inv_recv_source_ports[]

Some helpful info:

hh_lhosts[] is the array of nodes to which I must send tokens if I am a TM and we are running in HH mode. So, e.g. the value of hh_lhosts[0] is the node number of the first node to which I need to send tokens.

hh_source_ports_send[] and hh_source_ports_receive[]: These involve the passing of tokens. In the Isotach implementation document, one should not that there is a six bit source port field in Isotach token packet types. hh_source_ports_send[i] is the value of this six bit field for a token being sent from us to node i. I.e. it is the least significant six bits of the least significant routing byte (i.e. the last one consumed). hh_source_ports_receive[i] tells me the least significant six bits of the most significant routing byte that I would use to send a message to node i. For example, if I (when I use the term "I" it means the host or TM which is running this code—i.e. the code's concept of the node on which it is running) am connected to port 6 on the adjacent switch, and I receive something from node i that arrives at this switch on port 4, then for me to send a message to node i, the first routing byte in the message I send is

1011 1110 = 0xBE

(i.e. the leading 10 followed by -2 in six bit two's complement). In this case, we would have

hh_source_ports_receive[i] = 11 1110 (i.e. 0x3E).

Since these two arrays deal only with tokens, and tokens are at most two hops away, one can use the already computed paths (and look at the appropriate routing byte) to determine which host sent this to me, and that's what these arrays do. FM uses node numbers to determine source nodes and the like, but because of the Isotach packets types and the way the hardware TMs are designed, they only use the six bit fields mentioned above to determine sender.

Finally, the hh_inv_recv_source_ports[] array gives the inverse mapping of hh_recv_source_ports[]. It takes the six bit field and gives a node number. This is why we have

hh_inv_recv_source_ports[hh_source_ports_recv[i]] = i.

A.2.3 readconfiguration()

This routine does exactly what its name implies: it reads a static configuration file. Each line of the configuration file consists of a number (the switch number) followed by one or more device entries, each device entry consisting of a name and information as to whether or not the device is a machine running a software TM or whether the device is a HW TM. The code serves merely to parse the file.

The data structure initialized here are myriswitch[] and hostlist[]. The entries in hostlist[] are indexed so that TMs have the higher indices (i.e. are listed at the end of the array).

Note that readconfiguration() does not compute any routes. It simply defines which device is connected to which switch and which devices are TMs.

A.3 fastmsgs.c

This module contains a lot of short routines that are very straightforward such as FM_set_handler(), FM_clear_handler(), error(), error_if(), and safe_malloc().

Several of them raise questions, however. First off, why are copyLastPacket() and copyLastPacketData() identical but have different names. Also, what's the deal with FM_mallocalligned()? This function calls malloc and returns a pointer to a region of memory only if malloc() returns a region aligned on a long long (i.e. only if the starting address of the region is a multiple of 8). That's fine enough, but the implementation of this is strange. It calls malloc() only once, and if the pointer returned isn't aligned, FM_mallocalligned() returns an error. Why not just have the function keep calling malloc until it returns an aligned pointer, or better, take the pointer and increment it until you have an aligned pointer.

Some notes on control flow:

It is interesting to note that send_last_packet(), if there is no room in the sendqueue, calls FM_extract(), which calls consume() which calls the actual handlers for the packets.

A note on sendcredits:

It appears that any time a packet is sent to a node, the source updates the receiver's send credits so that the receiver has max send credits to the source.

Some more general notes:

There is no real host send queue (though there is a host receive queue—this is what the LANai DMAs into). Instead, FM writes packets directly into the send queue in the LANai. The variable stailptr is a pointer to the next available frame in the LANai send queue. When FM wishes to send a packet, it write the info into the LANai sendqueue directly, then increments both stailptr and the variable myhostsent.

So myhostsent records the number of frames that the host has written into the LANai send queue. THE FM CODE RUNNING ON THE HOST ACTUALLY CONSIDERS THE HOST SENT WHEN IT UPDATES THE LANAI COPY OF THE VARIABLE HOSTSENT, USUALLY WITH A STATEMENT SUCH AS

*hostsent = htonl(++myhostsent)

Why is this? Well, it's because the LANai code (i.e. lcp.c) uses checks like:

if (hostsent != LANaisent) send_messages();

Thus, the LANai doesn't actually place the packets on the wire unless hostsent is not equal to LANaisent. (Note here *hostsent is the pointer used by fastmsgs.c, while hostsent is used by lcp.c. They both refer to the same location in LANai SRAM). Until fastmsgs.c updates *hostsent then, the LANai hasn't really been given the "order" to send the packet.

The variable myLANaisent is updated by statements such as

myLANaisent=ntohl(*LANaisent))

The variable LANaisent is incremented every time the LANai places a packet on the wire.

Now, in the fastmsgs.c code, there are a number of occurrences of the conditional

```
        while (((myhostsent-myLANaisent) >= SENDFRAMES &&
  (myhostsent-(myLANaisent=ntohl(*LANaisent))) >= SENDFRAMES))
```

What this conditional is doing is first checking to be sure that there is space in the LANai send queue for whatever is to be placed there next by whatever routine in fastmsgs.c, then updating myLANaisent and checking again. It would seem that this is redundant and should probably be fixed.

A.3.1 refill_credit()

This routine does exactly what one would expect. It checks to see that there is room in the send queue (I'll use the generic send queue since there is only one, and it's on the LANai) for a packet. If there isn't, it spins waiting for the queue to be freed. Once freed, it writes a TAG_CREDIT packet into the send queue and updates the various pointers and counters, which in this case involves the sendcredit[] and recvcredit[] arrays.

One important note: Although it's bad programming practice, the folks who wrote FM apparently decided to send the amount of credits they are allotting into the packet.handler field. Thus, this is the field that a host should read if it receives a TAG_CREDIT packet.

Finally, the amount of credit allocated it just enough to give the receiver full send credits into our node.

A.3.2 consume()

This routine is really just a big switch statement. It takes a pointer to a struct RECVFRAME and passes the packet contained in the frame to the appropriate handler as determined by the packet.tag field as follows:

Case TAG_SEND_L: call the handler specified by the packet.handler field

Case TAG_ISO_MSG: calls iso_msg_handler()

Case TAG_CREDIT: no handler called here. Simply updates sendcredit[] array appropriately.

Case TAG_ISO_TOK: for the host to get this, we must be in HH mode. The packet is passed to iso_tok_handler().

Case TAG_ISO_EOP: packet passed to iso_eop_handler().

Case TAG_ISO_CHR: packet is passed to iso_chr_handler().

Case TAG_HELLO or TAG_HELLO_ACK: message is counted but ignored.

In each of the above cases, the sendcredit[] and recvcredit[] arrays are updated as appropriate, and if we note that recvcredit[i] is too low, the routine refill_credits() is called to give node i more credits. Note that unlike the situation with TAG_CREDIT packets, most other packet types piggyback credit updates in the packet.ackcount field. Note also that in the case of either TAG_ISO_TOK or TAG_ISO_EOP, no recvcredit is updated. Finally, if we receive a TAG_ISO_CHR packet, then my update the sendcredit to ourself:

sendcredit[FM_NODEID]++.

A.3.3 FM_extract()

Two variables here that are relevant are framesseen and framesdmaed. They are self explanatory. Basically, if framesseen equals framesdmaed, then we've handled everything we need to.

FM_extract() first checks this, and returns immediately if there is nothing to handle.

If, however, there are packets that have been DMAed but not seen, then FM_extract() calls consume() and then sets hrecvptr->valid to 0 (hrecvptr->valid is usually set to zero. It is set to one when a DMA that has been initiated has been completed).

A.3.4 send_last_packet()

This routine is passed a destination node, a handler, a pointer to a buffer, and a byte_count. It loads the buffer in the data portion of the packet in the sendqueue indicated by stailptr, and "sends" the packet (i.e. updates *hostsent with *hostsent = htonl(++myhostsent)).

What it does that is unique to last packets is not clear at the moment.

A.3.5 FM_reserve_credit()

This routine reserves enough send credits so that all the srefs in an isochron can be sent without any chance of blocking due to low send credit.

The array res[] is reloaded every time an isochron is ready to be sent from the smm to the SIU. Specifically, during a call to iso_end() (in smm.c) res[] is initialized to zero. Then the to_niu[] buffer in the smm is checked and the number of srefs that need to be sent to each node are determined, after which FM_reserve credit is called.

FM_reserve_credit() goes through each node and spins until sendcredit[i] is greater than res[i] for each i. While it's spinning, it's calling FM_extract() in an attempt to gain send-credits. Thus, when FM_reserve_credit() returns (if it returns) you are guaranteed that sendcredit[i] ¿ res[i] all i.

A.3.6 FM_iso_send()

Note this function is only called from iso_end() (i.e. you better be in an isochron when it's called).

In a nutshell, FM_iso_send() takes a struct net_sref (the part of an sref that goes out over the network) and builds it into the packet portion of a struct SENDFRAME in the LANai sendqueue. If the net_sref has flags set indicating that it is the last in an isochron, then this and all of the previous srefs from this isochron are sent (i.e. myhostsent is incremented, effectively telling the LANai to send them). Note that the routine really doesn't change any of the data in the net_sref it receives—it simply "wraps" the net_sref.

Understanding this routine requires understanding the purpose of the variable fm_in_isochron. When fm_in_isochron is zero, then we are not in the middle of sending an isochron. Once we are in the middle of sending an isochron, fm_in_isochron keeps track of the number of messages that have been sent in the isochron.

So, what does FM_iso_send() actually do? It takes as input a destination node and a pointer to a struct net_sref. If we are not in an isochron, then this must be the beginning of one, so the sendcredits to ourself are decremented to make room for the end of isochron marker that will inevitable have to be sent to ourself.

As always, the routine spins and calls FM_extract() until there is room in the sendqueue, at which time fm_in_isochron is incremented and the packet is built directly in the sendqueue. The latter includes updating the net_sref->head.flags field. If this field is equal to 0x2000, then this is the last message in the isochron, so *hostsent is updated, effectively sending the entire isochron, and fm_in_isochron is set back to zero (because we are now no longer in

an isochron). If the flags field is not 0x2000, then the packet is not sent yet (i.e. *hostsent is not updated yet). In either case, all relevant pointers (i.e. stailptr) and counters are updated.

A.3.7 FM_iso_send_token()

This routine is only used if we are running in HH mode. It takes as input a destination and flags, and places a TAG_ISO_TOK packet in the sendqueue. Documentation here indicates that there is work to be done on this routine. More on this later.

A.3.8 long_init()

This routine allocates a block of memory of size (sizeof (long long))* STATIC_BUF_SIZE * FM_NUMNODES bytes. It returns a pointer to the block that is stored in the variable mem_block.

A.3.9 H_first_packet()

Some background needed for this and the next two routines. FM can handle messages that are larger than the size of a packet. Before any of the routines below is called, fastmsgs.c sets up an area of static memory corresponding to each node, in which fragmentation and reassembly issues are handled. The variable mem_block is a pointer to the beginning of this block of memory. If messages are received that are too big for the statically allocated regions, then FM dynamically calls malloc() in an attempt to get a space big enough for the incoming message. Regardless of whether the space is allocated statically or dynamically, the array recv_buf[] gives pointers to the beginning of reassembly areas for each node, and the array fill_from[] provides pointers into these areas and indicates the next region in a given reassembly area that should be filled. If statically allocated memory is used for a packet, then the recv_buf[] values are given by

$$\text{recv\_buf[i]} = \text{mem\_block} + \text{source} * \text{STATIC\_BUF\_SIZE}$$

where STATIC_BUF_SIZE gives the size IN LONG LONGs of the static receive buffer of each node.

H_first_packet() is passed a pointer to a buffer, a size (in bytes), and the number of the source node.

It considers the buffer to be an array of long longs, and expects to find in the last element of the buffer both the address of the handler to use for the packet (which it stores in the array handler_size[], and don't ask me why it's called that) and the size in bytes of the entire message (which it stores in mess_size[]). It is the value of mess_size that determines whether or not the static reassembly arrays are used. Once the last long long in buf is read, the remaining contents of the buf array are placed into recv_buf[source] and the fill_from[source] pointer is appropriately updated.

Note recv_buf[source] is reset every time H_first_packet() is called because one never knows whether the static or dynamic reassembly areas will be used, and in either case, as mentioned above, recv_buf[source] must point to the correct reassembly area. Thus, with each call, the value of this pointer is reset to correspond either to the static case or the dynamic case.

A.3.10 H_middle_packet()

Since it is assumed that H_first_packet() will be called before H_middle_packet(), there is no need to set up the recv_buf[] arrays here. Like H_first_packet(), H_middle_packet() takes a buffer, a size and a source. It loads the buffer into the location specified by fill_from[source] and then updates fill_from[source].

A.3.11 H_last_packet()

This routine loads the contents of the buffer it is passed into the location specified by fill_from[], calls the message handler for the message (as recorded during the preceding call to H_first_packet()) and frees any dynamic space that may have been allocated during the handling of the message. (i.e. it does garbage collection).

A.3.12 FM_send()

This routine takes as input a destination node, a handler, a pointer to a buffer, and a size.

If the size of the message is less than PACKETSIZE, then the message is sent by calling send_last_packet(). If not, then FM-send() sets up the appropriate handler and size information in the last long long size space of the first PACKETSIZE size space in the buffer (see H_first_packet() to see how this works). Then FM_send() calls send_last_packet() with the handler function (in the call to send_last_packet() set to H_first_packet(). FM_send() then repeatedly calls send_last_packet() with the handler set to H_middle_packet() until it is left with a fragment of a packet, at which time it calls send_last_packet() with the handler set to H_last_packet(). Note that it is H_last_packet() that actually calls the handler specified in the call to FM_send().

A.4 smm.c

This is the module where the primary Isotach functionality is located.

Certainly a large part of understanding the operation of the smm is understanding its data structures. We discuss some of them here. For a better understanding of the abstract properties and purposes of these, it is best to first read John Regehr's Technical Report "An Isotach Implementation For Myrinet", available through the Isotach web page.

The first thing to know about the actual implementation is that different module of code in the Isotach prototype implement queues in different ways (three so far that I have counted). In smm.c, queues are implemented so that

head=tail–>empty and head=tail-1–>full

That is, the head always points to the next free spot on the queue (we add to the head), the tail always points to the next element to be removed from the queue (we take from the tail), and there is always one unused slot in the queue.

The most important abstract idea is that isochrons are composed of srefs. An sref (which I guess is an acronym for single reference) is a single reference to an isotach variable. Most isochrons will consist of a number of srefs. For example, a write to a shared variable needs to be considered an isochron (so all copies of the variable are updated at the same logical time). If, however, there are copies of the variable on nodes 1,2, and 4, then we need to send an sref to each of nodes 1,2, and 4 informing them of the update to the shared variable.

Some srefs in an isochron will involve shared variables for which our node has a copy. These are called local srefs. The srefs in the isochron that need to be executed on other nodes are remote srefs.

The HIT BUFFER: When an isochron contains srefs that need to be executed locally, these srefs are placed in the hit buffer.

In the implementation, the hit buffer, called hit_buf is an array of arrays. More precisely, each element in the hit_buf array is of type struct hit_array, which is to say it's an array of struct sref (a struct hit_array actually has three fields:

> remote — this is set to one if this particular hit_array corresponds to an isochron which contained remote srefs. If hit_array corresponds to a purely local isochron, then remote is set to 0 (these settings are done in iso_end()). This is important because purely local isochrons can be executed immediately if there are no outstanding isochrons (i.e. outstanding_markers = 0).
>
> padding — don't think this is used.
>
> buf — an array of struct sref.

The array hit_buf[i] corresponds to a single isochron, and hit_buf[i].buf[j] is the jth sref in the isochron represented by hit_buf[i].

Of some confusion here is the relation between the three variables hit_head, hit_cur_head, and hit_cur.

> hit_head—points to the next free hit_array in the hit buffer.
>
> hit_cur_head—points to the current struct hit_array on which we are working (i.e. it's current, but since we're working on it, it's not free anymore).
>
> hit_cur—points to the next free struct sref in the buf (array) field of the struct hit_array to which hit_cur_head is pointing. I.e. it points to the next free sref spot in the isochron on which we are working.

When we are setting up an isochron (done with a call to iso_start()), for example, one of the first things done in the code is

```
hit_cur_head = hit_head;
hit_cur = hit_head[0].buf;
if (++hit_head >= &hit_buf[HIT_BUF_SIZE]) {
    hit_head = hit_buf;
}
```

That is, we set hit_cur_head to point to what was the first free spot in the hit buffer, and hit_cur to point to the first free sref in the isochron pointed by by hit_cur_head (to be more readable, the second line of code above should read

```
hit\_cur = hit\_cur\_head[0].buf
```

Note the [0] is needed here because hit_head alone is just a pointer. The [0] essentially dereferences the pointer. So hit_head[0] is the struct hit_array to which hit_head points).

One more thing about the hit buffer: EVERY ISOCHRON, WHETHER HAVING A REMOTE COMPONENT OR PURELY LOCAL, IS PLACED IN A SPOT IN THE HIT

BUFFER. If the isochron is purely local and there are no outstanding chr markers (outstanding_markers = 0), then the isochron is executed immediately after having been placed in hit_buf, and that spot on the hit_buf is reclaimed (i.e. by adjusting the value of hit_head to point to hit_cur_head). If outstanding_markers != 0, then the remote field of the appropriate struct hit_array is set to zero, and the isochron has to "wait its turn" to be executed.

An important note regarding the effects of all of this on drain_buckets(): When a purely local isochron is issued, no corresponding SREF_CHR is placed in any pulse bucket! See notes on drain_buckets for an explanation of why this matters.

The PAGE TABLE: The page table is actually a misnomer. This data structure does contain copysets of pages, indicates whether or not our node has a copy of a page, and tells us where the nearest copy is if we don't. But it is more than that—it IS the actual memory. I.e. for any page of which our node has a copy, the actual values of the shared variables are stored in the page table data structure (as an array of struct iso_var32).

In the implementation, the page table is an array of struct page_table_ent. Each struct page_table_ent consists of the following fields:

loc — is the page local, remote, or invalid? loc is a variable of enum type page_status, which can take the values LOCAL, REMOTE, or INVALID.

copyset — this variable is of type copyset_t (i.e. an array of ints. It lists the pages on which the given page resides, with the node number of closest copy of the page in copyset[0], and the end of the list marked with a value of -1. (see comments in the read_mapfile() section below).

max_dist — the maximum distance to a copy of the page

page — a pointer to struct svar. Now, this is somewhat misleading. If our node has a copy of this page, then page is set to point to a contiguous block of variables of type struct svar by the call:

```
page_table[cur_page].page =
(struct svar *) malloc (sizeof(struct svar)*ISO_PAGE_SIZE);
```

Thus, page_table[i].page[j] is the contents of the jth variable of page i (where we start counting spaces at zero). This is set in the read_mapfile() routine. If this node does not have a copy of the page, then the page field is a pointer to NULL.

struct svar: This is an isotach shared variable. It consists of two fields: val (the value of the variable) and unsub (whether or not the variable is unsubstantiated). unsub is set to 1 only if the page is unsubstantiated (see John R's Tech Report for an explanation of unsubstantiated variables). When the memory for the page is allocated with the call to malloc above, all of the val and unsub fields are initialized to 0.

STRUCT SREF: By far the most complicated data type used here is the struct sref. An

sref consists of three parts:

net_sref — the part of the sref that actually gets sent out over the network

dest — the destination. A value of -1 indicates the sref is to be multicast. Like many other fields in the code, however, this gets misused at times. Specifically, when srefs are placed in pulse buckets, the value of the dest field is changed from our nodeid (obviously this was the destination to begin with since this node received the sref) to the value of the SOURCE of the sref. In the case of isochron markers (SREF_CHR), the dest field is set to our nodeid, since we sent it to ourself. In the case of EOP markers (SREF_EOP) the dest field is not used.

The above switch of the dest field from the destination node number to the source node number takes place in the routine iso_msg_handler().

copyset — the copyset variable is a pointer to int. It is only used if the struct sref needs to be multicast, in which case dest is set to -1, and copyset is set to point to the copyset (an array of integers) of the page of the variable being accessed by this sref.

A net_sref consists of an sref_header and an sref_body. The sref_header has the following fields:

ts – the logical timestamp

source

length

opcode — i.e. SREF_READ, SREF_WRITE, etc.

flags — used to determine whether we are at the end of an isochron, etc.

seqnum — used only in paranoid mode

The sref_body is a union type, and can be either a struct sref_mem, or a struct sref_msg.

A struct sref_mem has the following fields:

laddr — a pointer to a struct iso_var32. A struct iso_var32 has two fields, value and ready. This field seems to be frequently misused. In the read_pending list, this field is coerced to a pointer to sref and used as a pointer to the next entry in the read_pending list (see the detailed explanation in the section below on read_pending). When not being (mis)used as above, laddr actually points to a struct iso_var32. struct iso_var32 has a value field and a ready field. An explanation of the ready field is in order here.

Valid srefs are only created when an isochron is issued (to issue an isochron, iso_start() must be called. A call to iso_read32() or the like is really just a wrapper for a call to iso_op()) Within iso_op(), if the sref involves a read of any kind (either a straight read or as a read resulting from the execution of an SREF_ASSIGN), then the laddr->ready field will be involved. Basically, When the sref is first created, the value of this field is set to 0, indicating that an SREF has been set up, but not yet executed. At each point in the code where one of these srefs is executed, the "ready" field is changed to 1, indicating that the action associated with the given sref has now been completed (so I guess this prevents us from executing a given (read associated) sref twice). Note too that every time the ready field is set to zero, the counter outstanding_reads is incremented, and every time the value of the ready field is set to 1, outstanding_reads is decremented.

As for the value field of the struct iso_var32, this is still in some question. At times is seems like it really does hold the value of a variable. At other times, it's use is mysterious, as in during the call to read_handler() when the code section shows:

```
/* put the value in the proper location */
msg-$>$body.mem.laddr-$>$value = msg-$>$body.mem.val;
```

size — either SREF_32 or SREF_64. These don't really appear to be used for anything. Obviously they mean 32 or 64 bits. Right now only 32 bit sized variables are implemented.

val — the value of the shared variable.

shaddr — this is the "shared variable address". It is 4 bytes long. The most significant 22 bits give the page number, the least significant 10 bits give the offset into the page.

id — so far haven't been able to determine just what this is used for (it really doesn't appear to be used anywhere).

struct sref_msg: This is much more easy to understand. It consists of two field, an integer field that gives a handler number (i.e. specifies a handler) for the message, and an array of ULONG that represents the data.

The to_niu[] array: This is an array of struct sref. When we begin an isochron, the remote srefs are loaded into the to_niu[] array, beginning with the first spot. The entire array will be emptied during a call to the routine iso_end(), within which each sref will be sent to the NIU (network interface unit) with individual calls to FM_iso_send(). The variable to_niu_pos points to the next free spot in the to_niu array. Note that because of the mechanism described above, the to_niu[] array, at any given time, can contain only srefs from a single isochron. Why? Because the to_niu pointer is set to point to the beginning

of to_niu[] in iso_start(), and the array is emptied during calls to iso_end().

The BUCKETS array: buckets[] is an array of queues of srefs awaiting execution. Specifically, buckets[i][j] is the jth slot in the queue of srefs for bucket i. There is a bucket for each timestamp. The buckets are executed when the SIU indicates that it is time to do so. buck_head and buck_tail are both arrays whose entries are pointers to srefs. The ith entry of each of these arrays represents, respectively the head and tail of buckets[i]. That is, buck_head[i] points to the head of buckets[i], and buck_tail[i] points to the tail of buckets[i]. The initialization of the buckets array amounts to setting all of the buck_head and buck_tail entries pointing to the first entry of their respective bucket:

$$(\text{buck\_head[i]} = \text{buck\_tail[i]} = \text{buckets[i]}).$$

The READ PENDING list: read_pending[] is a queue of srefs that is implemented in a statically sized array. The "free list" part is determined via links through the "laddr" fields of the net_sref parts of these srefs. It works like this:

rp_free (a pointer to an sref) is initially set to point to the first entry in the read_pending array. This pointer from then on always points to the first entry in the free list. The other entries in the read_pending array have their opcode fields set to SREF_NONE:

$$(\text{p->msg.head.opcode} = \text{SREF\_NONE})$$

The laddr field of the net_sref is misused as a pointer to the next entry in the read_pending list:

$$(\text{p->msg.body.mem.laddr} = (\text{struct iso\_var *})(\text{p+1})).$$

The last entry in the array is initialized so that its laddr field starts as the NULL pointer.

tobe_exec[] array: This array is simply a list of SREFs that need to be executed (i.e. we have received them or placed them in the hit buffer but have not actually executed them yet). Srefs are placed on the tobe_exec[] by the routine iso_eop_handler() (and only by this routine). The srefs actually get executed when iso_poll() is called.

Concretely, tobe_exec[] is an array of struct tobe_t (a struct tobe_t is just a bucket field and a timestamp field). tobe_head and tobe_tail are INDICES, indicating the head and tail of the tobe_exec[] array. I.e. if tobe_head = 7 and tobe_tail = 3, then the head of the tobe_exec[] queue is tobe_exec[7], etc. The tobe_exec[] queue has elements added to it only during the routine iso_eop_handler(), which in turn is only called when the SMM receives an EOP marker from the SIU. tobe_exec[] keeps track of which buckets (if any) need to be drained. Buckets are drained only during a call to drain_bucket(), which is called only from iso_poll().

A.4.1 Barriers

Just some notes here to help me understand the workings of barriers. First, we have to realize that barriers are "understood" by different pieces of the system in different ways. I'll try to describe these here. Also note that there are two totally independent barriers implemented here (which is why just about all of the barrier data structures are arrays of size two). For this discussion, I'll assume that we have only implemented a single barrier bit (the two barrier bits work independently, so this simplification does not ignore any crucial

details). Also, when I write a "barrier token", that means a token with the barrier bit set to one.

If I'm a TM: Recall that at system startup, each TM and SIU sends an initial token wave to it's neighbors (neighbors in this context meaning the nodes with which tokens are exchanged). The initial token from a TM is a barrier token. Following this, we send a token wave of barrier tokens only if we have received barrier tokens from each of our neighbors.

If I'm an SIU: When I receive a barrier token, I increment my "barrier_send_credits". Initially, our barrier_send_credits are 0, but when we receive the initial token from our TM, since it is a barrier token, we have a single barrier credit (we can only send barrier tokens if we have barrier_send_credit. We increment barrier_send_credit when we receive a barrier token, and decrement it whenever we send a barrier token). Our host participates in a barrier (participates is a classic barrier term, but it is not a good one—a host "participates" in a barrier when it reaches the designated "stop and wait" point in the code) by sending its SIU a barrier/signal marker. The barrier "completes" (i.e. all hosts involved in the barrier are now free to continue) when the SIU sends the host an EOP marker with the barrier bit set. When a host sends a barrier/signal marker to the SIU, it also sends a "barrier count", which effectively determines how "long" the barrier will be. For example, if the host sends a barrier/signal marker to the SIU with the barrier count equal to 3, then the SIU (if it has at least one barrier_send_credit) will send a barrier token to its TM. When it receives the next barrier token from the TM, it will return a barrier token to the TM. It will continue in this way until it receives the third barrier token from the TM, at which time it will send the host an EOP marker with the barrier bit set.

There is an explanation of this Isotach specifications working paper by Craig. I kept forgetting little details, so I thought I'd fill them in here.

Another good reference is the V1.5 API (the V1 API doesn't really go into barriers in any detail). This lists the API barrier initiation calls. READ THE API—if you don't, then you're making more work for yourself. Once you understand the API for barriers, the purpose of many of the variables below becomes clear. In particular, understand iso_barrier(), iso_barrier_init(), and iso_barrier_poll().

Barrier data structures: They're all arrays because we've got more than one independent

barriers.

ISO_BARRIERS[i], ISO_SIGNALS[i]: These are just "pseudonyms" for the various barrier and signal bits (perhaps I should just call these enumerated variables). I.e. ISO_BARRIERS[0] = 0x01, so we can use the array instead of playing with the bit pattern. Likewise, ISO_SIGNALS[2] is the bit pattern 0x4.

The following group of variables is only used by TMs:

send_barrier_token[i]: When send_barrier_token[i] is set to one, then the next token wave we send should be barrier i tokens.

barrier_receive_clock[i][j]: The first subscript refers to the barrier number, the second to the port number. barrier_receive_clock[i][j] is the running count of the number of barrier i tokens we've received from our neighbor attached to port j.

barrier_send_clock[i]: The number of barrier i token waves we've sent to our neighbors.

The following are used by non-TMs:

barrier_token_credit[i]: If this is greater than zero, then we may send the SIU a barrier/signal marker with the barrier i bit set (saying a bit is set means that it is high). I.e. this is 1 if we have a barrier[i] credit.

barrier_token_outbound[i]: This is initialized to zero, and set to one whenever we initiate barrier i.

If in HH mode: During calls to enqueue_token(), if barrier_token_outbound[i] is high, then the routine knows to set the appropriate barrier bit in the outbound token. Immediately after the barrier bit is set, barrier_token_outbound[i] is reset to 0.

If in non-HH mode: FIXME! Not quite sure what happens here. Looks like I'll be the one implementing it, so I'll fill this in later.

barrier_count[i]: If barrier_count[i] is non zero, then there is already a barrier i in progress.

barrier_completed[i]: If barrier_completed[i] is equal to one, then it indicates that barrier i has completed, but that the user program has not yet been informed of this.

barrier_handler[i]:

A.4.2 SIGNALS

The primary data structures involving signals are:

next_signals:

receiving_signals:

received_signals[]:

A.4.3 read_mapfile()

This routine uses the "flex" package (as in Flex and Bison) to read the static copyset map file.

A few notes on Flex:

For those unfamiliar with Flex (or its predecessor lex), it is used to efficiently generate C routines that parse files and perform actions depending on the tokens encountered. For example, I might want a C routine that would read a file, pick out all instances of the word "Isotach", and then perform some command each time the word is found. To do this using flex (or lex) one writes a .lex file that essentially gives the rules for doing this, the rules being specified by a combination of regular expressions and corresponding commands. In any case, the C routine thus produced is named yylex().

Now, one can specify that every time a given token is specified, yylex() should return something. In general, when one does this, the token that caused the return is stored in the array yytext (an array of strings, or more correctly of characters, which is why the code for read_mapfile() uses atoi(yytext)).

Now, what does this have to do with read_mapfile()? Well, reading the mapfile is essentially parsing a file, so flex was used. The name of the file that is read here is shmem_map. The name of the flex specification file is called memmap.lex. Whoever wrote memmap.lex (I'm guessing it was John Regehr) specified that yylex() should ignore pound (#) signs at the beginning of a line, and that it should return the following:

Token:	Command:
any number	return TOK_NUMBER
:	return TOK_COLON
;	return TOK_SEMIC
,	return TOK_COMMA
-	return TOK_DASH
.	return ERROR

This corresponds to the definition in lex.h that creates the enumeration token_t type variable with the following line (in fact, the line below is essentially the entire content of lex.h)

```
enum token_t TOK_NUMBER=1, TOK_COLON, TOK_COMMA,
             TOK_DASH, TOK_SEMIC, ERROR;
```

Bottom line: when yylex() encounters one of the tokens above, it stops and returns the appropriate token_t. The next time it is called, it continues parsing the file shmem_map. In this way, the entire shared memory mapping file is parsed. Since there is no explicit instruction for what is returned when the entire shmem_map file has been parsed, I am guessing that based on convention, yylex() will return successfully (i.e. it will return 0). So as long as

$$mboxyylex() ! = 0$$

there are still some of the above tokens left to be scanned.

> If you find all of this confusing, simply read the first 8 or so pages of the O'Reilly book "lex & yacc". It's a quick read and it's very straightforward.
>
> I've included all of the above because you need to know it if you are planning on understanding the code in read_mapfile().

So now, what does read_mapfile() actually do? Well, it simply parses shmem_map and initializes the
page_table[i].copyset arrays. It does this in such a way that page_table[i].copyset[0] always contains the node number of the closest node to us that has a copy of page i (i.e. the first entry in the copyset array is always the closest with a copy of the page). The routine also initializes page_table[i].max_dist with the maximum distance to a copy of the page. If page i is not local, then page_table[i].loc is set to REMOTE, and page_table[i].page is set to NULL. If a copy of the page is supposed to be local, then page_table[i].loc is set to LOCAL, space is set up for the page, page_table[i].page is set to point to the allocated memory space, and the memory space is cleared.

One final note: although I haven't yet determined why, apparently the last page (i.e. page_table[MAX_PAGES-1]) is used for internal use, and it always resides only on node zero.

One other final note: the page_table[i].copyset[j] field is always initialized to -1. Another value other than -1 in
page_table[i].copyset[j-1] together with a value of -1 in page_table[i].copyset[j] indicates that there are j copies of page i in the system (a copy corresponding to each of 0,1,2,...,j-1). The locations of these j copies are given in the values of page_table[i].copyset[0] through page_table[i].copyset[j-1].

A.4.4 iso_initialize_global_state()

This routine does just what one thinks it should do— it initializes many of the global data structures.

These data structures and their purpose are listed below.

Steps 1 through 16 are only executed if we are in HH mode. Steps 6 through 10 are

done only if we are also a TM.

1. lt_buff[] is initialized to zero. lt_buff[] is the lost token buffer. It is only used in HH mode.

2. number_of_earlys is initialized to 0. This simply keeps a count of the number of tokens that we receive early. It is never decremented, so it is a running total.

3. number of repeats is initialized to 0. Similar to number_of_earlys.

4. token_q_head and token_q_tail, both pointers to type smm_token_t are initialized to NULL here. This is strange, because they are already initialized to NULL when they are declared. last_receive_time is initialized to zero, and current_signals is initialized to zero.

5. hh_send_clock is initialized to 2. This is apparently the send_clock when we are in HH mode. What I can't really figure out is why this is initialized to two here when it is already initialized to two when it is declared.

6. hh_recv_vec[] array is initialized. It I'm a TM, hh_recv_vec[i] gives the number of tokens I've received from neighbor with node number i. Note that for many i, neighbor i will not be a node from which we expect to receive tokens. If, however, we are supposed to receive tokens from node i, then the number of such tokens is hh_recv_vec[i]. Thus this could also be considered the logical time at node i. This array is only used if we are in HH mode AND we are a TM.

7. barrier_send_clock[i] gives the barrier send clock for barrier i (in the prototype, there are only two distinct barriers. There needs to be a send clock for each one, thus the need for the subscript i).

8. barrier_recv_clock[i][j] gives the barrier receive clock time for barrier i and port j (i.e. neighboring TM connected to port j).

9. hh_next_seq_num[]

hh_next_SEND_seq_num[]

10. init_token and init_token[i]. Both of these record whether we have sent an initial token to a particular destination. They are used in different submodes of HH mode. If we are a TM in HH mode, then we need to send an initial token to every neighboring TM or SIU (really host) that needs to receive tokens. We do this by sending tokens out to the appropriate ports on the switch to which we are attached. If we have sent an initial token to port i, then init_token[i] is 1. If not, then it's 0. If we are in HH mode but not a TM, then we only have one place to which we need to send an initial token (namely our tm). If init_token is 1, we've done this. If not, we still need to do it. In either case, the values of init_token and init_token[i] are initialized here.

11. next_signals and recieveing_signals are both initialized to 0x0000.

12. the recieved_signals[] array is initialized to 0.

13. the following arrays are all initialized to 0.

    barrier_token_credit[]

    barrier_token_outbound[]

    barrier_completed[]

    barrier_count[]

14. barrier_handler[] is initialized to NULL.

15. hh_recv_clock is set to 1.

16. seqNum is set to 0, seqSENDNum is set to 1, and init_token is set to 0.

17. The hit buffer is set up. hit_head and hit_tail are initialized and the msg.head.opcode fields of each sref are set to SREF_NONE.

18. Initializes outstanding_markers, markers_in_transit, outstanding_reads, etc.

outstanding_markers and markers_in_transit:

Bottom line: A marker is outstanding from the time an isochron is sent in the call to iso_end() until the time that the bucket containing the corresponding SREF_CHR is drained during a call to drain_bucket(). Since draining a bucket means executing the all of the isochrons corresponding to a given pulse component of logical time, when outstanding_markers is equal to zero, we are assured that all previously issued isochrons with remote components have been executed.

Let me just say that determining exactly what these two variables represent was quite a job. It took me a good deal of time to figure it out— be glad you don't have to do it. So, here's the story:

When an isochron is ready to be sent (during a call to iso_end(), outstanding_markers and markers_in_transit are both incremented. Now, the sending of the isochron occurs through multiple calls to FM_iso_send(). When FM_iso_send() is called, it builds a packet directly in the LANai send queue. The packet's tag field is set to TAG_ISO_MSG. Now, when the routine send_messages() is called (send_messages is in the main event loop of lcp.c. It gets called when there are messages to be placed on the wire.), if it encounters a packet with a tag of type TAG_ISO_MSG, and if the LANai presently does not think we are in an isochron, it sets its in_isochron variable to 1, and sets send_mark to 1. Setting send_mark to 1 has the effect of making send_messages() write a packet with a tag field of TAG_ISO_CHR directly into the receive queue on the host. I.e. whenever there is an isochron being placed on the wire, there must be an isochron marker returned to the SMM, and the LANai creates this isochron marker as soon as it knows that an isochron has begun (to be specific, a non purely local isochron, since obviously if the LANai is placing it on the wire, there is a remote component).

Now, when a call is made to consume() (in fastmsgs.c), if it runs across a packet with tag TAG_ISO_CHR, it calls iso_chr_handler() and passes it a timestamp (which is contained in the "handler" field of the packet, another gross misuse of a data field — for a detailed explanation of how the timestamp value is determined, see the explanation in the section on iso_end() in smm.c)). iso_chr_handler() then places an sref of type SREF_CHR in the bucket corresponding to the timestamp value. It is at this point that markers_in_transit is decremented. Thus, apparently, a marker is in transit from the time that an isochron is sent at the end of the call to iso_end() to the time that the SREF_CHR is placed in the correct bucket. A marker is outstanding from the time an isochron is sent in the call to iso_end() until the time that the bucket containing the corresponding SREF_CHR is drained during a call to drain_bucket().

outstanding_reads: apparently this variable keeps track of the number of outstanding reads, i.e. the number of reads that we have requested but for which we have not yet received the value of the variable that was requested in the read. This is decremented whenever we handle a returned read (i.e. when the requested value arrives).

max_outstanding_reads: this is apparently a system analysis variable. It keeps track of the largest value that the variable outstanding_reads has reached during system execution.

19. The read_pending[] list is initialized.

20. the tobe_exec[] array and pointers to it are initialized.

A.4.5 iso_init()

iso_init() performs the following operations:

1. Checks that the relative sizes of struct net_sref, PACKETSIZE, and similar objects are correct.

2. Calls FM_initialize()

3. Calls the standard library function signal() with the signal set to SIGINT (SIGINT is activated if someone presses a key on the keyboard). The handler specified is handle_sigint(), which simply prints a message to stderr, calls iso_print_stats() and exits with a status of 1.

4. calls FM_set_handlers() to set 5 to read_handler() and 6 to response_handler(). read_handler() is the handler for returned reads (i.e. when we've requested a read from a shared variable, and the result of the read has arrived). It places the value received in the correct place in shared memory. resp_handler() is a handler for paranoid responses. I wouldn't worry about it at this point.

5. Initializes the iso_msg_handlers[] and iso_signal_handlers[] arrays to NULL.

6. Calls read_mapfile(). (see above)

7. Calls iso_initialize_global_state(). (see above)

8. Calls iso_poll()

A.4.6 enqueue_first_token()

Summary: initial struct smm_token_t is set up and placed in the token queue.

Apparently two tokens are sent at startup. Because this is somewhat awkward, the first of the two tokens are sent by this routine. To be correct, nothing is really "sent" here at all. Instead, they are enqueued. All this routine does is construct a struct smm_token_t and then place it on the token queue. The routine is passed two input parameters: a destination and a source port (the latter in the form of the six bit source port field used by hh_source_port_send(), etc.). First, the source port field of the token is set. If we're a TM, then the two barrier bits are set (because the first token issued by a TM should have both barrier bits set). The "token_flags" field here of our token represents the two BYTES of the token (see Isotach design doc.) it gives source port and barrier bit information (and this should explain why there is the need to use the left shift 8 operator). The "token_flags" value is placed in the lost token buffer in entries 0 and 1 (i.e. lt_buff[0]=lt_buff[1]=token_flags).

A.4.7 iso_start()

The purpose of this routine is to set up for the execution of an isochron. It takes no input parameters.

It sets in_isochron to 1, iso_dist to 0, and sets to_niu_pos to point to the beginning of the to_niu array.

It sets hit_cur_head, hit_cur, and increments hit_head (see comments on these above).

A.4.8 iso_read32()

This is really a "wrapper" function. It takes as a parameter a shmem_addr_t (called shaddr—note that shmem_addr_t is just pseudonym for unsigned long, which on our machines are 4 bytes) and a pointer to a struct iso_var32 (called laddr) and then executes the call iso_op(SREF_READ, shaddr, laddr, NULL);

A.4.9 iso_write32()

Like iso_read32(), this is a "wrapper" function. It takes as a parameter a shmem_addr_t (called shaddr) and a pointer to void (called wvar—I guess short for write value) and then executes the call

iso_op(SREF_WRITE, shaddr, 0, wval);

A.4.10 iso_end()

This routine does the dirty work needed to end an isochron and then send it. It does:

1. Determines how much send credit to reserve by scanning the contents of the srefs in to_niu[] (it does account for multicast in this reservation scheme). This info is loaded into the res[] array (see above comments on FM_reserve_credit() in fastmsgs.c). It reserves the credit with a call to FM_reserve_credit().

2. Goes through to_niu[], counts the number of messages (srefs) that need to be sent (msg_count), and sets the flags field of the last sref in the isochron to the value 0x2000 (used by FM_iso_send()—see discussion in the fastmsgs.c section). Every other sref has flags set to 0.

3. Sends the individual srefs (actually only the net_sref parts of them) with calls to FM_iso_send().

4. Increments outstanding_markers and markers_in_transit, and sets in_isochron to 0.

This routine also determines the timestamp that the isochron marker and all the outgoing srefs will receive (the SIU actually determines the timestamp, but it depends on a value set up in iso_end()). Specifically, the timestamp value (in non HH mode) keys on the value of the variable iso_dist. iso_dist is set to zero in the call to iso_start() that initiates the isochron, and subsequently incremented if necessary during each intermediate call to iso_op(). When iso_end() is called to end the isochron, the current value of iso_dist is written into the head.ts field of all of the net_srefs leaving the to_niu array. These net_srefs are passed to FM_iso_send() which wraps them, but doesn't change the contents. It passes the newly created packets (which wrap the net_srefs) to send_messages() (in the LANai code). When send_messages() receives a packet representing the first sref to be sent from an isochron, it looks at the head.ts value. The LANai code variable last_receive_time is then set to max(send_clock + head.ts, last_receive_time). This value, in turn, is written into the (misused) handler field of the TAG_ISO_CHR packet that is written into the host's receive queue (i.e. it's the time stamp).

A.4.11 iso_op()

This is a generic shared variable routine. To skip the details, just know this: It takes (among other things) an sref as input, links it into either hit_buf or to_niu or both and fills in the appropriate fields of the sref (i.e. opcode, dest, etc.)

Precondition: iso_op() can only be called from within an isochron (i.e. in_isochron=1).

Another key component of the smm. This routine is called within the setup of a single isochron, its purpose being simply to set up an sref and link it into the appropriate data structures. Specifically, it takes as input the op code for the proposed sref, the address of the shared variable involved,

the local address (laddr) of the variable involved, and the

FIX ME—I'm not exactly sure what the laddr field represents or exactly why we use it here.

write value (wval) if a write is involved. Given these parameters, the routine creates an sref with the appropriate characteristics for the given operation. Note that wval is not used at all if the particular operation does not involve a write.

The details are as follows:

First the page and offset of the shared variable are extracted from the shaddr input parameter. (see comments on the shaddr field in the description of struct sref above).

Next, it is determined whether or not the page is local (by looking at page_table[page].loc).

If local: the new sref is linked into the appropriate spot in the appropriate element (i.e. the hit_array corresponding to this isochron) in hit_buf.

If remote: the new sref is linked into a spot in the to_niu array.

One note: in the code for iso_op() there are two temp storage variables, thisref and ref, which are pointers to struct sref. Either or both of these are used depending on the operation specified. Roughly, thisref refers to the sref that is to be linked into hit_buf, while ref refers to the sref that is to be linked into the to_niu array. (That's not always quite true, but close enough for a good understanding of this). Two such temps are needed because, obviously, various fields of the sref change depending on whether it is going to be executed only locally, only remotely, or both locally and remotely (as in the case of some writes, scheds, and assigns).

After the appropriate linking is done, the actual "filling in" of the sref occurs: the opcode field is set, the shaddr field is set, etc.

Next, the operation specific actions are taken: Note that in the case of a read, the sref goes into either the hit buffer (if there is a local copy of the shared variable) or the to_niu buffer (if the variable is kept on a page that does not reside at our node), but never into both. In the case of a write type operations (write, sched, or assign) the sref may have to be linked into both of these data structure. Also note that no matter what, only one copy of an sref gets linked to the to_niu array, as multicast is done later by repeatedly calling FM_iso_send().

case SREF_READ: the "ready" field is set to zero (thisref->msg.body.mem.laddr->ready = 0). If it's a local read, then the sref destination field is set to our node id. If it's remote, then the destination field is determined from the page table (specifically page_table[page].copyset[0], which has been configured to always list the closest node with a copy of the given page).

case SREF_WRITE, SREF_ASSIGN, SREF_SCHED: In the first two cases, the msg.body.mem.val field of the sref is set to wval (stands for "write value"). This is not done

for SREF_SCHED for obvious reasons. For all three the rest of the procedure is the same:

We check to see if there is a copy of the relevant page on any other node. If not, then since it must be somewhere, it must be on our node, so we set the dest field of the sref to our node id.

If there is another copy of the page somewhere, then we check the location of the page. If the page is marked as local, then we will have already linked thisref into the hit buffer from the linking done at the beginning of the routine. What we need to do now is to copy thisref to ref and link ref into the to_niu buffer. This is done, after which thisref->dest is set to our node id (because thisref is coming to us), ref->dest is set to -1, indicating that it is to be multicast, and ref->copyset is set to the copyset value of the shared variable (page_table[page].copyset) so that we know the nodes to which the multicast must go. If, on the other hand, the page has not been marked as local, we have already linked thisref into a spot in the to_niu buffer, but the sref needs to be modified: we set its destination to -1 because it needs to be multicast and load the copyset into the sref.

Finally, the variable iso_distance is updated to reflect the possible change in the max distance that this isochron must travel.

A.4.12 iso_poll()

This routine is considerably less involved if we're not in HH mode and not a TM. Assuming then, that we are simply a host running in non-HH mode, iso_poll() does the following:

1. calls FM_extract()
2. Drains all of the buckets linked into the tobe_exec[] queue.

If we are in HH mode or are a TM, then there are barrier issues with which we need to deal.

A.4.13 drain_bucket()

Shortest summary of this routine: it takes a bucket number and a timestamp as inputs. It drains the corresponding bucket. What that means is that it drains what is in the bucket, and it knows it has emptied the bucket when it hits an SREF_EOP. If along the way it hits an SREF_CHR, then it pauses draining the bucket, and instead empties the hit buffer, executing exactly one isochron with a remote component, and as many purely local isochrons as it encounters up to the second remote isochron. It then returns to emptying the bucket. The cycle continues until it hits the SREF_EOP.

A slightly more detailed summary: When draining a bucket, we continue until we hit an SREF_CHR (isochron marker). How is all of this coordinated? Well, every time an isochron with a remote component is issued, it causes an SREF_CHR to be placed in a pulse bucket AND the pointer hit_head to the hit_buf is incremented. So roughly, each SREF_CHR corresponds to a hit_array in the hit_buf. There are, however, hit_arrays in hit_buf that correspond to purely local isochrons (that couldn't be immediately executed at their issue time since there were outstanding CHR markers). So, when draining a bucket, when an SREF_CHR is encountered, the hit_buf can be drained until we hit the SECOND hit_array in the hit buffer with a remote component (i.e. with the remote field set to 1). The first hit_array with remote = 1 will correspond to the SREF_CHR encountered in the pulse bucket, so it can be executed. Anything with the remote field set to zero can also be

executed. Once the second hit_array with remote = 1 is encountered, however, we cannot execute it (until we run across another SREF_CHR in the bucket).

Running across an SREF_EOP in a bucket, of course, should mean that there are no more srefs in the bucket (since SREF_EOP says the logical time pulse has ended).

A number of points are of interest here. First, the two dimensional vector[] array, whose elements are sref pointers, is merely a place to keep the sorted srefs. The first index corresponds to the nodeid of the source host and the second index corresponds to the position of the sref (after sorting) among those srefs received from a given source (i.e. vector[4][5] would be the sixth sref in the sorted srefs in this bucket that we received from node 4.) vec_count[i], on the other hand, is the number of srefs from source i in this bucket. It is initialized to zero and then incremented as the routine works through the given bucket. So basically, the routine goes through the bucket (by starting with the sref pointed to by buck_tail[i] and incrementing buck_tail[i] until we hit an SREF_EOP. (remember, buckets[i] is a queue, and in smm.c, we remove entries from the tail of queues)) and sorting them based on source id and rank (assuming FIFO delivery of point to point messages through the network). Note that since all of the pulse components of timestamps in a given bucket are the same, this above mentioned sorting on source id and rank is all the sorting that is necessary.

The variable seen_remote is used simply to keep track of whether or not we've seen a remote isochron. Each time an SREF_CHR marker is encountered in the bucket, seen_remote is set to zero just before entering the "drain some of the hit buffer" stage. While draining the hit buffer, when the first remote isochron is encountered, seen_remote is set to 1. This ensures that only the correct group of isochrons from the hit buffer is executed for each SREF_CHR encountered.

The variable "check" simply keeps track of how many srefs we've removed from the bucket, so we can check that after sorting we still have the same amount of srefs. It's basically a sanity check.

One note on style: the C hack technique of updating the value of a variable by using it in an expression in a conditional is used here quite a bit. For example, the statement partial statement:

```
if (++buck\_tail[bucket] >= &buckets[bucket][BUCKET\_SIZE])
```

Note that whether or not the guard condition evaluates to true, the pointer buck_tail[bucket] is ALWAYS incremented. It makes the code more concise, but definitely causes a problem with readability. Also, (for my benefit as one who is relatively new to C) note that the construct ++buck_tail[bucket] is really unambiguous – square brackets have a higher precedence than the unary operator ++, so this is really ++(buck_tail[bucket]), which, since buck_tail[bucket] is a pointer to sref, just causes buck_tail[bucket] to point to the next element of the queue representing buckets[i].

The actual execution of srefs takes place through calls to either execute_sref() (to actually execute an sref in the bucket) or execute_hit_buf() which takes care of the srefs that need to be executed during the "drain some of the hit buffer" cycles.

Note also that the variable outstanding_markers is decremented essentially every time we hit an SREF_CHR. (It's actually decremented just before the corresponding remote isochron in the hit buffer is executed).

A.4.14 execute_sref()

The short summary: this routine executes an sref. The only complicated cases involve unsubstantiated reads and their complications. That is, an SREF_READ on an unsubstantiated value causes the sref to be place in the read_pending list. SREF_ASSIGNS cause the read_pending list to be checked, which may result in the execution of additional srefs.

Remember here: the dest field of net_srefs here don't represent destination nodes, but instead represent source nodes (because, as mentioned in the section on drain_bucket(), we change the dest field before placing srefs in buckets (or the hit buffer).

execute_sref is an important routine for understanding the workings of the shared memory manager. The input to the routine is a pointer to an sref. As always, the page and offset values are determined by bit masking the value of the s->msg.body.mem.shaddr field. A switch statement handles each possible opcode option.

SREF_READ: Two cases here: the variable to be read is either unsubstantiated or substantiated.

unsubstantiated: the sref is placed on the read_pending list (in the first free spot as determined by rp_free, which is then updated to point to a new free spot). Before being placed on the read_pending list, however, the s->msg.head.flags field is set equal to the val field of the shared variable (specifically page_table[page].page[offset].val)

substantiated: again two cases: we requested the read (i.e. s->dest is our node id), or we didn't.

we requested it (local): The value of the shared variable is placed into the value field of s:

s->msg.body.mem.laddr->value = page_table[page].page[offset].val;

and the value of outstanding_reads is decremented.

not our node (remote): the s->msg.body.mem.val field is loaded with the value of the variable (page_table[page].page[offset].val). The sref is then sent to its destination by a call to FM_send().

A general comment here: apparently, when a read is executed, the value of the shared variable is placed in the msg.body.mem.laddr->value field, but when the sref is loaded but not executed, the value of the variable in question is placed in the msg.body.mem.val field instead.

SREF_WRITE: the value of the variable is transferred from the s->msg.body.mem.val field to the page table (page_table[page].page[offset].val = s->msg.body.mem.val) the unsub field of the shared variable in the page table is set to 0, indicating that this shared variable is substantiated. (a shared variable is always substantiated following a write to it).

SREF_SCHED: the unsub field of the shared variable is set to 1, indicating that it is now unsubstantiated, and the value of the source node for the SREF_SCHED is placed in the val field of the shared variable. (page_table[page].page[offset].val = s->dest). This last moves ensures that only the node that caused the sched can "resolve" it with the corresponding SREF_ASSIGN. It also assures that WHEN A SHARED VARIABLE IS UNSUBSTANTIATED, THE VALUE IN THE

page_table[page].page[offset].val

FIELD IS THE NODE ID OF THE HOST THAT INITIATED THE SREF_SCHED THAT CAUSED THE VARIABLE TO BE UNSUBSTANTIATED.

SREF_ASSIGN: First, there is a check to ensure that both the variable in question is unsubstantiated and that the node requesting the SREF_ASSIGN is the same node that initiated the corresponding SREF_SCHED.

(Interesting aside (and question): What happens if one node initiates an SREF_SCHED on a shared variable, and before that variable is SREF_ASSIGNed another node attempts to initiate another SREF_SCHED? From the code here it looks like the first node is out of luck— that is, the second SREF_SCHED overwrites the first, because there is no check to ensure that a variable is substantiated before the SREF_SCHED is done. We might want to change this).

Then the value in s->msg.body.mem.val is loaded into the value of the shared variable in the page table, and the unsub field of the shared variable is set to 0. The read_pending list is searched, to see if there are any matches. This search is interesting because of the way it checks for a match. It goes through the read_pending list looking for SREF_READs (it's possible some spots in the read_pending list will be SREF_NONE) with the same address as the shared variable, and checks whether the msg.head.flags field the sref in the read_pending list matches the dest field of the SREF_ASSIGN. Why should this be? Well in the convoluted scheme used here, remember that SREF_ASSIGN->dest is the source node id. Also remember that when an SREF_SCHED is done, the source node id is placed in the

page_table[page].page[offset].val

field (see note above). When a read is attempted on an unsubstantiated variable, the value of that variables

page_table[page].page[offset].val

field is placed into the

msg.head.flags

field of the entry in the read_pending list. Thus, checking whether

p->msg.head.flags == s->dest

is checking whether this particular ASSIGN is resolving the SCHED that caused the SREF_READ in the read_pending list to be placed in the read_pending list in the first place.

Now, if we do get a match with an sref in the read_pending list, then we handle it like a substantiated SREF_READ (i.e. it's either a read for us, or it's a read requested by a remote node, etc.). The last thing done here is some updating of pointers to keep the free list of read_pending current.

SREF_MSG: In this case we get a handler by calling iso_get_handler() with s->msg.body.msg.handler as input. We then execute the handler that is returned, if we are returned a valid handler.

Once exactly one of the above options has been followed, the opcode field of the sref s is set to SREF_NONE.

A.4.15 execute_hit_buf()

Execute hit buf accepts an sref and a timestamp as inputs. It causes a single row of the hit buffer (i.e. a single isochron) to be executed. It is implemented by having the timestamp loaded into the msg.head.ts field of each of the isochron's srefs (which makes sense because srefs in the hit buffer shouldn't have timestamps—we don't know when they should be executed until we're draining pulse buckets), and then they are individually executed with calls the execute_sref().

FIX ME!—a question here: why load the timestamps into the srefs just before executing them? It must be a debugging thing, because once execute_sref() is called, the srefs are getting executed. Also, execute_sref() doesn't seem to use the timestamps anywhere. Maybe it has something to do with scheds and assigns? In fact, it is exactly assigns that should need these timestamps—when a sched is done, it is reserving a slice of logical time in which to write to a variable. Without using the timestamps, you really can't distinguish one sched from another. This might be a way to allow two different "unresolved" scheds on the sam variable at the same time.

How would this be implemented? Well, when an SREF_SCHED is executed, it obviously comes from a pulse bucket. We thus know, from the pulse bucket, what the timestamp of execution of the sched should be. This information should somehow be included with the unsub information when the variable in question is changed to unsubstantiated (i.e. maybe instead of having unsubstantiated = 1, have unsubstantiated = timestamp of SREF_SCHED). Then when an SREF_READ is attempted on an unsubstantiated variable, before it's placed in read_pending, the timestamp of the SREF_SCHED gets loaded into some field in the SREF_READ. Then later, when an SREF_ASSIGN comes along, based on the source ID of the SREF_ASSIGN, we find the timestamp of the associated SREF_SCHED, and while checking the reading pending list for SREF_READS, we checks not only for all the stuff that is presently checked, but also whether the timestamp for the SREF_READ matches the timestamp for the appropriate SREF_SCHED.

A.4.16 iso_eop_handler()

Before being able to fully understand this routine, one must become familiar with the path taken in the generation of an SREF_EOP. As always in this code, there are fields that are misused. Now, SREF_EOP are generated as follows: First, the lcp.c routine receive_messages() receives a token (it knows it's a token because only tokens are 5 bytes long—when it receives a message of this length, it checks to see if it's a token). receive_messages() then writes a packet into the host receive buffer. This packet has tag TAG_ISO_EOP, and loaded into its handler field is the value of the LANai receive clock minus one (recv_clock - 1). Why the -1? Well, I'm not exactly sure. When the SIU receives a token, it automatically advances its receive clock. BUT, it only does this AFTER loading the packet in the host receive queue. So the time corresponding to this EOP should be recv_clock. Maybe it has to do with arrays starting at 0 (since there would be a pulse bucket 0?). In any case, the important thing here is that the timestamp, such as it is, is placed in the handler field, I guess because packets have no timestamp field. When FM reads the packet out of the queue (in a call to consume(), it sees the TAG_ISO_EOP tag and know that it should call iso_eop_handler(), which is does, passing the contents of the handler field as input parameter. When iso_eop_handler() is called, it creates an SREF_EOP and places

it in a bucket, the bucket number being determined by the timestamp that was passed and the BUCKET_MASK.

A.4.17 iso_read_handler()

This routine takes a buffer as input (actually, it's just an sref), then places the value of the read "in the proper location". This means doing the usual

msg->body.mem.laddr->value = msg->body.mem.val

move. The only other line of any consequence is the line

msg->body.mem.laddr->ready = 1

which performs the usual "ready" variable manipulation for a read.

A.4.18 enqueue_token()

enqueue_token() does just what it's name implies. It only exists in HH mode, (at least in all of the current versions of the software). It takes as input a destination, a source_port, a seqnum, and an integer flag (called remember). The routine simply places the info it is passed into the correct bits in the token, then enqueues the token (on the token_q). If remember is nonzero, it moves the flags of the current token into lt_buff[1], and the flags of the previous lt_buff[1] into lt_buff[0]. The routine also sets the appropriate signal and barrier bits if it is the correct time to do so (i.e. the beginning of an epoch for most signal bits). Note also that signals are set in calls to the API functions iso_send_reset_signal() and iso_send_signal().

Since the lt_buff[] array is not explained well elsewhere in this document, here is the explanation. We need to possibly store two tokens to implement the lost token algorithm. Specifically, we need to "remember" both the last token sent, and the one previous to that. lt_buff[1] always stores the last one sent (technically the last one enqueued) and lt_buff[0] stores the one previous to that.

NOTE: iso_barrier(), iso_barrier_init(), and iso_barrier_poll(), are only used by non-TMs.

A.4.19 iso_barrier()

This code initiates a blocking barrier. It takes an integer (count) as input. count determines the logical time length of the barrier.

Checks that we do not still have a 0 barrier in progress (and that if we've just completed one, the user program knows it).

A.4.20 iso_barrier_init()

A.4.21 iso_barrier_poll()

A.4.22 lost_token_handler()

This routine simply enqueues the last two tokens sent (actually, it enqueues the last two tokens previously enqueued—FM_iso_send_token() does the real sending). It first checks lt_buff[0] and lt_buff[1] ti be sure that something has been sent since the last reset, and if so, it enqueues the two tokens, with lt_buff[0] going first.

A.4.23 iso_send_message()

A.4.24 iso_msg_handler()

A.4.25 iso_tok_handler()

A.4.26 iso_chr_handler()

A.4.27 iso_send_reset_signal()

A.5 Comments on various critical paths

I thought it would be nice to have a general overview of what exactly the control sequences look like. Specifically, it is easy to see what each function does from a "microscopic" perspective by looking at the descriptions above. Here, I hope to give a more macroscopic perspective.

A.5.1 lcp.c

The basic control loop here is simply:

```
while(1)
{
        read_messages();
        if (something needs to be sent)
        {
                send_messages();
        }
}
```

Depending on whether the node under consideration is a TM and/or we're in HH mode, we might have to also send_tokens().

read_messages() simply takes packets off the wire and places them in LANai SRAM, checks CRC and packet types (and modifies a few fields if necessary), then DMAs them into the host receive queue.

send_messages() sends a single packet, giving it a timestamp if necessary. If it has to spin waiting for the send to complete, it calls receive_messages().

A.5.2 fastmsgs.c

consume() takes a packet as input and passes it to the appropriate handler (which could be iso_eop_handler(), iso_tok_handler(), iso_msg_handler(), iso_chr_handler(), among others).

FM_extract() consumes all packets that have been DMAed to the host but not yet processed (i.e. it looks at the new stuff on the receive queue). It's simply a wrapper for a bunch of calls to consume().

FM_send() is really a misnomer—send_last_packet() is the real sending function for FM. FM_send() is necessary only to handle packet fragmentation. If the size of a message is less than PACKETSIZE, FM_send() just calls send_last_packet(). If not, then it fragments the packet, calling send_last_packet() first with the handler input set to H_first_packet() (for the

first fragment), then with handler set to H_middle_packet() (for the middle fragments), and finally with H_last_packet() (for the last fragment).

send_last_packet() simply takes a packet as input and write it directly into the LANai send queue, updating *hostsent to tell the LANai to send it when ready.

FM_iso_send() is used to send most Isotach packets. It takes a struct net_sref as input, tags it with TAG_ISO_MSG, sets flags, copies it directly into the LANai send queue, then updates *hostsent (the LANai code variable that governs when things are to be sent). An interesting note is that it completely bypasses FM_send(). This routine is only called from iso_end().

When is iso_poll() called?

> It is called during iso_init(), possibly more than once since it is called, it appears, whenever we are waiting on a barrier. It is called from iso_barrier(), again while waiting for a barrier to complete. It is called from iso_deinit(), again while waiting for a barrier, and it is called from the main() routine if we're a TM in non HH mode. The primary call to iso_poll() (i.e. the recurring somewhat periodic call) is from iso_end(). Thus, if we're not sending isochrons, then we're not generally polling.

A.6 An explanation of the whole packet header maneuvering saga

If one looks in the lcp.c and lcp.h (as well as a few other places), one will notice that there is a lot of code that deals with struct ISOPACKETS, and talks about being sure that sizeof(struct ISOPACKETS) is the same as sizeof(struct PACKET). What is this all about? Well, the original isotach prototype was implemented entirely in software. The data structures that conveyed isotach information contained all of the same material required for a prototype with added H/W components (i.e. the H/W TM and H/W SIU), but they were not set up in a format that met the isotach specifications (see Craig's working specification document). Some of the problems were:

1. A H/W SIU expects to see a specific isotach header on isotach messages generated by the corresponding host. The SIU consumes this header, and then passes the remainder of the packet to the network (thus routing bytes must immediately follow the isotach header). Following the routing bytes are a few more bytes of isotach info needed by the H/W SIU at the destination, which reads these bytes to get info on timestamps and the like. The destination SIU does not remove the additional isotach info bytes but instead passes the entire packet (without modification) directly to the destination LANai. Now, the isotach code running on the LANai is a modification of the FM 1.x LANai code, and is thus set up to handle FM packets, which have a different format than isotach packets. In particular, FM expects to see an FM tag as the first two bytes of a packet, etc. the LANai FM code uses this and other FM header information when passing the packet into the FM receive queue on the host. Now, isotach is intertwined with FM (and FM gives us some non isotach functionality as well), and I didn't really have the time (at this point) to completely implement isotach without any traces of FM, so I wanted to keep most of the FM flow control and non-isotach message passing functionality. This meant somehow modifying the LANai code to handle both packets with the isotach format and packets with the FM format, but having BOTH

packet types placed in the FM host receive queue (since it is a staging area for some of our isotach functionality). A final issue is that FM LANai code expects all messages to be of length equal to the size of a struct PACKET. This could be worked around by modifying the LANai code to handle both FM packet size and isotach packet size messages, but that would have meant fooling around with a lot of the nice FM data structures on both the LANai and the host. It also meant that on each message receive, the LANai code would first need to read the two initial bytes of the message into LANai SRAM, look at the packet type in these bytes and then decide how many bytes the rest of the message was (note this would have required either two distinct receive queues on the LANai or some fancy pointer stuff to deal with queues with two different sized elements). The other option was simply creating an isotach packet type with the same size as an FM packet. Then queue management remains relatively simple, and there is no question about how many bytes must be read into the LANai on a message receive.

In short, what I needed was a system satisfying the following:

1. It had Isotach packets and FM packets, and both were the same length.

2. Each isotach packet had an FM packet embedded in its payload field (so that all the info FM needed was still in there somewhere and in the correct format).

   Now this, of course, is a fundamental problem: how can you have one data structure embedded inside another and yet both are the same size.

Well, there's more to the story. When we send an sref in an FM packet, we send it in the payload ("data") field of the FM packet. Specifically, we load a struct net_sref into the data array field of an FM struct PACKET. It is the info in this net_sref that isotach uses. Now, a net_sref is smaller than the data field of an FM packet (by at least 24 bytes — the size of a net_sref is determined at compile time by how the system is configured (i.e. in paranoid mode, etc.). Also, the extra info required in an Isotach packet is less (but not much less) than 24 bytes. So the idea is to simply fit a shortened FM packet into an isotach packet. The "shortened" FM packet (called struct FM_SHORT_PACKET) is identical to an FM packet, except that its data field is smaller— big enough to contain a net_sref, but small enough so that the shortened FM packet fits into the payload field of an Isotach packet and makes the size of an Isotach packet equal to the size of a regular FM packet.

So now how does all of this work? Well, when sending a regular FM message (non-isotach), nothing changes. But when sending an isotach message that requires isotach headers (note not all isotach messages do require these—for example returning the value of a read to a remote node does not require isotach info because it doesn't need to be timestamped) we load the appropriate info into the fields in the ISOPACKET, then write a shortened FM packet into the payload field of the ISOPACKET. On the receive end, the LANai reads the entire data structure into SRAM, then looks at the first two bytes to decided what type of packet it has. If it's a straight FM packet, it places it in the FM host receive queue as usual. But if it's an ISOPACKET, then it moves the isotach info from the header of the ISOPACKET into the appropriate fields in the net_sref in the payload field of the shortened FM packet, which itself is in the payload field of the ISOPACKET. The LANai code then writes the shortened FM packet payload of the ISOPACKET into the FM host receive queue. In this way, the FM receive queue gets a packet with the correct format (so it's happy), it contains the isotach information needed for subsequent isotach operations

(so the isotach host code is happy), but the message is transported over the wire in a form that conforms to the H/W SIU specs (so the H/W SIU is happy).

A.7 Some general comments about byte ordering

As mentioned above, the byte ordering on the LANai is network ordering (big endian—the least significant byte is at the higher address), while on the host it's little endian (host is a pentium architecture). There are many different conceivable ways in which this difference could become manifest when data is transfered from the LANai to the host and vice versa. As an example, a struct ISOCHRON_MARKER has the following fields:

```
struct ISOCHRON\_MARKER {
  USHORT tag;
  UCHAR isochron\_crc;
  UCHAR timestamp;
  USHORT source;
  UCHAR iso\_id;
};
```

Now, suppose that the values of the individual fields are

tag = 0xabcd; isochron_crc = 0xef;
timestamp = 0x01;
source = 0x23;
iso_id = 0x45;

and that the struct is stored in the LANai SRAM beginning at LANai memory address 0. Then we could expect the SRAM memory to look like this:

address 0: ab
address 1: cd
address 2: ef
address 3: 01
address 4: 23
address 5: 45
address 6: xx
address 7: xx

(here xx = don't care)

When this is DMAed to the host, we might expect the memory (because of the endian switch) to look like

address 3: ab
address 2: cd
address 1: ef
address 0: 01
address 7: 23
address 6: 45

address 5: xx
address 4: xx

so that if we arrange the host memory in order of ascending addresses:

address 0: 01
address 1: ef
address 2: cd
address 3: ab
address 4: xx
address 5: xx
address 6: 45
address 7: 23

Now, the "tag" field of the struct is a two byte field. If we have a pointer to address 0, and we cast it to a pointer to struct ISOCHRON_MARKER, and we ask for the value of the "tag" field, we might reasonably expect that it would return a value based on the contents of bytes 0 and 1, the first two bytes (according to its address scheme). Fortunately, it doesn't.

Bottom line: Byte ordering here works as follows.

If you have any single byte field, then just referencing that field will always give you the correct byte (on either the LANai or the host processor). For two byte fields, the byte order will be switched (so you have to use htons() or ntohs()). For four bytes fields, the same rule applies, only one needs to use htonl() or ntohl().

Now, how is byte order switching managed in our code? Well, not as cleanly as it might be, but with good reason. It would be nice to say everything on level "blah" is in whatever order, etc. But that would mean lots of unnecessary byte order switching in the critical path. So, it information is loaded into a packet at the host level, and this info is not needed by the LANai (i.e. the LANai does not need to interpret that particular field), then the byte ordering of the field isn't switched. If the LANai does need to interpret the field, then the switching is done.

For example, if one looks at the version 2 code for the routine FM_iso_send() (in fastmsgs.c), one will see first isopacket headers being switched (LANai needs to know they're isopackets so it can prepend the appropriate bytes for the H/W SIU), then FM packet headers that are NOT switched, then finally, the flags fields in the embedded net_sref (in the payload field of the FM packet in the isopacket) switched (because the LANai needs to interpret these to know if this represents the end of an isochron, etc.).

If you modify this code, be sure you understand how the byte ordering stuff is done!

A.7.1 How tokens are "turned around" in SIU mode

In the V1 prototype, the token turnaround mechanism in SIU mode works this way: When a token is received, the NIU code increments the variable recv_clock. In the main loop of the lcp.c code, a check is performed—if recv_clock ¿ send_clock, then call send_token(). The call to send_token() increments send_clock, and queues up the required return token.

A.7.2 The Path of a token in HH mode

This is a bit more involved than the SIU mode case. Once again, we are discussing V1.5 code, not V2 or V1.75. In the V1.5 code, when a token is received, it's flags are stored in the "prevsender" field of the FM packet.

[Aside: This is really not the best way to code this in terms of readability—the prevsender field happens to be the two byte field (in an FM packet) that begins two bytes from the left. (i.e. these would be the third and fourth bytes received off the wire). Since a token_t is a four byte struct with the flags field also of length two bytes and starting two bytes from the left, it's natural to just use the prevsender field. For readability, though, there really should be a cast to token_t and dereferencing of the token_t.flags field.]

When the fastmsgs.c consume() routine is called, it takes the token and passes the flags to the smm.c iso_tok_handler() routine. iso_tok_handler() checks whether the flags are correct, whether it's an early token, etc. If it turns out that we've received another isochron in the pulse corresponding to the token, then iso_eop_handler() is called, and passed a timestamp. iso_eop_handler() uses the timestamp to place an SREF_EOP in the appropriate bucket and put the bucket on the tobe_exec[] list. The routine also increments hh_recv_clock. This is key, because it causes the return token to be sent. Specifically, during the next call to iso_poll(), iso_poll() checks whether hh_recv_clock is greater than hh_send_clock. If it is (it should be if we received an in-sequence token), then iso_poll() calls enqueue_token(). iso_poll() then empties the token queue by repeated calls to FM_iso_send_token(), the routine that actually places the enqueued tokens on the wire.

recall the purposes of each of the routines mentioned:

enqueue_token() — ORs in current barrier and signal bits, places token on queue, updates lost token buffers (lt_buff[]).

FM_iso_send_token() — places token on LANai send queue (i.e. sends it).

iso_poll() — extracts messages (calls FM_extract()), processes srefs, calls enqueue_token() (if correct time), calls FM_iso_send_token() (sends the token), sets timers, and if necessary, calls lost_token_handler().

lost_token_handler() — requeues the appropriate tokens according to the lost token algorithm.

A.8 Some Miscellaneous Rambling

A.8.1 Resets on initialization

The sequence is fairly straightforward. On system initialization, after the FM code has mapped the network, the routine iso_init() is called, at which time node 0 sends a reset token to its TM. The TM, upon receiving this, should immediately send a wave of reset tokens to each TM and SIU neighbor.

A.8.2 Identifying Isochron Markers

Distinguishing (at the LANai) between Isotach messages and Isotach CHR markers is a little problematic because they both come with the same "packet type" field. We have to distinguish based on the size of the packet. If it turns out (for whatever reason) that an Isotach message arrives and is seven bytes long, then we could see some real interesting bugs here. (So consider this note a "heads up".)

A.8.3 Thoughts on Packet Issues

The notes below were the ones I wrote when deciding how to change the V1.5 packet formats to those that would work for V2 (or V1.75). I've included this because it reflects the thought process and may clarify why certain packet handling mechanisms work the way they do.

Another header issue (they just keep popping up): we are not just adding isotach headers here, but instead adding both a header to the front of a packet and a portion of a packet to the "middle" of a packet (because FM packets, as put on the wire, don't conform to the format in our specification). A problem then arises with packet boundaries not being word aligned. Specifically, FM is implemented so that packets end up aligned on word boundaries in the various queues. With our added stuff, this doesn't happen. So we need to consider this particular problem. (It affects how many bytes we read off the wire, in particular. It also affects where the checksum ends up in the LANai SRAM—since this location is kind of hard wired into the code, that's another consideration.)

Perhaps a way to get around this is by having a struct ISOPACKET. When a packet arrives, we need to specify how many bytes need to be read off the wire. If we are receiving PACKETSIZE bytes for non-isotach stuff and ISOPACKETSIZE bytes for isotach stuff, which do we use? And if we use the larger ISOPACKETSIZE, then we have to change some of the initialization constants (such as as HRECVFRAMES) which determine how many of a given type of struct can fit in a fixed size chunk of memory.

OK, maybe there is a way around this: We have some room to play with in a packet because a net_sref doesn't come close to filling all of the data field in a packet (by my calculations (below), we've got around 24 extra bytes to play with). We could have a struct iso_packet be the same size as an FM packet, but have it adjusted so that the FM packet fields are translated a few fields over. Then, upon receipt, if it's an isotach packet, we extract the packet one way, and if it's a non isotach packet, we extract it the way it is now (by extract I mean remove from the LANai receive queue and write the contents into the FM host receive queue). I.e we would be embedding an FM packet IN an iso_packet. The difference would be that the FM packet in an iso_packet would have a shorter data field in order to compensate for the extra bytes thrown on in front. This shouldn't present problems to the FM code (except possibly for the CRC stuff) because an isotach packet carrying an FM packet is really transporting a net_sref, which as mentioned above is shorter than a regular FM packet.

This definitely seems the way to go: I could change the code so that when FM_iso_send() writes directly into the LANai send queue, it's writing to a struct ISOPACKET (I could do this with a simple type cast of the sendqueue tail pointer (stailptr)). Then at the other end, I'll have the LANai code check the packet type byte and handle it according to whether it indicates an isotach packet or a straight fast messages packet.

A.8.4 More Random Stuff

1. Where should the byte order switching take place? On the LANai as an incoming packet arrives? Or should we just wait and do it in the host? I say do it in the host. This way there is less that the LANai needs to do, and we only need switch the stuff that the FM layer actually uses.

2. I've also decided (actually, the decision was thrust upon me—the alternative will probably be too slow) to modify the FM code to handle Isopackets on the host. It would have been nice to just do a few modifications on the LANai and leave the host FM code unchanged, but that would require TWO extra copies of each Isotach message, probably killing our performance.

3. Recall from the spec that an isochron marker is received when it arrives in the host receive buffer, and delivered when the SIU notifies the host that the pulse corresponding to the marker TS has arrived (i.e. when the host receives the corresponding EOP marker). Thus, on receipt of an EOP mark, the host is free to begin executing the srefs in the corresponding bucket in the order dictated by the sort vector (and just for informational purposes, the sort vector is simply a permutation—the srefs have arrived in the bucket in FIFO order, so if the sort vector reads 2,1,5,8... this means execute the second sref in the bucket, then the first, then the fifth, etc. Note also that the sort vector starts counting from 0, not from 1.) So a decision needs to be made here. When an EOP marker arrives, do we copy the sort vector to another buffer (thus freeing the frame in the host receive queue, and obviously requiring a perhaps unnecessary copy), or do we simply read off the indices from the sort vector and execute the srefs in that order (so that the frame in the host receive queue is not freed until after all of the srefs corresponding to the sort vector have been executed)? Right now (in the all software prototype), at the receipt of an EOP marker, the marker is placed in the appropriate bucket, the index of the bucket is "added" to the tobe_exec[] array, and then the buckets listed in tobe_exec[] are drained (and executed) when drain_buckets() is called during an iso_poll(). So as things are presently implemented, it would seem that the choice is made for us. We need to make the copy of the sort vector and then use this to tell us which srefs to drain from the bucket.

A.8.5 Token Handlers Differences Among Versions

Some of the handler routines in V1.5 were designed to accept simply a timestamp. The ones that come to mind are iso_chr_handler() and iso_eop_handler(). I wrote versions of both of these for V2. The former accepts a pointer to struct ISOCHRON_MARKER as input, while the latter accepts a pointer to struct EOP_MARKER as input. I also wrote a new routine iso_hh_eop_handler() which takes care of the situation where we are in HH mode, which is significantly different than when we are in SIU mode.

A.8.6 Sort Vectors

One of the modifications in changing to V2 involved handling sort vectors. Like the packet handling musings above, this section shows the thought process through which we arrived at the current implementation (for every bucket there is an associated sort vector queue). Note that because this reflects a thought process, one should read the whole section to understand the way the code currently works.

I implemented a struct sort_vector_t consisting of an integer field called count, and a field called sort_vector_entry[] that is an array of UCHARs. I then implemented an array of size NUM_BUCKETS of struct sort_vector_t called sort_vectors[]. sort_vector[i] contains the count information and sort vector most recently received (from the H/W SIU) for bucket i. The array is statically allocated, and is necessary since buckets are not drained immediately after receiving the EOP marker from the SIU (so we need a place to store the sort vectors—they can't just remain in the FM host receive queue).

Also, handling sort vectors from the SIU means not having to sort all of the entries in the buckets. Thus, I wrote a VERSION2_HOST version of drain_bucket() to do only the sorting necessary. Specifically, if the bucket has 32 or less srefs, then there is no need to sort. If 33 or more, then we need to sort the rest on the host and merge the two sorted lists. The routine decides which is the case by looking at the count field in the appropriate element of sort_vectors[].

A technical note: I asked Max Salinas about the lowest index that the SIU can send the host in the sort vector (i.e. is it zero or one). He informed me that SIU simulations show that the lowest index is zero, i.e. the first sref sent by the SIU to the host is numbered 0, not 1. That's not a big deal, but we need to be specific to get the code working.

Some more interesting stuff to consider: The memory for the buckets on the host is statically allocated, but the buckets are implemented as a queue as opposed to an array. The reason for this, I assume, is that is makes it unlikely that information in a particular bucket would be overwritten before the information was used (because it might be three or four cycles through the particular bucket before particular memory locations were overwritten). This implementation choice, however, makes for lots of unnecessary bookkeeping when utilizing the sort vectors supplied by the SIU. (In particular, with an array implementation, "finding" the srefs corresponding to a particular entry in the sort vector would be trivial. Without it, it's an exercise in pointer arithmetic and management.) Moreover, my implementation of the sort vectors stores them in static ARRAYS (as opposed to queues), so even if the bucket is not overwritten for a few cycles, the sort vectors will be, effectively eliminating the benefits of using queues for buckets. So it would probably be best to implement the buckets as arrays (for a number of reasons). This switch, however, would take significant effort, and for that reason, it will have to wait for a later time. Nonetheless, for an efficient system, it's a change that needs to be made.

Addendum to the above paragraph. After talking with Craig and Paul, it's clear that if the buckets had been implemented as arrays instead of queues, overwriting buckets before they were emptied could be a real problem. I had hoped to implement the sort_vectors[] array as arrays as opposed to queues, but if I do that, then sort_vectors[] are in danger of being overwritten (just as buckets[] would be). Now, implementing sort_vectors[] as a queue causes a problem not encountered with buckets[], and this problem stems from the different arrival times of the contents of each data structure. With buckets[], srefs arrive somewhat piecemeal. With sort_vectors[], however, the data arrives all at once (through a call to memcpy()).

Now, memcpy doesn't work (or at least not well), if the region one is copying to has to be "wrapped around". So to avoid this, I've implemented sort_arrays[] as a weird sort of queue that never wraps around. When data is to be copied into sort_vectors[j] (specifically into it's associated sort_vector_entry[] array), we do a quick check to ensure that there is a contiguous chunk of memory from the current pointer (into the sort_vectors[j].sort_vector_entry[] array) to the end of the array large enough to contain the data. If so, we memcpy() it in. If not, we reset the pointer to point to the start of the sort_vectors[j].sort_vector_entry array, and then memcpy() the data in. We get pretty much the same amount of guarantee of not being overwritten as with a traditional wrap around queue, but get to use memcpy() AND avoid the headache of wrap around.

One other comment: the routine drain_bucket() accepts as input both a bucket index (bucket) and a timestamp (ts). It never uses the timestamp, so why include it? Perhaps because we may need it some day if we include a call to a message handler in the routine. The above sentences are not accurate—drain_bucket() in V1.5 uses the ts input in the call to execute_hit_buf().

B Some Tips for Programming the LANai

Some comments on programming the LANai:

First off, programming the LANai is a bit of black magic. The interface seems as if it's relatively straightforward, but subtle differences can have a major impact on the

performance of code. Some things that are good to know:

1) For whatever reason, the LANai sometimes gets itself into strange states. If a program has been working OK and then suddenly starts to produce unexpected results, reboot. Soft reboots may not suffice—a power cycle is sometimes necessary.

2) When writing LANai code, you always have to write in an "infinite loop" style. That is, once the LANai code starts, it should never terminate (even if all it does is hum in a loop that does nothing, that's still fine). It can spin on various events, but under no circumstances should the execution path run out or terminate.

3) The easiest way to write LANai code is to modify code that already exists. The reason for this is that the way variables are declared can have a big impact on behavior. Many memory mapped variables, for example, need to be global (so that storage is set aside for them prior to run time). Do not try to memory map any variable that has it's storage allocated on the run-time stack. Bottom line: when modifying LANai code, be especially careful of how variables are declared (i.e. static, volatile, etc.).

4) It is especially important to understand the timing issues involved. Read the LANai documentation carefully! Many status flags, for example, are undefined for a few cycles after the event that triggers then occurs. (which is why there are a number of no-ops in the code—they all involve timing).

5) It is also important to understand the behavior of various registers (the behavior of which is somewhat poorly described in the LANai documentation). For example, when performing a receive DMA (from net onto LANai SRAM), one sets the value of the register RMP to point to the beginning of the SRAM memory buffer into which the data will be placed. The value of RMP absolutely MUST be a multiple of 4 (effectively, the two least significant bytes of this register are hard wired to zero). The register RML, on the other hand specifies the address of the last 4 byte WORD in the buffer (RML stands for receive memory limit). So, for example, if you set the value of RMP to 8020 (we'll assume the LANai is byte addressable, since it is), and you set RML to 8024, then what you are saying is that your buffer consists of the 8 bytes in addresses 8020-8027. If you only intend to receive 4 bytes, you can simply set RMP and RML to the same value. The registers SMP and SML work similarly, except they are for DMAing data from SRAM out to the net.

One particular issue to watch for. Suppose you have a 6 byte struct which you wish to DMA out to the net. Lets call it "mystruct". You might be tempted to do something like this:

```
SMP = (ULONG *)&mystruct;
SML = (ULONG *)&mystruct + sizeof(struct) - 4;
```

> This looks like it should send six bytes if sizeof(struct) is six. But as with RML, SML also has it's two least significant bits hardwired to zero. So, so that &mystruct turns out to be 8020. It looks like the two lines above are effectively saying

```
SMP = 8020;
SML = 8020 + 6 - 4 = 8022;
```

But in fact, because of the two least bit phenomenon, what you're getting is

```
SMP = 8020;
SML = 8020;
```

Thus, you're only sending 4 bytes instead of 6.

There are a couple of ways around this. One is to just send the last two bytes using the SH or SB registers.

Another way is to use the LANai registers that instruct the DMA engine to ignore the first x number of bytes in a word. The LANai documentation lists the commands for doing this sort or maneuver.

Finally, one should be aware of the four functions htonl(), ntohl(), htons(), and ntohs(). These are not available on the LANai, but they do can run on the host. They switch between network byte order and host byte order (or vice versa) and they operator on either long integers (assumed to be 4 bytes) or short integers (2 bytes) corresponding to the "l" or "s" letter in the function name. The "h" and "n" letters correspond to "host" and "network", so that ntohl() converts long integers from network byte order to host byte order. I believe one needs to # include <netinet/in.h> to use these, although the man pages say the correct header file is <arpa/inet.h>. I think the man pages are wrong, though, since there is no mention of the latter header file anywhere in the code, and we do use these functions. For a complete explanation of these functions and their use, see [7].

C Locating The Existing Isotach Code Base

The existing Isotach code base consists of both the code for the various components (and various versions), as well as the test generation tools being used to debug the SIU.

C.1 The System Code

The system code is located on our file server (grover) in Small Hall. Typically when a new member of the Isotach Group acquires an account on this machine (through Dale Newfield), they are given a copy of the "isofm" directory. This was (is supposed to be?) the root of the CVS tree for the code. Unfortunately, with the recent crash and subsequent replacement of grover, the CVS system seems to have remained in a state of limbo. In any case, to aquire the most recent copy of the V1.5 code, and specifically the isofm directory (there have been no changes to it since sometime in January 1999), one should look in Dale's personal isofm directory on grover (`/home/din5w/research/isotach/isofm/`). The last time I checked, the files were world readable. There are also other helpful isotach documents, such as copies of

Dales project writeup (`/home/din5w/research/isotach/masters.project.zip`—be sure to transfer it to the regular CS department machines before trying to unzip and view it).

The V1.75 and V2 code are in separate directories, the former in `/home/dcs8y/isofm/v2hh/`, the latter in `/home/dcs8y/isofm/HH_mode/` (I know they could have been given better names, but at the time this was set up, I was thinking that before I left these would be placed in special directories in the CVS repository).

One should be aware of a few things concerning this code. To begin with, each version of the code has its own makefile (in the appropriate directory). These makefiles are presently configured so that the host mode versions of the code are compiled. Compiling for SIU mode is accomplished by removing the `-DHH` markers in the makefile. Also, there is a large amount of overlap between the V1.75 and V2 code (for example the module lcp.c in V2 has versions of the routine receive_messages() written for V2, for V1.75, and for V1.5). My original intention was to simply use preprocessor directives to distinguish between V1.75 and V2 compiles. When it came time to start the debugging, I realized that this was the first situation during the isotach project in which two different versions of the isotach library and two different versions of the same executable needed to run concurrently. The easiest solution was to put each code base in a separate directory.

One final note: when a host is intended to emulate the performance of a TM, it needs to run the executable "tmhost" (again, compile and run this from Dale's directory, not mine). Hosts that are running application code should simply run the executable named after the application (to run dining philosophers, for example, the executable is "philo").

C.2 The Test Drivers

All of the test drivers are stored in their own subdirectory of `/home/dcs8y/isofm/test_drivers/`. Each of these subdirectories contains at least the following files:

- receiving node host source code in the file `receive_test_host.c`.
- receiving node LANai source code in the file `receive_test_lcp.c`.
- sending node host source code in the file `send_test_host.c`.
- sending node LANai source code in the file `send_test_lcp.c`.
- the executables `receive_test_host`, `receive_test_lcp`, `send_test_host`, and `send_test_lcp`.
- the header file `lcp_macros.h` which contains macros that check various interrupt status bits.
- the header file `lcp.h` which is "# included" in each of the above ".c" files and contains the definitions for data structures used in the V1.75 and V2 system code modules.
- the makefile `Makefile`.

To run any the tests, one need only start the executable `./receive_test_host` on the receiving machine and `./send_test_host` on the sending machine. Note that the "./" may be necessary if the current directory is not at the head of your PATH environment variable (there are, obviously, a lot of `send_test_host` executables in nearby directories).

Note also that the "linking" phase that connects the LANai executable with the corresponding host executable does not occur in the compilation stage. Instead, it is the "lanai_load_and_reset()" routine called by the host that loads the LANai and instructs it to begin execution. Be sure that the input file parameter passed to lanai_load_and_reset() is

correct. Because of difficulties with this, I hardwired the absolute file name into each host code source file (there are a lot of `receive_test_lcp` executables floating around in these directories and every once in a while the wrong one was loaded.

D Useful Implementation Constants

I thought it might be nice to have a record of the sizes of various data types used here (it's good to know these when things are being placed in the payloads of packets, etc).

Packet—-76 bytes, broken down as 12 header bytes and 64 payload bytes (i.e. the "data" array).

ISOPACKET—76 bytes. The whole point was to make it the same size as a packet.

sref_opcode_t — this is an enumerated type whose first value is 77770 which translates to 1 0010 1111 1100 1010 in binary. It thus requires 4 bytes.

sref_header— 20 bytes if in paranoid mode, 16 if not.

sref_body — a union type. either an sref_mem or an sref_msg. Since an sref_msg is the larger of the two, an sref_body always has this size. From below we see that it can be either 36, 40, 44, or 48 bytes, depending on which preprocessor flags have been set.

sref_msg — 44 bytes if in V1 or V1.5 and not in paranoid mode, 48 bytes if in v1 or v1.5 and in paranoid mode, 40 bytes if in V1.75 or V2 and paranoid, and 36 bytes if in V1.75 or V2 and in paranoid mode.

MAX_ISO_MESSAGE_SIZE — 38 if in V1.75 or V2 and not in paranoid mode, 34 if in V1.75 or V2 and in paranoid mode.

sref_size_t — it's an enum type, so size of an int = 4 bytes in our implementation.

sref_mem — 20 bytes. (five fields, each 4 bytes in length, (pointers take up 4 bytes)).

net_sref — 64 bytes if in V1 or V1.5, 56 bytes if in V1.75 or V2.

sref — 72 bytes if in V1 or V1.5, 64 bytes if in V1.75 or V2.

sref_header — 20 bytes if in paranoid mode, 16 bytes if not.

FM_SHORT_PACKET — 68 bytes.

SHORTPACKETSIZE — 58

Note that since SHORTPACKETSIZE is the size of the payload in a struct FM_SHORT_PACKET, a net_sref does fit into this payload.

E Further Sources of Information

Certainly the documents listed in the bibliography to this paper are some of the best sources of information concerning isotach systems and our prototype implementations.

For information and technical specifications regarding Myrinet networks, the Myricom website, `http://www.myri.com` is helpful. Especially useful are the documentation page (`http://www.myri.com/scs/documentation/`), the Myrinet Users Guide (`http://www.myri.com:80/scs/documentation/mug/`), and the LANai 4.X Documentation (`http://www.myri.com:80/scs/documentation/mug/development/LANai4.X.txt`). The latter is an invaluable reference for anyone who needs to write code that runs on the LANai.

References

[1] AT THE UNIVERSITY OF VIRGINIA, I. G. Isotach system web page. At URL http://www.cs.virginia.edu/~isotach/.

[2] BODEN, N. J., COHEN, D., FELDERMAN, R. E., KULAWIK, A. E., SEITZ, C. L., SEIZOVIC, J. N., AND SU, W.-K. Myrinet—a gigabit-per-second local area network. *IEEE Micro 15*, 1 (February 1995), 29–36. Available from http://www.myri.com/research/publications/Hot.ps.

[3] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (July 1978), 558–565.

[4] PAKIN, S., KARAMCHETI, V., AND CHIEN, A. A. Fast messages (fm): Efficient, portable communication for workstation clusters and massively-parallel processors. *IEEE Concurrency 5*, 2 (April-June 1997), 60–73. Available from http://www-csag.cs.uiuc.edu/papers/index.html#communication.

[5] REGEHR, J. An isotach implementation for myrinet. Tech. Rep. CS-97-12, University of Virginia, 1997. Available from http://www.cs.virginia.edu/~isotach/pubs.html.

[6] REYNOLDS, P. F., WILLIAMS, C., AND JR., R. R. W. Isotach networks. *IEEE Transactions on Parallel and Distributed Systems 8*, 4 (April 1997). Available from http://www.cs.virginia.edu/~isotach/pubs.html.

[7] STEVENS, W. R. *Unix Network Programming.* Prentice Hall, 1990.

[8] WILLIAMS, C. C. Design of the isotach prototype. Internal Working Paper.

[9] WILLIAMS, C. C. *Concurrency Control in Asynchronous Computations.* PhD thesis, University of Virginia, January 1993.