Testing the Functional Requirements of the Switch Interface Unit

Timothy L. Highley, Jr.

Department of Computer Science
University of Virginia
Charlottesville, VA 22903
Email: tjhighley@virginia.edu

May 3, 2000

Abstract

Isotach networks provide special guarantees about message ordering and delivery times in a parallel or distributed computing environment. Although isotach networks can be implemented entirely in software running on commercial off-the-shelf hardware, an isotach network that uses custom hardware to maintain logical time may be particularly efficient. A hardware-based implementation of an isotach network has been designed and built using custom components, one of which is a switch interface unit (SIU). An SIU handles isotach functions for a host. These functions include the maintenance of logical time, the timestamping of outgoing isotach messages, and the exchange of tokens with another custom hardware device, the token manager. We have designed and performed tests to analyze the switch interface units and increase confidence that the SIUs are functioning correctly. The analysis has indicated that the new units are working properly.

1 Introduction

Isotach networks provide special guarantees about message ordering and delivery times in a parallel or distributed computing environment. A hardware-based implementation of an isotach network has been designed and built at the University of Virginia. We performed correctness analysis on one of the hardware components of that network (the switch interface unit). The analysis has shown that the new units are working properly.

When messages are sent in a distributed or parallel computation environment, it can be valuable to know the order in which those messages will be received. With this knowledge, it is possible to guarantee that all messages in a multicast will be received isochronously. Isotach networks provide a mechanism to predict and control the logical times at which messages are received.

An isotach network provides this mechanism by using isotach logical time, which extends Lamport's logical time system by relating the logical time required for a message to travel to its destination with the logical distance the message travels. There are many possible implementations of an isotach network, ranging from hardware to a complete software system. An isotach network that uses hardware to maintain logical time may be particularly efficient.

The hardware-based isotach network at the University of Virginia has two types of hardware: token managers and switch interface units (SIUs). Correctness analysis for the token manager has already been performed [New98]; we now report the analysis for the SIUs.

The switch interface unit (SIU) is a hardware device that attempts to speed isotach networks by handling most isotach logic for a host. The SIU maintains logical time, maintains ordering guarantees, calculates logical distances, and determines delivery order among messages with the same logical timestamp.

Our goal was to establish, to the extent possible, that the SIU meets the requirements set forth in its specification [HWSpec]. In order to confirm that the hardware fulfills these requirements, we developed and performed a series of tests that directly address the requirements. These tests do not exhaustively test the prototype hardware, but they provide a high level of confidence that any SIU that passes all of the tests meets the requirements set forth in the hardware specification.

One of the major challenges we faced during the testing process was the interaction between untested software and untested hardware. When tests would fail, it was often difficult to identify whether the problem was caused by a problem in hardware or software. Another difficulty we faced was in developing tests that would convincingly demonstrate that the hardware was working correctly when we could not directly observe what happened in many parts of the hardware. This became particularly difficult when we needed to confirm the sequence of events during hardware processing.

This report describes our testing environment and methodology, reports the results of each of our tests, and examines the limitations of what the tests have demonstrated. During the testing process, we used two main types of tests:

- For almost every requirement, we developed input data to cover the cases that the requirement should address. We verified that the SIU provided the correct responses to the input data.
- For some of the more critical requirements, we developed stress tests to determine how the SIU would perform under heavy loads for long periods of time.

There are a few requirements, primarily those related to performance, which were not tested. Most of those will be addressed by future research. Two SIUs have already passed our tests, and four others are in the process of being tested.

2 Background

We assume the reader has some familiarity with isotach networks. A good overview of isotach networks can be found in [Bar99]. For more detailed information, see [HWSpec] and [RWW97]. For readers who are familiar with isotach networks but may need a reminder, a brief review is included here.

2.1 Isotach Networks

Isotach networks extend Lamport's logical time [Lam78] and provide a total ordering of all messages in the network. To implement a logical time system the networks use tokens, special messages that indicate that the sender has advanced his local logical clock.

Each message that is sent has an associated n-tuple, on which message ordering is based. In our case the n-tuple is a 3-tuple consisting of (pulse, pid, rank), where pulse corresponds to the number of tokens exchanged before the message is delivered, pid identifies the process that issued the message, and rank indicates an ordering of the messages that a single process sends.

An isotach network gives a sender the capability to control the logical times at which messages are received, thus enabling a sender to ensure that a group of messages are received isochronously. Isotach networks also enable the enforcement of the sequential consistency model or, if such a strong consistency model is not needed, causal consistency model.

Our current implementation of an isotach network over Myrinet consists of four major types of hardware components: hosts, network switches, token managers (TMs) and switch interface units (SIUs). TMs and SIUs are *isotach components*, custom devices built to implement isotach functionality. A TM is connected to each switch. Each SIU is connected to a host and to a switch. A TM's *neighbors* are defined as the SIUs attached to the same switch as the TM and also TMs that are connected to adjacent switches. An SIU's only *neighbor* is the TM that is connected to the same switch as the SIU.

Logical time progresses as tokens are exchanged between neighbors. An isotach component sends tokens only in waves, meaning that whenever it sends a token to any neighbor, it sends a token to all of its neighbors. An isotach component increments its logical clock when it sends a token wave; it sends another token wave only after it has received a response token from each one of its neighbors. In that way, the logical clocks of any two neighbors never differ by more than one time pulse.

When a host sends a message, the message passes through the host's SIU, where it is assigned a timestamp. The timestamp corresponds to the logical time pulse at which the message will be delivered. The SIU determines the message's logical delivery time by relating the logical time required for the message to travel to its destination to the logical distance the message must travel. Logical distance is a metric for measuring the routing distance; in our case it is the number of switches through which the message must pass. The isotach invariant is the relationship between logical time and logical distance.

When the receiver receives the message, the message has a 3-tuple (pulse, pid, rank) associated with it, which orders it with respect to all other messages. Because of the isotach invariant and the rules for exchanging tokens, a host is able to determine when it will not receive any more messages with a given timestamp. When it is determined that no such messages will be received, the receiver can process the messages for that time pulse with the guarantee that it is interpreting all of the messages in exactly the same order that any other host would interpret them. With this guarantee, a message can be sent isochronously to multiple hosts.

An isotach system does not address real time guarantees, but provides strong guarantees about message ordering. These guarantees are made possible by the isotach invariant.

3 Hardware Setup

The University of Virginia has developed a prototype implementation of an isotach network including custom hardware. This section describes the components in the isotach network and the layout of the hardware.

3.1 Myrinet Networks

The prototype system is based on a Myrinet network. Myrinet [BCF95] is a local area network with an extremely high data rate developed from technology used for massively parallel processors. A Myrinet link is a full-duplex pair of 1.28 Gbit/s point-to-point channels. The Myrinet physical layer provides a very low error rate combined with high bandwidth and low latency. Flow control is provided on every link. The switches use cut-through routing. Myrinet networks can be of an arbitrary topology, and performance is very scalable because they do not use a common bus like many data-link layer protocols.

Some interesting features of the Myrinet network interface are its general-purpose microprocessor, called a LANai, and up to 1 MB of high-performance SRAM. The network interface is distributed with source code and a C programming language compiler, allowing any customer to tailor the network interface to its needs. This combination of interface, LANai, SRAM, source code, and compiler has recently facilitated a push in the research of high performance messaging layers.

The Myrinet network interface makes use of send and receive DMA engines to move data to and from the network. The host is given access to the SRAM through programmed I/O, and the LANai can move data from the SRAM to the host memory and back with a third DMA engine. The speed at which the LANai runs makes the use of this third DMA engine critical to obtaining good performance.

For the tests we performed, we used a Myricom M2M-DUAL-SW8 Dual 8-port Myrinet SAN Switch, a LANai card in each of three hosts and 6 Myrinet cables to connect the network.

3.2 Switch Interface Unit

The switch interface unit (SIU) is custom hardware designed for the isotach prototype system. As its name indicates, it provides an interface between a host and a Myrinet switch. An important part of the SIU is Myricom's FI chip. The FI chip serves as an interface between the SIU and the Myrinet network. One FI communicates with the LANai on the host while the other FI communicates with the switch.

The SIU maintains logical time for the host, notifies the host of significant pulses, maintains ordering guarantees, ensures atomicity of isochrons, calculates logical distance and logical delivery times, determines delivery order among messages with the same logical timestamp, and handles signals and barriers.

3.2.1 Switch Interface Unit Configuration

There are a number of configuration options on the SIU that must be set to the appropriate values before the SIU can be used correctly. These options include the offset to the token manager, the epoch length for the system, the choice of SIU mode or host mode, and the token timeout interval.

The configuration options for the hardware SIU are encoded via toggle switches. One of the toggle switches is used to select whether the SIU will operate in SIU mode or host mode. Another toggle switch simply toggles a light on and off. That same light changes status at every reset. The switch can be used to make a set of SIUs have their lights either all on or all off. In that way, it is possible to easily verify that when a reset signal is sent to the system, each component in the system responds to the reset.

Dip-switch value	Epoch Length
0	2
1	4
2	8
3	16
4	32
5	Undefined
6	Undefined
7	Undefined

The other switches are collected together and numbered 1 through 12. These switches are known as dip-switches. The label on the dip-switches indicates that one direction is “on”. The “on” direction is a logical zero and the “off” direction is a logical one. Dip-switches 1 through 6 encode the six-bit two’s complement routing offset to the token manager, with dip-switch 1 being the most significant bit. Dip-switches 7 through 9 encode a value to represent the epoch length, with dip-switch 7 being the most significant bit. The table presents the values that should be encoded with dip-switches 7 through 9 and the epoch lengths that they represent. Dip-switches 10-12 encode a value to represent the token timeout interval, with dip-switch 10 being the most significant bit. The token timeout interval will be ($2^{(\text{Dip-switch value})} * 6.4$ microseconds).

3.3 Token Manager

The report of the testing of the hardware token manager can be found in [New98]. That report covers the testing of one token manager. Since that time, a second token manager has arrived and is in working order.

A replaceable configuration PROM in the token manager tells the token manager where its neighbors are and sets the epoch length.

3.4 Configurations

Most of the tests were performed in one of the following hardware configurations.

3.4.1 Configuration A1

Hardware configuration A1 consists of a hardware token manager on port 7 and two hosts, each with hardware SIUs, connected to ports 4 and 6 of the switch. We will refer to the components on port 4 as host A and SIU A, and those on port 6 as host B and SIU

B. The PROM for the hardware TM should be programmed for one SIU on port 6. In this configuration, the host on port 4 may be used for some purposes, but it is not involved in token exchange. Unless stated otherwise, the SIUs operate in SIU mode.

3.4.2 Configuration A2

Hardware configuration A2 is the same as configuration A1 with the exception that the TM is programmed for two SIUs, one on port 4 and one on port 6.

3.4.3 Configuration B

Hardware configuration B is similar to configuration A1 except a third host takes the place of the token manager on port 7. The third host, which replaces the token manager, often provides part of the functionality of a token manager. For this reason, we will refer to this host as the TM-host. The TM-host uses the sr program (section 4.1.1) to provide token manager functionality.

4 Testing the SIU

In order to ensure the correct operation of the hardware SIU, we have attempted to verify that the hardware satisfies the functionality requirements that are set forth in the SIU's hardware specification [HWSpec]; some requirements deal with performance goals and we have left those for future work. This section describes the individual requirements and the tests that served to check whether a given requirement is being satisfied. Also, we assess the degree of confidence that each test provides.

Many of the tests we perform deal with the logical clocks on the SIUs. It is useful to have some concise language to describe common occurrences in the logical time system. An SIU advances its logical clock when it sends a token. Under normal circumstances, an SIU will send the next in-sequence token as soon as possible after it receives a token from the network. If all of the components in the isotach network are sending tokens as quickly as possible, then the sendclocks of the components are rapidly increasing and we say that *logical time is progressing*. During the testing process we often needed to have precise control over the isotach network. For instance, we often needed to know the exact value of the sendclock of an SIU at a particular time. Frequently, we would have one of the components in the network receive a token, but then have it refrain from immediately sending out the next in-sequence token. At that time, we say the component is *holding the token*. If there is a component in the system that is holding a token in this way, and thus preventing other components from advancing their sendclocks, we say that *logical time is paused* or *stopped*.

If a component is holding a token but then sends a token to its neighbor, it would then receive the next in-sequence token from its neighbor. If the component then holds that token, we say that it has *exchanged a token* with its neighbor. In the source code, this process is also frequently called *bouncing a token*. When logical time is progressing, all components in the network are constantly exchanging tokens with their neighbors.

4.1 Testing Tools

Several small software tools were written as part of the testing process. The tools were intentionally kept small to limit the amount of code that had to be verified, making it easier to isolate bugs to either the hardware or the software. Descriptions of some of the tools follow.

4.1.1 Testing Tool: sr

Many of the tests make use of a tool called sr, which stands for “send and receive”. The tool’s first purpose was simply to send and receive individual packets, but a number of modifications have been made to more easily accommodate common isotach related tasks.

The sr tool can currently be found in the /home2/isotach/test/send_receive directory on the file server in the isotach laboratory (pepe.cs.virginia.edu). Appendix B contains instructions for using the tool.

4.1.2 Testing Tool: req21

The req21 tool is very similar to the sr tool. It accepts many of the same commands, but it is not an interactive program. Instead, it is designed to read its input from an input script. The sr tool provides slow, careful checking by always having a human as part of the process. The stress tests that are described below provide automated, large volume testing, but the test data is not carefully selected, as it is when sr is used. The req21 tool falls in between the two extremes. Because it is trivial to run the scripts once they are written, req21 facilitates multiple executions of larger test cases. However, the cases are carefully designed for the control that sr provides. The program was originally written to test requirement 21 (hence the name), but scripts were written for other requirements as well.

4.1.3 Testing Tool: barrier_stress

The barrier_stress tool was developed to test the barrier mechanisms of the SIUs. It runs under hardware configuration A2. We designed the program to be executed simultaneously on two nodes, which each send messages to the other node. The command line parameter when the program is executed is the offset to the other node (i.e. host A runs 'barrier_stress 2' and host B runs 'barrier_stress –2'.)

The barrier_stress program begins by sending out a barrier/signal marker with a barrier bit set. After sending the initial barrier/signal marker, the program enters a main event loop where it monitors incoming traffic. It will send out another barrier/signal marker whenever it receives an EOP marker that indicates the completion of the previous barrier. In between the barrier/signal markers the program will intermittently, at random intervals of time, send an isotach message to the other host. When one of the hosts sends an isotach message in this context, the message will contain as part of its payload the number of barrier/signal markers that the host had sent before sending the message. When the other host receives the message, it verifies that barriers have not arrived "early" by looking at the payload. The sending host had indicated in the payload how many barrier/signal markers it sent before it sent the message; the receiving host verifies that it received the same number of EOP markers before the message was received. The program continually checks for completed barriers and verifies that messages arrive in the correct order.

4.1.4 Testing Tool: 33_35

A stress test was written to verify that requirements 33 through 35 are being satisfied. It can be found at /home2/isotach/test/33_35/stress.c. This program runs under hardware configuration A2. During the test, each host sends isochrons to the other host at random intervals of time. The number of messages within each isochron is selected randomly from the interval 1-32; each message in the isochron is the same length. The source field for each outgoing message is randomly selected so that the receiving SIU's sorting mechanism is tested. By changing the source port field, the sending host is "lying" about who is sending the message, but that does not affect the test. The hosts receive the messages and eventually receive an EOP marker. The count field in the EOP marker should always match up with the number of messages and isochron markers the host received for that timestamp. If they do not match, the program reports a violation.

4.2 Individual Requirements

The requirements here are taken from the hardware specification for the SIU [HWSpec]. Many of the tests described are based on the preliminary ideas generated by Doug Szajda [Sza99]. A few of the tests come directly from that paper.

Following each number is a code that is of the form A/9/99. Respectively, the three symbols in the code represent relative importance, difficulty of testing, and degree of confidence for the given requirement. All three of the ratings for a given requirement are subjective assessments determined by the author.

Relative importance is graded from A+ to F, and is based primarily on how functional the system would be if the requirement were not to be satisfiable. Requirements related to host mode are automatically downgraded. Requirements related to performance and those related barrier or signal bits are also downgraded, since the system may still be of some value without meeting those requirements.

Difficulty of testing, rated from 0 (easy) to 9 (hard), is based on a combination of two factors. The first factor is the difficulty of determining how to adequately test the requirement. The second is the effort involved in actually performing the test.

The degree of confidence is given on a scale from 0 to 100. The degree of confidence is limited by the thoroughness of the test for that requirement. During testing, the score for each requirement began at zero and rose until verified. A score of less than ten indicated, “I have no idea if this requirement is met,” or even, “I know that this requirement is not being met.” A score near fifty indicated, “This requirement appears to be met.” A score of ninety or greater indicated, “I am confident that this requirement is met.” Our general approach was to treat the hardware as a black box, and in almost all cases a formal proof is not feasible when treating the hardware in that way. For this reason, none of the requirements have been rated 100 for degree of confidence.

Unless stated otherwise, the hosts in the following tests utilize the sr tool.

1.A+/1/99

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 mater as long as all SIUs and TMs use the same starting sequence number. (note that start actually equals 00)

We used a TM-host (section 3.4.2) to print out the packets that the SIU sent to it. We confirmed that when the SIU was turned on, the first packet it sent was a token with sequence number start.

We were able to see the output by using sr (section 4.1.1) on the TM-host (configuration B) with output turned on. We put the SIU in SIU mode, checked the dip-switches to verify that the SIU was programmed to send tokens to the TM-host, and then turned on the SIU. The first token that the TM-host received had a sequence number of 00, which is the value of start.

This was verified for both SIUs.

2.A+/3/99

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

We sent tokens to the SIU and monitored the t okens that the SIU was sending out. We verified that the sequence numbers of the tokens that SIU sent increased each time a token was sent to the SIU.

Sequence numbers are described in two bits. The initial sequence number in binary is 00, and the order follows a sequence of 00, 01, 10, 11, 00, 01, etc. A token with sequence number X may be described as an X-sequenced token. Using hardware configuration B, we used the s option of sr to send tokens one at a time from the TM-host to the SIU. Each successive token had a sequence number one greater (mod 4) than the previous token. After sending each token, we checked to see what the SIU was sending to the TM-host. Under normal circumstances, if the SIU is not receiving tokens from the token manager, the SIU should send reminder tokens such that if it is waiting for an X-sequenced token, it alternately sends tokens with sequence numbers X and X-1 mod 4 (requirement 9 states this). Thus, when the TM-host sent a 00-sequenced token, the sequence numbers of the tokens that the SIU was sending shifted from 11 and 00 to 00 and 01. Similarly, when the TM-host sent a 01-sequenced token, the sequence numbers of the tokens that the SIU was sending shifted from 00 and 01 to 01 and 10. Similar shifts were observed when a 10-sequenced token and then an 11-sequenced token were subsequently sent by the TM-host to the SIU. This process demonstrated that the SIU correctly responded to each of the four possible sequence numbers.

This was verified for both SIUs.

3.D/5/80

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 deals with a performance goal and was not directly tested. However, there is an easy test that was performed to determine if tokens were being exchanged at an extremely slow rate. We looked at the token traffic light on the TM and verified that it was yellow.

If the light for the hardware token manager's logical clock turns yellow, that indicates that the token manager is sending tokens. If the token manager is sending tokens, then that implies that it is receiving tokens as well. The yellow light also implies that the tokens are arriving quickly, since the light is in reality a light that is rapidly alternating between red and green. It is beyond the scope of this project to determine if they are arriving so quickly that the SIU is sending "as many as possible", but if the light is yellow it indicates that logical time is not progressing so unreasonably slow that the eye can track it. We used hardware configuration A1 and confirmed that the light turns yellow. Using the digital analyzer, we looked at the inter-arrival times between four pairs of tokens. Tokens arrived at an average interval of approximately 4 microseconds. This requirement should be investigated in greater depth during a performance analysis.

4.A+/6/97

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

We were able to predict what the sendclock value should have been at a given point in time by carefully controlling how many tokens were exchanged. Having predicted the sendclock value, we sent an isochron through the SIU and verified that the timestamp of the isochron corresponded to the predicted sendclock value. After exchanging one more token, we sent another isochron and checked its timestamp. We verified that the difference between the two timestamps was one, corresponding to the token that was exchanged between them.

We were able to control the number of tokens that were exchanged by running the sr tool on the TM-host (hardware configuration B) and then using the b## option. During this test, we had to assume that the timestamp mechanism (requirements 24 through 29) worked properly. During the tests for those requirements, we assumed that this requirement was satisfied. There is some circularity in this, but because the results of the tests were what we expected, we can still be quite confident that the SIU is working correctly.

This was verified for both SIUs and also for SIU #5.

5.C/6/97

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

The approach to this requirement was the same as that for requirement 4. We were able to predict what the sendclock value should have been by carefully controlling the exchange of tokens. We then sent one message, exchanged one token, sent another message, and verified the timestamps of the two messages.

The method for controlling token exchange in host mode is different from the method used for SIU mode. Rather than using a TM-host to control the token exchange rate as in the test for requirement 4, we used hardware configuration A1 and used a host to control the exchange rate. This method only works in host mode because in SIU mode, hosts are not involved in the token exchange. Host B sent a message through SIU B to host A. At host A, we checked the timestamp that SIU B had assigned to the message. We then sent a token with sequence number START from host B through SIU B to the hardware token manager (i.e. the SIU sent a non-duplicate token). We sent a second message through SIU B and checked the timestamp that was given to the message. The second message's timestamp was one greater than that of the first message.

This was verified for both SIUs.

6.A/2/99

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

We directly checked the source port field for a few tokens. Any error in loading the source port field will be immediately noticeable because the hardware token manager depends on the source port field to determine when it should send out the next token wave. Thus, if this requirement fails to be met, then the isotach system will cease to function.

Using hardware configuration B, we set the TM-host to print out the tokens it received. We verified that the source field was correctly loaded: the source port field was 3 for an SIU on port 4, and the source port field was 1 for an SIU on port 6. It would have been tedious to verify that every other individual value works as well. Since a hardware token manager checks the source port field of each token to determine when it should send out the next token wave, an SIU that began sending tokens with an incorrect source port value would cause logical time to stop progressing. Therefore, we can conclude that whenever logical time progresses then this requirement is being met.

This was verified for source port values 1 and 3 for both SIUs.

7.B/9/85

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.

The primary difficulty in testing this requirement was the problem of generating a CRC error, something that should very rarely or never occur. We managed to generate a CRC error by exploiting the fact that an SIU in host mode corrupts the CRC of an outgoing token if the token has an early sequence number. We then used a unique hardware configuration to direct the corrupted token to an SIU and verify that the token was dropped.

We connected two SIUs to the switch, one to port 7 and the other to port 4, and then set both to host mode. We connected a hardware TM to port 5. The PROM for the TM specified that the system had one SIU on port 6, but since the TM was connected to the "wrong" port of the switch, the TM was actually communicating with port 4. In host mode, the SIU on port 4 will corrupt an outgoing token if that token is an early token (see requirement 8). The host on port 4 sent a 00-sequenced token to port 7 and we verified that the host at port 7 received the token. (The hexadecimal representation of the complete token was 60830601 0300.) We then reset all the hardware. We used the 'b3' option of sr to have the port 4 SIU send three tokens to the TM. At that point, the next token sent by the SIU should have had sequence number 11, and token with sequence number 00 would have been considered early. We had port 4 again send a 00-sequenced token to port 7, but this time the SIU on port 4 corrupted the CRC of the token. The SIU on port 7 dropped the token and it was not received at the host. The sequence number was correct as far as port 7 was concerned, so it is possible to verify that the token was dropped because the CRC was corrupted and not because it was an early token. This only verified the requirement for host mode, but this verification helps build confidence that a similar mechanism for SIU mode is working.

This was verified for SIUs 1, 2, and 3.

8.C/4/92

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

We sent tokens though an SIU in host mode to another host and verified at the receiving host that the CRC was corrupted when the conditions were met.

Using hardware configuration B, we used the s option of sr to send tokens one at a time from host B to the TM-host. A token is an early token if the SIU is expecting a token with sequence number X but receives a token with sequence number X+1 mod 4. All tokens had valid CRC bytes when received at the TM-host, unless the token was an early token. For each early token, the least significant bit of the CRC byte was corrupted. We sent an early token and a valid token with each of the four possible sequence numbers.

This was verified for both SIUs.

9.C/3/99

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.

We did not evaluate the timing requirement listed here, but by using hardware configuration B, we verified that the TM-host continually received repeat tokens from the SIU when tokens were not being exchanged.

This was verified for both SIUs.

10.A/7/94

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.

We wanted to verify both the "if" and the "only if" portions of this requirement. First, in order to detect if an EOP marker is ever generated without proper cause, an assertion can be placed in the isotach software. The assertion would signal an error if an EOP marker ever came back with no barrier or signal bits set and with a zero in the count field.

Second, we wanted to check that an EOP marker is delivered for each of the following cases: (1) isochron marker was delivered for that pulse, (2) isochron message was delivered for that pulse, (3) completing a barrier, (4) reset signal, and (5) normal signal. We developed tests for each of these conditions.

(1) isochron marker was delivered for that pulse

We sent a message from one host to another host. We checked the timestamp on the isochron marker that the first host received and verified that it also received an EOP marker for that same timestamp.

This test used hardware configuration A2. Using the s option of sr, host A sent an isotach message (26000000 00820600 00000000 5a5b5c5d) through SIU A to host B in order to generate an isochron marker. We checked the timestamp on the isochron marker that the host A received; it also received an EOP marker for that timestamp.

This was verified for both SIUs.

(2) isochron message was delivered for that pulse

We sent an isochron and verified that the receiving host received an EOP marker corresponding to the timestamp of the isochron it received.

As in the test for 10-(1), we used hardware configuration A2 and sent an isotach message from host A to host B. We checked the timestamp on the message that host B received and verified that it also received an EOP marker for that timestamp. We watched for other EOP markers and verified that since no other significant pulses were present (i.e. no barriers, signals, or isochron markers), no EOP marker was received without a corresponding isochron.

This was verified for both SIUs.

(3) completing a barrier

We developed a stress test to verify that barriers worked correctly. The program is called barrier_stress (see section 4.1.3) and can currently be found in the isotach cluster at /home2/isotach/test/barrier_stress/ on pepe.cs.virginia.edu. The program will not run correctly if an EOP marker is not sent to the host upon completion of a barrier.

All six SIUs have successfully run barrier_test.

To test this part of the requirement manually, we had a host send its SIU a barrier/signal marker and then counted the number of barrier tokens that the TM-host sent before the SIU sent an EOP marker indicating the completion of the barrier.

The manual part of the test was run under hardware configuration B. First, we ensured that the TM-host sent the initial barriers at start-up. We then used the s option of sr to send a barrier/signal marker from the host to the SIU with count = X. Using the TM-host, we sent barrier tokens to the SIU one at a time so we could count them (using the w and b1 options of sr). In each trial, the host received an EOP marker after the TM-host had sent X-1 barrier tokens to the SIU. This was repeated for several of the possible values of X (2-37). We checked the minimum value but not the maximum value. We performed this test on both SIUs.

A possible problem may arise if a barrier/signal marker arrives at the SIU when the SIU has not yet received a barrier token from the network, either at startup or immediately after completing the previous barrier. Under normal circumstances, the SIU will copy the barrier count in the barrier/signal marker and will then immediately decrement it by one because the SIU had already received a barrier token. However, if the SIU had not yet received the barrier token from the network then it should not immediately decrement the barrier count. We believe the SIU behaves properly but did not explicitly test this scenario.

(4) reset signal

We sent an SIU a token with its reset signal set and verified that the SIU's host received an EOP marker before the SIU reset. This test used hardware configuration B. We sent the token with the reset signal from the TM-host (using the s option of sr). As expected, the host attached to the SIU received an EOP marker and then the SIU reset.

We also tested to see whether the SIU would send the EOP marker if it was the one that initiated the reset. We used hardware configuration A1 and sent a barrier/signal marker with the reset signal set from host B to SIU B. The host received an EOP marker as expected.

This was verified for both SIUs.

(5) normal signal

We used hardware configuration B. From the TM-host, we sent SIU B a token with a signal bit set. We then used sr's b## option on the TM-host to send enough tokens to reach the end of the epoch. The EOP marker was sent from the SIU to the host at the end of the epoch, and it always arrived at the proper time.

When working with signals on isotach hardware, the user must ensure that the SIUs and the token managers are using the same epoch length. A token manager's epoch length is programmed into the configuration PROM. Note that the value which is actually encoded into the PROM is one less than the epoch length. For example, 32 is represented in hexadecimal as 0x20, but 0x1F is actually encoded for an epoch length of 32. That is because 0x1F is the last timestamp of the first epoch, which begins with timestamp 0x00. The SIU's epoch length is encoded via dip-switches.

An EOP marker should come back with a signal bit set when the token represents the end of epoch. Due to the SIUs' encoding scheme, the epoch length will be a power of two. If a token was received with a signal bit set, then an EOP marker should come back with the signal bit set at the end of the epoch. If the epoch length is Y, then the end of the epoch comes after token number X is received by the SIU, where X=Y-1 mod Y. Since in general token number n will generate the EOP marker for timestamp n-1, the EOP marker for the pulse at the end of the epoch will have a timestamp of Y-2 mod Y.

This was verified for both SIUs.

11.C/3/98

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

We put an SIU in host mode and ran a program on the SIU's host to automatically exchange tokens with the token manager. The program would not have been able to exchange tokens if the SIU did not send the host an EOP marker for each token it received.

This test used hardware configuration A1, with SIU B in host mode. We used option b of sr (section 4.1.1) to exchange tokens. We could see that logical time was progressing, which confirmed that the SIU was passing normal tokens to the host in EOP markers. The test for requirement 18 confirms that the SIU also sends EOP markers for repeat tokens.

This was verified for SIUs 1, 2, 3, and 6.

12.C/3/98

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

We sent early tokens through an SIU, some from the host side and some from the network side. Early tokens received from the host side were corrupted by the SIU and early tokens received from the network side were dropped by the SIU.

We attached one host to the host side of the SIU and another host to the network side of the SIU. We sent tokens one at a time from the host side to the network side and kept track of the current sequence number. (We used the 'b' option of sr on the other host in order to make it easier to exchange tokens.) Recall that an early token is a token with sequence number X+1 mod 4 when the SIU was expecting a token with sequence number X. Whenever the SIU received an early token from the host side, it corrupted the token and passed it along. In a similar test, we sent tokens from the network side to the host side and kept track of the current sequence number. Whenever the SIU received an early token from the network side, it dropped the token.

This was verified for both SIUs.

13.A/5/92

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. (Restating the requirement: If pulse i is significant, then the SIU sends an EOP marker for pulse i only after receiving token i+1.)

We sent a message to the host of the SIU we were testing. We then sent tokens to the SIU one at a time and noted that the EOP marker for pulse i was delivered when token i+1 was sent to the SIU.

This test used hardware configuration B. We had the TM-host send isotach messages to the SIU. After sending a few isotach messages, we sent tokens to the SIU one at a time. We used the s option of sr to send the messages and b## to send the tokens. When token i+1 was sent to the SIU, it generated an EOP marker for timestamp i if and only if pulse i was significant (i.e. a message was sent with timestamp i). We repeated this enough times (around twelve) to convince ourselves that requirement 13 was satisfied.

This was verified for both SIUs.

14.A/5/97

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

This requirement was verified using the same test as requirement 10-(1). The EOP markers that came back always arrived after the isochron markers of the same timestamp.

This was verified for both SIUs.

15.A/9/89

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.

The particular error that this requirement is intended to prevent is not likely to occur. Because it is so rare we could not directly observe the error and thus did not test this requirement. However, we can put an assertion in isotach code that will alert the user if this requirement is ever violated.

Page 11 of the hardware specification of the SIU [HWSpec] dictates that the maximum timestamp of any message received by an SIU in pulse i is (i+2d+max_send_delta+2). The timestamp range is 256. In order to continue this discussion with a concrete number, timestamp 0x162 will be chosen arbitrarily (where "0x" indicates hexadecimal notation). The earliest time when an SIU could receive a message with timestamp 0x162 is (0x162-2d-max send delta-2) which is equal to 293, or 0x125 in hexadecimal. If the SIU is to send an EOP marker for pulse 0x062, then this requirement states that it must send the marker before sending the host any isotach traffic from pulse 0x162. However, the traffic for 0x162 cannot arrive at the SIU before logical time 0x125, so if the SIU would ever violate this requirement, then it would have to perform the following steps. First, it must receive the token indicating that it can send the pulse 0x062 EOP marker to the host. Then it must exchange at least 0x125-0x062=0xC3=195 tokens with the TM. Only after that is it possible for the SIU to receive a message for timestamp 0x162. In order to violate the requirement, the SIU would have to send that message with timestamp 0x162 before sending the EOP marker for timestamp 0x062. It is unlikely that the SIU would ever wait 195 token intervals before delivering an EOP marker. Due to the unlikelihood of this problem arising, this requirement was not tested directly. It does appear from the design that the message with timestamp 0x162 would indeed be held until after the delivery of the EOP marker for timestamp 0x062 (in the event that such a check would be needed), but this is hard to verify directly.

In isotach code, an assertion can be made that the count field in an EOP marker always matches the number of isotach messages and isochron markers that were received for that pulse. If any isotach traffic for the next congruent pulse modulo the timestamp range were to be sent to the host before the EOP marker for the first pulse, then the assertion would alert the user to the violation.

16.B/2/98

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

This requirement is implicitly verified whenever anything relating to EOP markers works properly. If the SIU would ever fail to meet this requirement, the failure would be obvious.

We verified this requirement by using the tests for requirement 10. All of the EOP markers were received with packet type 0x0601 and with a good CRC byte. If the SIU did not copy the packet type correctly, the host would never know when to process the isotach messages, so any isotach program would halt.

This was verified for both SIUs.

17.A/3/98

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

To verify this requirement, we used the same test that was used to verify requirement 13. We looked at the timestamp field of the EOP markers. We were able to count the number of tokens that were sent to the SIU so it was possible to know what the timestamp field should have been. We verified that the field did indeed hold that value.

This was verified for both SIUs.

18.C/2/99

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.

We sent tokens to an SIU; some were repeat tokens and some were normal tokens. We verified that the repeat bit was set for the repeat tokens but not for the normal tokens.

This test used hardware configuration B. We sent tokens one at a time in sequence (using s option of sr) from the TM-host to an SIU and checked that the repeat bit was not set in the EOP markers that the host received. After sending a few tokens in sequence, we interspersed repeat tokens with the normal tokens; we checked that the repeat bit was set in the EOP markers corresponding to those tokens, and only in those EOP markers. We sent a repeat token at each of the four sequence numbers.

This was verified for all four sequence numbers for both SIUs.

We also checked to see whether a token following an early token would be incorrectly interpreted as a repeat token. If the sequence numbers of the tokens follow this order, then the last token should not have its repeat bit set: 00, 01, 11 (early), 10, 11. We verified this scenario on SIUs #2 and #5.

19.A/6/97

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.

One of the test programs, 33_35, tests this requirement by tabulating, for each timestamp, the number of isotach messages and isochron markers that it receives. Whenever the program receives an EOP marker, it compares the count field of the EOP marker to the actual number of isochron markers and messages that it received. The two values should always be equal and it reports any violations.

Both SIUs ran the 33_35 program without generating any violation reports.

In order to test the case when the count field is greater than bucket_size, we used hardware configuration B. From the TM-host we sent an isochron comprised of a number of messages greater than bucket_size. We then sent a few tokens to advance the SIU’s clock far enough to generate the EOP marker. The count in the EOP marker matched with the number of messages that were sent, even though count was greater than bucket_size.

This was verified for both SIUs.

20.D/7/89

An SIU does not unnecessarily delay non-isotach traffic.

We did not address requirement 20 during this analysis because it expresses a performance goal rather than a functional requirement. This requirement should be addressed in the future during performance studies.

21.A/7/98

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-net or both are net-host) if m1 arrives at the SIU before m2, the SIU sends m1 before m2.

We sent a series of isotach messages with unique message bodies from the network side of the SIU to the host side and then another series from the host side to the network side. We performed these tests with logical time both stopped and progressing because we wanted to verify that the messages were delivered in FIFO order regardless of whether tokens were interspersed between the messages. For each of the four cases, we confirmed at the receiving end that the messages came through in the correct (FIFO) order.

This test used the req21 testing tool (see section 4.1.2) running under hardware configuration B. The input scripts were req21/test21a.input (ran on host B, sent to the TM-host), test21b.input (ran on the TM-host, sent to host B), and test21c.input (ran on the TM-host, sent to host B). We ran test21a.input twice, once with logical time progressing and once with logical time stopped. (We used the TM-host to pause logical time.) All of the scripts sent twenty-two messages; the number of messages was selected to be sufficiently large to demonstrate FIFO ordering, but small enough that a human could verify the order of the messages.

Both SIUs passed this test. (On SIU #1, we only ran test21a.input with logical time progressing.)

22.B/7/97

The SIU sends non-isotach messages in FIFO order.

This requirement is very similar to requirement 21, and the tests were nearly identical as well. We followed the same procedure as that for requirement 21, except the scripts were req21/test22a.input (ran on host B), test22b.input (ran on the TM-host), and test22c.input (ran on the TM-host).

Both SIUs passed this test. (On SIU #1, we only ran test22a.input with logical time progressing.)

23.C/8/92

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.

The test for requirement 23 followed the same procedure as that for requirement 21. We ran sr on the TM-host to view the order of the packets that were received, and to send the packets we ran req21 on host B, using the input script req21/test23a.input. The input script has a mixture of twenty-two messages and tokens, enough to demonstrate FIFO ordering, but small enough that a human could verify the order. A token manager does not normally receive messages, but in this case the TM-host was used to print out the messages after they passed through the SIU. Any other host could have done the same thing.

The script test23b.input was used to test the ordering when packets were sent in the other direction: from the network side to the host side. We ran sr on host B and req21 on the

TM-host, and verified the order of the packets that host B received.

The order was verified in both directions for both SIUs.

24.A+/6/92

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

Requirement 24 should hold true for every isochron, so we sent isochrons of varying sizes and with different destinations through the SIU. We wanted to verify that varying isochron size and destination would not cause messages within the same isochron to receive different timestamps. We checked the timestamps that the individual messages within isochrons received and verified that all of the messages in the same isochron were assigned the same timestamp.

We used hardware configuration A2, with the addition of a host on port 5 of the switch to receive messages (using sr) that host A sent. On host A, we ran req21 with the input script req21/test24.input. The test24.input script dictates that several isochrons with different sizes and compositions be sent. The number of isotach messages within a single isochron varied from one (a singleton isochron) to 45. The payload lengths of the messages used in the isochrons varied from 3 to 213 bytes. Some of the isochrons in the script contained messages that were all sent to the same destination. Some isochrons contained messages that were sent to different destinations (up to 5 different destinations in the same isochron). In the isochrons that contained messages sent to different destinations, the lengths of the routes to the different destinations also varied, from one routing byte to thirteen routing bytes. (See requirement 27 for details on how we tested with long routes on a small hardware setup.) We did not test all possible combinations of these variations, but we tested enough to convince ourselves that requirement 24 was being satisfied. In total, test24.input sends 64 isotach messages comprising 5 isochrons.

This was verified for both SIUs.

25.A+/7/91

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.

We set the sendclock in the SIU to a known value and sent isochrons that had various isochron distances. For each isochron, we verified that the timestamp assigned to it was not less than the value specified in requirement 25.

We set the sendclock of SIU B to a known value by using the method described in requirement 4 (hardware configuration B, b## option of sr on TM-host). Host B sent isotach messages through SIU B to host A where the timestamps could be viewed. For each message, the known sendclock value, the send delta, and the isochron distance were added together, and we verified that the timestamp was not less than the sum. We sent messages with isochron distances ranging from one to six routing bytes, and with send deltas ranging from zero to the maximum value of fifteen. We also verified this requirement for the condition where the sendclock value was near its maximum and the timestamp calculation had to wrap around.

This was verified for both SIUs.

26.C/9/80

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.

The test for this requirement is very similar to the test for requirement 25. It differs in that the timestamp verification is more complicated.

Using the same setup as in requirement 25, we set the sendclock of SIU B to a known value, and then sent part of an isochron from host B to host A. After sending part of the isochron to SIU B, we used the b## option on the TM-host to increase the sendclock value of SIU B. We did this because we wanted to verify that the timestamp calculation was not based on the sendclock value when the first part of the isochron was sent. After increasing the value of the sendclock, we sent the last part of the isochron from host B to SIU B and SIU B then sent the isochron to host A. At host A, we checked the timestamp value that the messages in the isochron received. Because logical time was stopped, we were able to verify that the timestamp was less than the sendclock at the time the last flit in the isochron was sent plus the isochron distance.

This test was performed for both SIUs.

The process described above partially tests this requirement, but does not address the send delta issue. We did not test the send delta issue. In order to do so, we would use hardware configuration B with logical time progressing and have host B send a long isochron of multiple messages to the TM-host. (By doing this, the TM-host would be serving as both the token manager and the receiver of isotach messages). We could then see the interleaving order in which SIU B sent the tokens and isotach messages, since the Myrinet network maintains FIFO ordering. It would be necessary to verify that the number of tokens the SIU sent between the time it started sending the isochron and before it finished sending the isochron did not exceed the send delta value.

27.A/7/95

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.

From the hardware specification:

> 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.)

It may sometimes be useful to use a weaker consistency model in order to improve performance. Requirement 27 addresses weaker consistency models by describing when a timestamp of an isochron should necessarily be later than the timestamps of previously sent isochrons, and when the timestamp calculation should be made without regard to previous isochrons. To test the requirement, we sent two isochrons, the first one with a large isochron distance and the second one with a small isochron distance. We sent them in a way such that if the timestamp calculation for the second isochron is made without regard to the first isochron's timestamp, then the second isochron's timestamp will be earlier than the first isochron's timestamp. By using this technique, we were able to determine that whenever the second isochron's timestamp was later than the first isochron's timestamp, it was because the SIU was enforcing sequential consistency.

Using the same setup as in requirement 25, we set the sendclock of SIU B to a known value. With logical time paused, we sent a singleton isochron from host B to host A, but made it appear to go to a far away host. That is, we gave the message a long route (e.g. 26000000 00BE8181 81820600 00000000 AAAAAAAA). The switch consumed one of the routing bytes, and host A received the message with the other four routing bytes still attached. We sent a second singleton isochron, this time with the normal, one-byte route (e.g. 2E000000 00BE0600 00000000 AAAAAAAB). The second message was in the opposite concurrency set from the first isochron (different seqcon bits), but in the same channel (same channel bit). The timestamp for the second isochron was no earlier than that of the first isochron, even though (sendclock + isochron distance) was smaller for the second isochron.

We performed the test again but with the variation that the logts bit of the first isochron was switched. (The first byte of the first message was 24.) With the logts bit switched, the isochron was not "logged" and would not affect the timestamp calculations of the following isochrons. As expected, the timestamp for the second isochron was earlier than that for the first isochron.

We performed the test a third time with the seqcon bit on the second isochron switched. (The first byte of the second isochron was 2A while the first byte of the first isochron was still 26.) With the different value for the seqcon bit, the second isochron does not take into consideration the timestamp of the first isochron. Under these conditions, the timestamp for the second isochron was earlier than that for the first isochron.

This was verified for both SIUs.

28.D/6/91

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].

We first verified that the host-timestamp functionality works correctly by setting the host-ts bit of a singleton isochron and checking that the SIU did not alter the timestamp when the isochron was sent. We then sent an isochron with a large GAP field and verified that the timestamp of the next isochron we sent was calculated with the GAP taken into consideration.

In order to verify that the host-timestamp bit was working, we set SIU B to host mode and then set the SIU's sendclock to a known value using the technique described in requirement 5. We then had host B send host A a singleton isochron with the host-ts bit set. At the host A, we printed out the isochron that was received and verified that the SIU did not alter the timestamp.

The host-ts was observed working on both SIUs.

To verify requirement 28, we set SIU B's sendclock to a known value (see requirement 5). After setting the sendclock, we sent a logged isochron with the host-ts bit set and a GAP value of 15 from host B to host A. We then sent a sequenced message in the same channel but the other seqcon set. It received a timestamp no less than (sendclock + GAP from the previous message + isochron distance). From this we inferred that the GAP field from the previous message was copied into gap[x][y], which was used for the timestamp calculation.

We repeated the test with the variation that between the sending of the first and second isochrons, we increased the value of SIU B's sendclock by using the b## option of sr to exchange a few tokens. As the sendclock value was increased, gap[x][y] should have been correspondingly decreased. The timestamp on the second isochron would then be (sendclock + (GAP – number of tokens exchanged) + isochron distance). However, since the sendclock value was incremented by the same amount that gap[x][y] was decremented, there was no difference in the timestamp assigned.

It is worthwhile to note that whenever the GAP field of the SOI message is copied, gap[x][y] should be receiving a value larger than its previous value. It is the software's job to ensure that this is the case; the hardware does not necessarily check to make sure that this is true.

This was verified for both SIUs.

29.C/8/92
> The SIU assigns each isochron the earliest possible TS which satisfies the requirements given above.

Minor violations of this requirement would have an impact only on performance, but major violations could cause a violation of requirement 15. In order to test it, we used the same tests that were used to verify requirement 27. In requirement 27, we had verified that the timestamps assigned were late enough to satisfy the requirements given above. Here we verified that they were as early as possible.

This requirement was verified for both SIUs.

30.B/8/96
> 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 TS it assigns the isochrons are not increasing).

Isochron markers serve several functions, including informing the host of the timestamp that was assigned to the isochron that was sent. The exception to this is when the host-ts bit is used in an isochron, and the timestamp in the corresponding isochron marker cannot be trusted. The software must keep track of the timestamps in those situations, contrary to what was written in the hardware specification.

We sent isochrons with different isochron distances in order to generate a sequence of isochrons where their timestamps were not strictly increasing with relation to the order in which the isochrons were sent. We then verified, based on the iso id's in the isochron markers, that the isochron markers sent to the SIU's host were sent in the correct order.

We wanted to perform this test with logical time progressing, but we did not want the isochrons' timestamps to be strictly increasing. In order for the isochrons to receive timestamps that are not strictly increasing, they must be sent very rapidly, that is, they cannot be sent by hand with the sr tool. A program was written to automatically send these isochrons with varying isochron distances. It can be found at /home2/isotach/test/req30/stress.c. There are two factors, the speed at which the isochrons are sent and the fact that isochrons are different isochron distances, that combine to cause some timestamps to be greater than their preceding timestamp, some to be smaller, and some to be equal. We used hardware configuration A2. We ran the req30/stress script on the host connected to the SIU we were testing, with the SIU set to SIU mode. We set the other SIU to host mode in order to slow down the progression of logical time and help prevent the timestamps from being assigned in strictly increasing order. We checked the order of the isochron markers that the SIU sent back to the host, and verified that the markers were ordered by increasing iso id, but not necessarily by timestamp.

This was verified for both SIUs.

31.D/9/50

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.

Requirement 31 is included in order to address a particular type of flow control problem. The problem that may arise is that the buffers for holding out-going isochrons on the SIU may fill up. We did not verify this requirement directly, since it deals with timing issues in areas of the hardware that we could not directly observe. However, we were able to use the req31 program to make some observations that are consistent with the way the SIU should perform if it is satisfying this requirement.

The req31 program sends out a large isochron that is made up of many small messages. While it is sending the large isochron, it monitors how much data it has sent so that it knows how much space should be left in the buffers on the SIU. When the buffers reach a certain limit (i.e. whenever the space available is below X number of bytes), it sends an end of isochron (EOI) message. While the space available is below X, the program will continue to send messages after sending the EOI message; each of the subsequent messages the program sends will also be an EOI message. The program will notice if there is ever no remaining space in the buffer and will not send any more messages until it receives the isochron marker back from the SIU. Once the isochron marker comes back from the SIU, the program will send another large isochron containing many small messages.

If the program runs with a larger X, then the large isochron that is being sent is smaller (since an EOI message will be sent sooner). If the isochron is smaller, then the SIU should be able to send it out to the network faster. If the SIU is waiting until the entire message is sent to the network before sending the isochron marker back to the host (as requirement 31 states), then it should take longer to get an isochron marker back if the marker is for a larger isochron.

We ran the req31 program with several values of X. When the program was run with X=500, it was consistently able to send out three or four EOI messages before the SIU sent back the first isochron marker. When we made X=7500 the program was not able to send out any EOI messages before getting back an isochron marker. With X=3500 the program was only able to consistently send out two extra EOI messages before receiving an isochron marker. From this we inferred that the reason the isochron marker did not come back as quickly in some cases was because the SIU was satisfying requirement 31, and waiting to send the isochron marker until the isochron was sent to the network.

This test is only a software level argument and does not actually look at the timing on the hardware; it does not establish that the hardware is meeting this requirement, but if the hardware does meet this requirement, these are the results we would expect to see. The test was run on both SIUs.

32.D/9/50

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.

Throughout all of our testing, we always received zeros in the CRC fields of the isochron markers, which is what is normal for an isochron that does not contain a CRC error. In order to establish that the CRC field is indeed the non-exclusive OR'ing of the CRC fields of all the messages of the isochron, it would be more convincing to see a CRC error on a message reflected in the CRC field of the corresponding isochron marker. However, CRC errors are very rare hardware errors that we were not able to generate on an isotach message.

33.A/5/90

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.

The 33_35/stress.c program (see section 4.1.4) was written to test requirements 33 through 35. The program uses sort vectors of random lengths for each pulse, and this serves to help verify requirement 33 for the following reason. For each time pulse, the sort vector in the EOP marker was the correct length for that time pulse. Since this was true for each of millions of EOP markers the SIU sent during the stress tests, and the sort vectors are not all the same length, we can be confident that the sort vector in each EOP marker is the one for the corresponding time pulse.

In the 33_35 stress test, the decision of when one host sends an isochron to the other host is determined randomly. However, the average rate at which isochrons are sent is a tunable quantity. At lower average rates, the program ran without error. As we increased the average rate, we eventually ran into a problem that appeared to be directly related to the Myrinet flow control issue that is discussed in section 6.

Aside from the problems we observed at higher rates, this requirement was verified for both SIUs.

34.A/5/90

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.

To test this requirement we used the 33_35 stress test (section 4.1.4). The 33_35 stress test verifies requirement 33 because if the EOP marker contained the sort vector for a different pulse, the count field would not match with the number of messages and markers received. Similarly, the stress test verifies requirement 34 because if the EOP marker does not contain an entry for each message and marker received then the count field in the EOP marker would not match the count the program was expecting. If an element would ever be missing from the sort vector, the program would detect it and report the error.

Both SIUs have successfully run the 33_35 program. (See requirement 33.)

35.A/7/91

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.

We used the 33_35/stress.c program to test requirement 35. In that program, the received messages and isochron markers are printed out, as are the sort vectors. With this information it is possible for a human to visually verify that the sort vector of an EOP marker is correct, based on the messages and isochron markers that were received.

We checked the sort vectors for ten EOP markers of varying lengths, including the condition when the number of packets received exceeded bucket size. This requirement was verified for both SIUs.

36.C/9/50

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).

We did not thoroughly test this requirement because functionality regarding the send delta was deemed low priority. We did not, however, notice any indication that this requirement was not being met.

37.C/6/91

In SIU mode, the SIU zeroes 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.

The tests for requirement 10 demonstrated that the EOP markers are delivered at the correct logical times. During those same tests, we also verified that if a signal token was received by the SIU from the network, the appropriate signal bit was set for the EOP marker at the end of that epoch.

When no signal tokens are sent, the signal fields of EOP markers are always zeroed out. We saw this in the test for requirement 10.

To verify that the SIU always zeroes out the signal field in EOP markers that are not at the end of an epoch, we sent an isochron to the SIU. We then sent a token with a signal bit set. We controlled logical time using the technique described in requirement 4, and advanced logical time until the isochron's timestamp was reached. The SIU sent an EOP marker for the isochron's timestamp, but that timestamp did not mark the end of an epoch. The signal bits of the EOP marker were therefore zeroed out. At the end of the epoch, another EOP marker was generated, which had the appropriate signal bit set.

This was verified for both SIUs.

38.B/5/91

In SIU mode, the SIU zeroes 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.

The barrier_stress program tests this requirement (see section 4.1.3). If the SIU would ever fail to zero out the barrier field, the host would see extra EOP markers with barrier bits set. These EOP markers would be the EOP markers generated by isotach messages that should not have barrier bits set. The barrier_stress test is designed to check for that condition. Each host sends messages to the other host. In each message, the payload of the message includes the number of barriers that were completed before the message was sent. If the receiving host has seen more completed barriers than the body of the messages says should have been seen, then a barrier bit was not cleared out when it should have been.

The barrier_stress program has run successfully on all six SIUs.

39.C/4/95

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.

Tokens passed through an SIU from the network side to the host side. At the host side, we verified that the barrier and signal bits were the same as they were when they were sent from the network side.

We used hardware configuration B. From the TM-host, we used the 'w' and 'g' options of sr to toggle on and off one of the barrier bits and one of the signal bits as we sent tokens one at a time to SIU B. We verified that the signal and barrier bits in the EOP marker that host B received corresponded to the signal and barrier bits of the original token. We verified this with both a signal bit and a barrier bit set, but we did not test all of the barrier and signal bits.

This was verified for both SIUs.

5 Challenges

This project presented a number of challenges. One challenge was in developing tests to convince ourselves that the hardware was working correctly, particularly when we needed to confirm the sequence of events on the hardware while we were treating the hardware as a black box. Throughout the testing process, one of the biggest difficulties we faced was the interaction between untested software and untested hardware. If a test would fail, it was difficult to identify whether hardware or software caused the problem. One specific limitation in the Myrinet network proved to be especially difficult to identify.

The asynchronous, distributed nature of our system made the debugging process unusually difficult. When an error was encountered, it was necessary to analyze the interactions of all of the components in the system, commonly including the relative order of events at different locations. Additionally, the stochastic nature of the system often made errors difficult to reproduce.

Whenever an error was isolated to the hardware, the two primary causes were physical problems and errors in the VHDL of a chip on the SIU. Physical problems were generally straightforward to identify because they frequently had drastic consequences, practically halting the system. Occasionally, while probing for a physical problem, the probe itself would actually fix the problem (by pressing together a loose connection, for instance), but as long as the problem persisted during the probing process, locating the problem was relatively automatic. Logic errors in the VHDL were more elusive, particularly when there was interaction between multiple errors. One problem caused by the interaction of multiple errors dealt with epoch lengths and the delivery of signals. One error was in the VHDL on the SIU and the other was in the PROM on the token manager. The problem with the token manager was with the encoding for the epoch length; the epoch length should be encoded with the value of the last timestamp of the first epoch rather than the first timestamp of the second epoch as was originally programmed. Having the error in the TM present made it more difficult to identify what the SIU was doing, which also happened to be incorrect behavior.

Another complication did not fall into either the category of hardware or VHDL: the fact that Myrinet does not ensure that all packets reach their destination. There is always a timeout value after which Myrinet will give up and discard the packet. We now have this timeout value set to as high a value as possible (4 seconds) on the LANai cards. Before we considered the possibility of Myrinet dropping packets, we had several unidentified bugs, some very sporadic and hard to reproduce. We went through quite a number of theories about what might have caused them. It turned out that many of them could be attributed to this Myrinet flow control issue. The flow control issue is still present, but for the most part we are able to work around it by not overloading the host processor, by increasing the timeout value as high as possible and by favoring receiving over sending [LM00].

A final difficulty was the sporadic nature of some problems. One problem that was never resolved involved a test that ran perfectly for seven hours and then had a one-byte message erroneously appear. We never figured out what caused it and the error was not reproducible. A number of errors have fallen into this category, though as testing progressed the sporadic bugs became less frequent.

These examples illustrate the general difficulty that is found when dealing simultaneously with untested prototype hardware and newly written software. The main approach we took to overcome these challenges was the well-known principle of simplicity. For any exhibited error, we removed code and eliminated steps in the process until we had the simplest test case that still exhibited the problem. We could not remove parts of the hardware, but simpler test cases made the testing process easier. It became easier to trace the probable state of the hardware, to determine the best places to probe the hardware to gain more information, and to deduce (from the probe data and hardware output) the errors the hardware may have made.

6 Recommendations

6.1 Hardware Issues

The following are the known remaining issues with the hardware of the current isotach implementation. As mentioned in section 5, there are occasionally sporadic errors of unknown cause, which are not reproducible.

1. The hardware TM does not always come up in a good state. In order to ensure that the TM is in a good state, the TM should be turned on before the SIUs are turned on. On power up, the TM's token traffic light should go through a series of quick changes, from yellow to green to red. If the light is not red before turning on the SIUs then the system may or may not be in a good state. Additionally, the reset buttons on the token managers do not work properly. They do not send out the initial barrier bits and they do not start on the right sequence numbers. To reset the system, either turn off all of the hardware and bring it up in the order described here, or use a reset token for a software reset.
2. There are limits to Myrinet's hardware flow control. There is always a possibility of dropped packets. If one entity in the network tries to send a packet but is unable to do so due to flow control, that entity will continue trying to send the packet until some timeout value. We would like that timeout value to be unlimited, but it can only be as large as 4 seconds for the LANai cards. We do not know the timeout value for the FIs on the SIU, and do not see a way to change it, although we have not confirmed with Myricom that it is hardwired. Under heavy loads we can lose data due to this problem, but taking this into consideration during software design can make this problem less likely to occur. The flow control issue exhibits itself on some of the stress tests if they are run with high data rates.
3. When the SIU sends isochron markers or EOP markers to the host, the SIUs sometimes send back an extra byte for no apparent reason. Isotach software is able to recognize that this is spurious data and can then ignore it, but it would be better to not have this problem at all.
4. When an isochron has the host-ts bit set, the corresponding isochron marker has an invalid timestamp. The timestamp in the isochron marker should be the same as that in the isochron, but instead it is some other value. We did not determine what that other value is. It might be the timestamp that the SIU would have calculated if the host-ts bit was not set, or it may be a random value. Either way, the software must remember the timestamp that it set and not trust the isochron marker.

6.2 Suggested Improvements

The following issues, while not unmanageable problems with the system, are issues that could be addressed in future hardware implementations to help isotach run more smoothly.

1. The TM does not always respond to resets that come from "invalid" hosts, that is, from hosts that the token manager does not recognize as part of the isotach system. We would like to be able to reset the system from a host that is not in the system in order to do so more easily.
2. If a program ends abnormally, it may be in the middle of sending an isochron; this can prevent the next program from starting correctly. The problem can be handled easily in software by first sending a null isotach message with the EOI bit set. That will end an open isochron if there is one, and perform no function if there is not an open isochron. Conceptually, it would be cleaner if the SOI messages had an SOI bit to indicate the start of isochron in a way that is distinguishable from mid-isochron messages.
3. The implementation details regarding epoch lengths are counter-intuitive. On the token manager, the "epoch length" that should be programmed on the configuration PROM is actually "epoch length – 1". For example, an epoch that is 32 time pulses long would be programmed as x1F instead of x20. Using x1F specifies the first epoch as time pulses x00 through x1F, which is 32 time pulses. While the token manager's epoch length can be any desired length, the SIU's epoch length is constrained to the values that can be encoded with the dip-switches: 2, 4, 8, 16 and 32. Additionally, when the SIU receives a barrier/signal marker with a signal bit set, it will not send the signal to the token manager until one time pulse into the epoch. For example, if the epoch length is 32, and a barrier/signal marker is sent at x1A, the signal token should be sent at the beginning of the next epoch, which would be x20. However, it is not sent until timestamp x21. This might present a problem in the system if it was overly aggressive in selecting a short epoch length to reduce signal latency. However, if the epoch length is sufficiently large relative to the diameter of the network, this will not present a problem. In future versions, the details regarding epoch lengths and signal delivery should be more intuitive.

7 Summary

The goal of this project was to design and perform tests that convincingly demonstrated whether or not an SIU satisfies the functionality requirements in the SIU's hardware specification. We designed the tests and performed them on two SIUs, which passed the tests. We began performing the tests on four other SIUs. Our work shows that the hardware implementation essentially meets the functional requirements of the design. Also, our prototype implementation of an isotach network with custom hardware components serves as a proof of concept.

With properly functioning SIUs and token managers, it is possible to develop applications and carry out performance tests for an isotach network. These are tasks to be completed as part of future work.

Appendix A: Source Code

The source code for the test programs can be found on pepe.cs.virginia.edu in the following directories:

sr - /home2/isotach/test/send_receive

barrier_stress - /home2/isotach/test/barrier_stress

req21 - /home2/isotach/test/req21

req30 - /home2/isotach/test/req30

req31 - /home2/isotach/test/req31

33_35 - /home2/isotach/test/33_35

Appendix B: Using the sr testing tool

Command line parameters

When executing the sr program, there are two mandatory command line parameters. The first is the offset to the port where tokens should be sent when the program exchanges tokens. If sr is running on a TM-host, then this is the offset to the SIU with which sr will communicate. (If sr is running on a TM-host, the TM-host will only communicate with one SIU at a time.) If sr is not running on a TM-host, then the first parameter is the offset to the token manager. The second command line parameter is a Boolean (1/0) value indicating whether or not there is a hardware SIU between the host and the network. This value is used when the hardware SIU is running in host mode and the host needs to send tokens with a header. If a user does not intend to use the token exchange features of the sr program, then it does not matter what values are entered for the two command line parameters.

Commands in sr

After sr is started, the user is presented with a number of options.

s : Entering 's' will allow the user to send a packet. The packet should be entered in hex notation with no spaces. The user should not enter the CRC byte.

r : After sending the first packet, the user can enter 'r' to resend the last packet that was sent.

o : This option will toggle the output of incoming traffic from the network. If traffic is not printed, it is still received but it is discarded.

t : This option will toggle the output of token traffic. Because token traffic can reach such a high volume, it is sometimes useful to turn off the output of tokens but leave on the output of other network traffic.

b : The 'b' here stands for token bouncing, or token exchange (explained in section 4). This option toggles token exchange. When sr is exchanging tokens, it will send a token to the network each time it receives a new (non-repeat) token from the network. Entering 'b' a second time will turn off token exchange.

b## : Entering b followed by a number, with no space, will exchange (bounce) the specified number of tokens and then stop. For example, 'b16' will exchange sixteen tokens. Valid numbers are 1-99.

w : This option will toggle a barrier bit on tokens that are automatically exchanged. It controls the second barrier bit (the last bit in a token).

v : Similar to option 'w', this option will toggle the other barrier bit on tokens that are automatically exchanged. It controls the first barrier bit.

g : This option will toggle a signal bit on tokens that are automatically exchanged.

c : Entering 'c' will print out the status of some of the variables in the program.

Specifically, the program will print out the value of barrier status variables, the signal status, the send and receive clocks, and the count of the number of packets sent. The barrier and signal status bits are 1 if set and 0 if not set.

Bibliography

[Bar99] Bartholet, Robert. A Performance Study of Isotach Version 1.0. Technical report, University of Virginia, April 1999.

[BCF95] Boden, Nanette J., Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet – A Gigabit-per-Second Local-Area Network. *IEEE Micro*, 15(1):29-36, February 1995.

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

[LM00] Lack, Michael and Perry Myers. The Isotach Messaging Layer: Ironman Design. Technical report, University of Virginia, 2000.

[New98] Newfield, Dale. Integration & Testing of the Hardware Token Managers. University of Virginia Master's Project, December 1998.

[Reg97] Regher, John. An Isotach Implementation for Myrinet. Technical report, University of Virginia, May 1997.

[RWW97] Reynolds, Paul F. Jr., Craig Williams and Raymond R. Wagner, Jr. Isotach Networks. *IEEE Transactions on Parallel and Distributed Systems*, 8(4), April 1997.

[Sza99] Szajda, Doug. Testing the Isotach Prototype Hardware Switch Interface Unit. Technical report, University of Virginia, June 1999.

[HWSpec] Williams, C. C. Design of the isotach prototype. Internal Working Paper.