April 1996

Order Number: 312824-004

Paragon™ System High Performance Parallel Interface Manual

Intel® Corporation

Copyright ©1996 by Intel Server Systems Product Development, Beaverton, Oregon. All rights reserved. No part of this work may be reproduced or copied in any form or by any means...graphic, electronic, or mechanical including photocopying, taping, or information storage and retrieval systems...without the express written consent of Intel Corporation. The information in this document is subject to change without notice.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update or to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

Intel software products are copyrighted by and shall remain the property of Intel Corporation. Use, duplication, or disclosure is subject to restrictions stated in Intel's software license agreement. Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraphs (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at 252.227-7013. Intel Corporation, 2200 Mission College Boulevard, Santa Clara, CA 95052-8119. For all Federal use or contracts other than DoD, Restricted Rights under FAR 52.227-14, ALT. III shall apply.

The following are trademarks of Intel Corporation and its affiliates and may be used only to identify Intel products:

286	i386	Intel	iPSC
287	i387	Intel386	Paragon
i	i486	Intel387	
	i487	Intel486	
	i860	Intel487	

Other brands and names are the property of their respective owners.

WARNING

Some of the circuitry inside this system operates at hazardous energy and electric shock voltage levels. To avoid the risk of personal injury due to contact with an energy hazard, or risk of electric shock, do not enter any portion of this system unless it is intended to be accessible without the use of a tool. The areas that are considered accessible are the outer enclosure and the area just inside the front door when all of the front panels are installed, and the front of the diagnostic station. There are no user serviceable areas inside the system. Refer any need for such access only to technical personnel that have been qualified by Intel Corporation.

CAUTION

This equipment has been tested and found to comply with the limits for a Class A digital device, pursuant to Part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment. This equipment generates, uses, and can radiate radio frequency energy and, if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications. Operation of this equipment in a residential area is likely to cause harmful interference in which case the user will be required to correct the interference at his own expense.

LIMITED RIGHTS

The information contained in this document is copyrighted by and shall remain the property of Intel Corporation. Use, duplication or disclosure by the U.S. Government is subject to Limited Rights as set forth in subparagraphs (a)(15) of the Rights in Technical Data and Computer Software clause at 252.227-7013. Intel Corporation, 2200 Mission College Boulevard, Santa Clara, CA 95052. For all Federal use or contracts other than DoD Limited Rights under FAR 52.2272-14, ALT. III shall apply. Unpublished—rights reserved under the copyright laws of the United States.

Preface

The American National Standards Institute (ANSI) has standardized a high speed external connection for supercomputers. That product is called High Performance Parallel Interface, or HIPPI. This manual describes the Paragon™ system HIPPI controller and explains how to install and configure the controller in a Paragon system.

NOTE

Because Paragon™ system HIPPI controllers are supported by both Paragon™ XP/S Systems and Paragon™ XP/E Systems, this manual discusses the HIPPI controller (as part of a Paragon system) in generic terms.

Audience

This manual has two audiences:

- Intel Customer Support Engineers who have completed the *Paragon System Site Support* course will be primarily interested in those portions of the manual that discuss installation procedures (Chapters 3, 5, and 6).

- Engineers and system administrators who need to understand how the HIPPI controller works on their system will be primarily interested in the portions of the manual that discuss HIPPI packets, the raw HIPPI interface library, *libhippi.a*, the HIPPI commands, and usage notes (Chapters 1, 2, and 4, plus Appendices A, B, C, and D).

NOTE

This manual assumes that you understand the ANSI HIPPI specifications HIPPI-SC, HIPPI-LE, and HIPPI-FP.

In this manual, "operating system" refers to the operating system that runs on the nodes of the Paragon(TM) supercomputer.

Organization

Chapter 1	Introduces HIPPI usage, protocol, and standards.
Chapter 2	Describes the HIPPI controller, its architecture, and its operation.
Chapter 3	Explains how to install a HIPPI controller.
Chapter 4	Discusses software configuration issues (network configuration, packet building, and raw HIPPI).
Chapter 5	Discusses HIPPI diagnostics.
Chapter 6	Describes the HIPPI internal, external, and loopback (diagnostic) cables.
Appendix A	Contains manual pages for the raw HIPPI library (*libhippi.a*).
Appendix B	Contains manual pages for the HIPPI commands (**hippi_setmap** and **hippi_showmap**).
Appendix C	Contains usage notes and examples for learning how to use the raw HIPPI interfaces.
Appendix D	Contains instructions for setting up IPI-3 device drivers for use with the HIPPI interface.

Notational Conventions

This manual uses the following notational conventions:

Bold Identifies command names and switches, system call names, reserved words, and other items that must be used exactly as shown.

Italic Identifies variables, filenames, directories, processes, user names, and writer annotations in examples. Italic type style is also occasionally used to emphasize a word or phrase.

`Plain-Monospace`
Identifies computer output (prompts and messages), examples, and values of variables. Some examples contain annotations that describe specific parts of the example. These annotations (which are not part of the example code or session) appear in *italic* type style and flush with the right margin.

Bold-Italic-Monospace
Identifies user input (what you enter in response to some prompt).

Bold-Monospace
Identifies the names of keyboard keys (which are also enclosed in angle brackets). A dash indicates that the key preceding the dash is to be held down *while* the key following the dash is pressed. For example:

<Break> **<s>** **<Ctrl-Alt-Del>**

[ ] (Brackets) Surround optional items.

... (Ellipsis dots) Indicate that the preceding item may be repeated.

| (Bar) Separates two or more items of which you may select only one.

{ } (Braces) Surround two or more items of which you must select one.

Applicable Documents

For information about limitations and workarounds, see the *Paragon™ System Software Release Notes*. Release notes are also located in the directory */usr/share/release_notes* on your Paragon system.

For more information, refer to the current versions of the following manuals:

Intel Corporation manuals:

Paragon™ System User's Guide
312489
Provides an overview of the operating system. Tells how to develop and run programs.

Paragon™ System Commands Reference Manual
312486
Provides detailed information about the commands for the operating system.

Paragon™ System Hardware Maintenance Manual
312822
Provides detailed maintenance information for the Paragon system.

Paragon™ System Administrator's Guide
312544
Provides detailed instructions for system administration for the Paragon system.

Paragon™ System Diagnostic Reference Manual
312702
Provides detailed information about the configuration files *SYSCONF.TXT* and *DEVCONF.TXT*.

ANSI documents:

High-Performance Parallel Interface-Physical Switch Control (HIPPI-SC) REV 1.7
X3T9.3/91-023

High-Performance Parallel Interface-Framing Protocol (HIPPI-FP) REV 4.2
X3T6/91-146

High-Performance Parallel Interface-Physical Layer Protocol (HIPPI-PH) REV 8.1
X3T6/91-127

High-Performance Parallel Interface-Link Encapsulation Protocol (HIPPI-LE) REV 2.0
X3T9/90-119

Comments and Assistance

Intel Scalable Systems Division is eager to hear of your experiences with our products. Please call us if you need assistance, have questions, or otherwise want to comment on your Paragon system.

U.S.A./Canada Intel Corporation
Phone: 800-421-2823
Internet: support@ssd.intel.com

France Intel Corporation
1 Rue Edison-BP303
78054 St. Quentin-en-Yvelines Cedex
France
0590 8602 (toll free)

United Kingdom Intel Corporation (UK) Ltd.
Scalable Systems Division
Pipers Way
Swindon SN3 IRJ
England
0800 212665 (toll free)
(44) 793 491056
(44) 793 431062
(44) 793 480874
(44) 793 495108

Intel Japan K.K.
Scalable Systems Division
5-6 Tokodai, Tsukuba City
Ibaraki-Ken 300-26
Japan
0298-47-8904

Germany Intel Semiconductor GmbH
Dornacher Strasse 1
85622 Feldkirchen bei Muenchen
Germany
0130 813741 (toll free)

World Headquarters
Intel Corporation
Scalable Systems Division
15201 N.W. Greenbrier Parkway
Beaverton, Oregon 97006
U.S.A.
(503) 677-7600 (Monday through Friday, 8 AM to 5 PM Pacific Time)
Fax: (503) 677-9147

Table of Contents

Chapter 1
Introduction

How the HIPPI Controller is Used 1-1

HIPPI Protocol 1-3

Standards 1-4

The Physical Standard 1-4

The Switch Control Facility Standard 1-4

The Framing Protocol Standard 1-5

The Link Encapsulation Standard 1-5

Chapter 2
Theory of Operation

The HIPPI Controller 2-1

HIPPI Controller Architecture 2-3

HIPPI Controller Operation 2-4

Basic HIPPI Framing Protocol 2-4

Data Flow 2-5

Sending a Packet to Its Destination 2-7
The I-Field 2-9
I-Field Source Addressing 2-10
I-Field Destination Addressing 2-11
The Use of Switches In the HIPPI Environment 2-11

Chapter 3
Installing the HIPPI Controller

Tools Needed 3-1
The HIPPI Controller and the Node Board 3-1
Installing the HIPPI Controller 3-2
Installing the Cables 3-4
Closing the Cabinet Door 3-5

Chapter 4
Software Configuration

Network Configuration 4-1
Configuring The System For HIPPI 4-2
Configuring the Network Interface 4-2
Activating the Network Interface 4-3
Optimizing TCP/IP Performance Over the HIPPI Channel 4-4
Determining ULA Addresses for HIPPI Boards 4-5
Routing Tables 4-6
Routing Tables for Simple Networks 4-6
Routing Tables for Complex Networks 4-7
Routing Table Commands 4-8

Server Interface and Packet Building 4-8
The FP_Header_Area 4-9
The D1_Area 4-10
The LE_Header 4-10
The D2_Area 4-11
Inbound Packets 4-11
Raw HIPPI 4-11
Raw HIPPI Usage Models 4-11
Using Raw HIPPI 4-14

Chapter 5
HIPPI Diagnostics

Diagnostics 5-1
Power-On Self Test 5-1
Paragon™ System Diagnostic Program 5-1
Loopback Tests 5-2

Chapter 6
Cable Parts and Specifications

Internal Cables 6-1
Internal Cable Characteristics 6-1
Internal Cable Implementation 6-1
External Cables 6-2
External Cable Characteristics 6-2
External Cable Implementation 6-2

Loopback Cable 6-2

Loopback Cable Implementation 6-2

Appendix A HIPPI Calls

HIPPI_BIND() A-2
HIPPI_CLOSE() A-4
HIPPI_CONFIG() A-5
HIPPI_MEMFREE() A-7
HIPPI_MEMGET() A-9
HIPPI_OPEN() A-11
HIPPI_READ() A-13
HIPPI_READ_COMPLETE() A-15
HIPPI_READ_REQUEST() A-17
HIPPI_WRITE() A-19

Appendix B HIPPI Commands

HIPPI_SETMAP B-2
HIPPI_SHOWMAP B-4

Appendix C
Usage Notes

Introduction C-1

Using the HIPPI_DATA Mode C-1

HIPPI Packets C-1
FP Header C-1
hippi_open(dev_name, hippi_mode, mode) C-2
hippi_bind(ihandle, ulp, port) C-2
hippi_config(ihandle, ifield, ulp, b, d1_data, d1_len) C-3
hippi_memget(ihandle, size) C-3
hippi_write(ihandle, ptr, length) C-3
hippi_read(ihandle, ptr, bytes_wanted) C-5
hippi_memfree(ihandle, ptr, size, how) C-5
hippi_read_request(ihandle, bytes_wanted) C-6
hippi_read_complete(ihandle, ptr) C-6
hippi_close(ihandle) C-6
Code Example C-7

Using The HIPPI_RAW Mode C-13

HIPPI Packets C-13
hippi_open(dev_name, hippi_mode, mode) C-14
hippi_bind(ihandle, ulp, port) C-14
hippi_config(ihandle, ifield, ulp, b, d1_data, d1_len) C-14
hippi_memget(ihandle, size) C-14
hippi_write(ihandle, ptr, length) C-14
hippi_read(ihandle, ptr, bytes_wanted) C-16
hippi_memfree(ihandle, ptr, size, how) C-17
hippi_read_request(ihandle, bytes_wanted) C-17
hippi_read_complete(ihandle, ptr) C-18
hippi_close(ihandle) C-18
Code Example C-18

Appendix D
Using IPI Devices With Paragon™ Systems

Using The IPI-3 Interface on Paragon™ Systems D-1
The IPI Protocol D-1
Using the IPI-3 Interface D-1
System Setup Overview D-2
Device Requirements D-2
IPI Addressing D-3
Setting Up An IPI-3 Interface on Paragon™ Systems D-3
Connecting the IPI-3 Interface to a HIPPI Channel D-3
Using Bootmagic Variables D-3
Setting IPI Slave Connection Control Variables D-4
Setting IPI Slave Facility Variables D-5
Setting IPI Slave Partitions D-5
Setting IPI Command Reference Numbers D-6
Dynamic Control of IPI-3 Connections With IOCTL Functions D-7
Creating IPI-3 Device Entries D-8
Creating a Disk Label D-10
Labeling the IPI Device D-11
Creating New File Systems D-11
Adding IPI Entries to Mount Tables D-11
Mounting The File Systems D-12
Optimizing IPI-3 PFS Performance D-12

List of Illustrations

Figure 1-1. How HIPPI Can Be Used 1-2
Figure 1-2. HIPPI Protocol 1-3

Figure 2-1. The HIPPI Controller Mounted on a Paragon™ System GP Node Board 2-2
Figure 2-2. HIPPI Framing Protocol 2-5
Figure 2-3. Data Flow in a HIPPI Controller 2-6
Figure 2-4. The HIPPI Signal Diagram 2-8
Figure 2-5. I-Field Format 2-9
Figure 2-6. I-Field with Source Routing, D = 0 2-10
Figure 2-7. I-Field with Source Routing, D = 1 2-11
Figure 2-8. I-Field with Destination Address, D = 0 2-11
Figure 2-9. I-Field with Destination Address, D = 1 2-12
Figure 2-10. An 8 x 8 HIPPI Switch 2-12

Figure 3-1. Opening the Paragon™ System Cabinet Door 3-3
Figure 3-2. Removing a Node Board from the Cardcage 3-4
Figure 3-3. Installing the Combined HIPPI/Node Board in the Cardcage 3-5

Figure 4-1. Sample Network with One HIPPI Switch 4-7
Figure 4-2. HIPPI Packet Format 4-9
Figure 4-3. Link Encapsulation Packet Format Header 4-10
Figure 4-4. A HIPPI Framing Protocol Packet with an LE_Header 4-12
Figure 4-5. HIPPI Packet Incoming and Outgoing Flow 4-13

Figure 5-1. Connecting the Loopback Cables 5-2

Figure C-1. FP Header C-2
Figure C-2. The ptr and length Arguments for hippi_write (HIPPI_DATA Mode) C-3
Figure C-3. The ptr and length Values Returned by hippi_read C-5
Figure C-4. The ptr and length Arguments for hippi_write (HIPPI_RAW Mode) C-15
Figure C-5. The ptr and length Values Returned by hippi_read C-17

Introduction 1

The Paragon™ System High Performance Parallel Interface controller (called the HIPPI controller in this manual) is a daughtercard that attaches to a Paragon system node board (GP or MP node board) and provides high speed communications with other, heterogeneous systems, networks, and devices.

This chapter briefly describes how the HIPPI controller is used, how data is transmitted across the HIPPI channel, and the HIPPI standards.

How the HIPPI Controller is Used

As shown in Figure 1-1 on page 1-2, the HIPPI controller is typically used in one of three ways:

- As a *channel device* between a Paragon system and a disk farm or another mass storage device.
- As a *network device* that enables applications to communicate with remote systems by using data-communications protocols such as TCP/IP (Transmission Control Protocol /Internet Protocol). This form of communication relies on an external switch.
- As a *point-to-point device* that enables two systems to communicate.

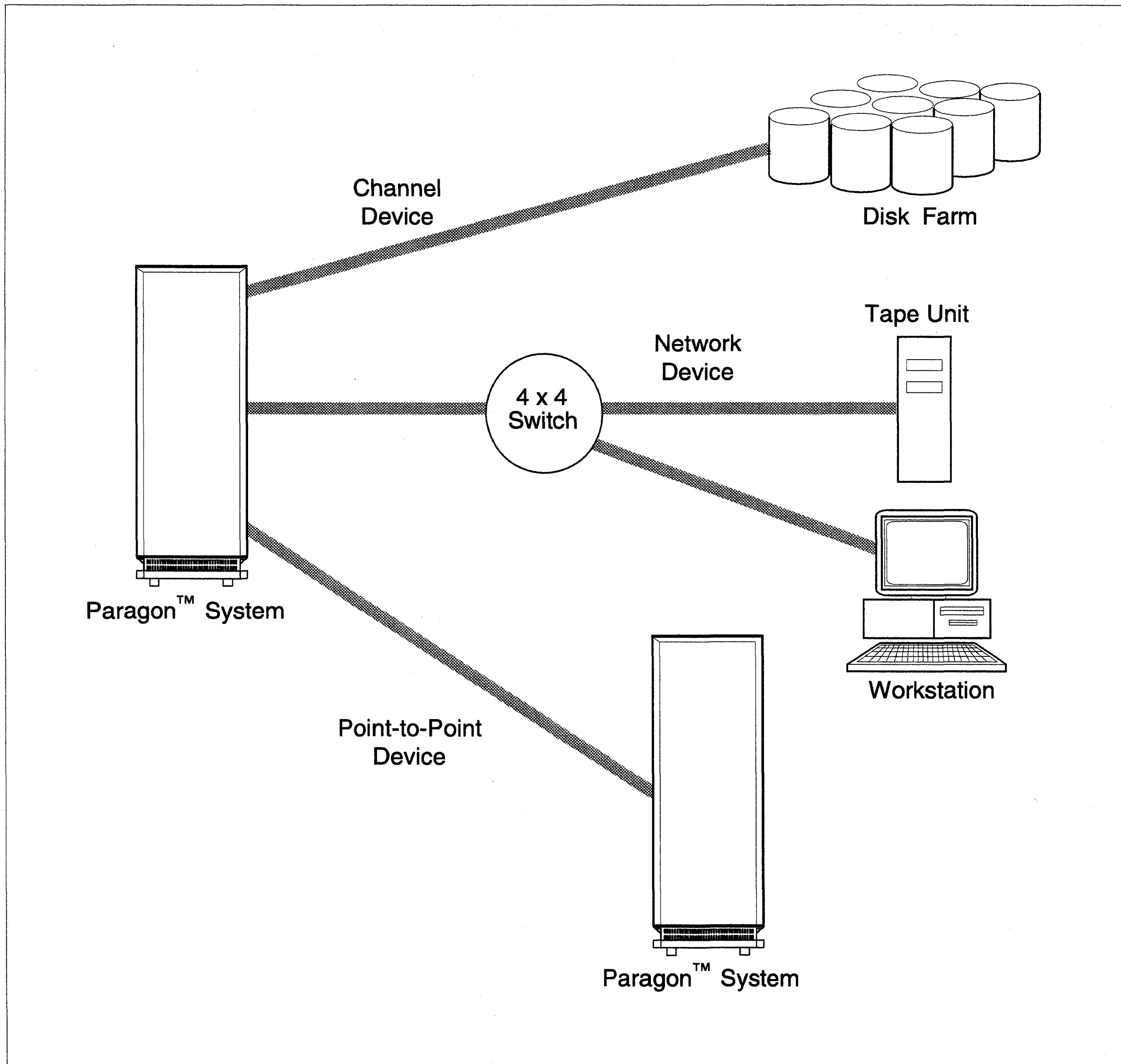

Figure 1-1. How HIPPI Can Be Used

HIPPI Protocol

The HIPPI controller transmits bursts of data using the protocol illustrated in Figure 1-2. Every time a connection is established, one or more packets are sent across the HIPPI channel. Each packet contains one or more bursts. Bursts contain the actual data and are 1 word wide (32 bits of data, 4 bits of parity) and 256 words long. To accommodate packets that are not exact multiples of 256, the ANSI standard allows either the first burst or the last to be shorter than 256 words. The Paragon system HIPPI controller sends the short burst last. Refer to Chapter 4 for more information about how the controller transmits data.

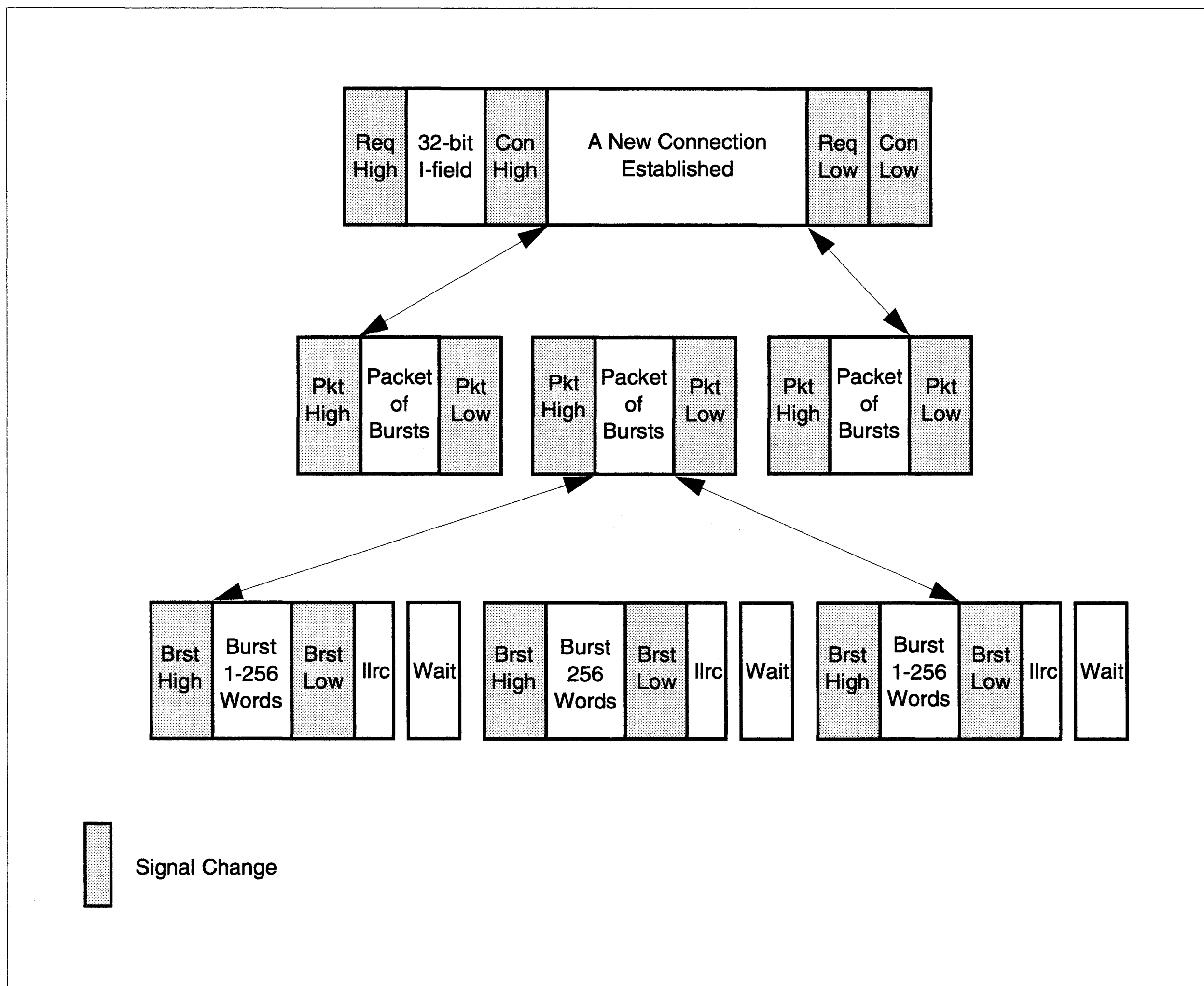

Figure 1-2. HIPPI Protocol

The number of packets-per-connection and bursts-per-packet depends upon the HIPPI mode used. The MPC mode transmits multiple packets per connection. The CNT (CoNTinuation) mode transmits multiple I/O reads and writes per connection and packet.

Standards

The ANSI HIPPI standard is composed of six individual standards that address the Physical and the Data Link layer (the lowest two layers) of the International Standards Organization (ISO) reference model. The ANSI HIPPI standard does not address the other elements of the ISO reference model. The six ANSI HIPPI standards are:

- The Physical standard (HIPPI-PH).
- The Switch Control Facility standard (HIPPI-SC).
- The Framing Protocol standard (HIPPI-FP).
- The Link Encapsulation standard (HIPPI-LE).
- The Intelligent Peripheral Interface standard (HIPPI-IPI-3).
- The Memory Interface (HIPPI-MI).

The following sections discuss the first four of these standards.

The Physical Standard

The Physical Standard defines the mechanical, electrical, and signaling protocol specifications for a one-way, point-to-point interface. The Physical Layer uses a pair of cables, which consist of a bundle of twisted-pairs, with a maximum length of 25 meters.

Data transfer occurs via data bursts. Each burst contains up to 256 words, each containing 32 bits of data and four bits of parity. A word is transferred during every 25 Mhz clock cycle. In general, the Physical Layer operates by host-based flow control (or READY signals) that regulate the transmission of each data burst. Look-ahead flow control (or sending multiple READY signals) allows the average data transfer rate to approach the peak transfer rate of 800M bits per second.

The Switch Control Facility Standard

The Switch Control Facility Standard defines the control for physical layer switching in a HIPPI environment. A physical switch provides the method for interconnecting multiple HIPPI-based systems. The Switch Control Facility provides source routing and destination addressing support along with supporting different switch sizes. The Switch Control Facility does not generate the I-field, but it defines how a switch should interpret the I-field. Refer to "The I-Field" on page 2-9 for more information.

The Framing Protocol Standard

The Framing Protocol Standard defines a common data framing protocol that supports large data transfers, a best-effort delivery mechanism with no error recovery, and an identifier field for multiplexed upper layer protocols. One HIPPI frame is equivalent to one packet. Each packet contains one or more bursts and each burst can contain anywhere from 1 to 256 words. Each word contains four bytes (32 bits of data and four parity bits). If a burst contains less than the maximum of 256 words, it is referred to as a short burst. The Framing protocol consists of three parts:

- The FP_Header_Area.
- The D1_Data_Area.
- The D2_Data_Area.

Refer to "Server Interface and Packet Building" on page 4-8 for more information about the framing protocol.

The Link Encapsulation Standard

The Link Encapsulation Standard defines the methodology by which packets of data can be transported. The standard conforms to the IEEE 802.2 and ISO 8802-2 Logical Link Control standards. The Link Encapsulation Standard also provides the methodology for sending the 48-bit destination and source addresses which conform to the IEEE 802.1A standard.

Theory of Operation 2

This chapter describes the HIPPI controller, its architecture, and its operation.

The HIPPI Controller

As shown in Figure 2-1 on page 2-2, the HIPPI controller is a daughtercard that mounts on a Paragon™ system node board via the node board's expansion interface. The controller measures 8.7 by 9.8 inches and has its own front panel connectors and LED cutouts. (The HIPPI controller's front panel replaces the front panel of the node board.)

The HIPPI controller contains the logic and connectors for two 32-bit simplex HIPPI channels. One channel is called the *source channel* (for outgoing data) and the other is called the *destination channel* (for incoming data). The HIPPI controller is a dual-simplex I/O channel. *Only one HIPPI node is required in order to achieve full dual-simplex operation.*

The HIPPI controller is a high-speed, direct memory access (DMA) device that transfers data, commands, and status between the node board memory and the HIPPI channel. The controller is designed to transfer data on both channels simultaneously. An on-board microcontroller manages channel connect and disconnect functions as well as burst-level and packet-level data transfers. The host node supplies power to the HIPPI controller through the expansion connector and mounting standoffs.

Packet descriptors in host node memory are constructed by the driver and include a pointer to a list of variable-sized memory blocks. The lists are implemented using a series of packet and buffer descriptors (data structures containing pointers and control and status bits that describe the buffers).

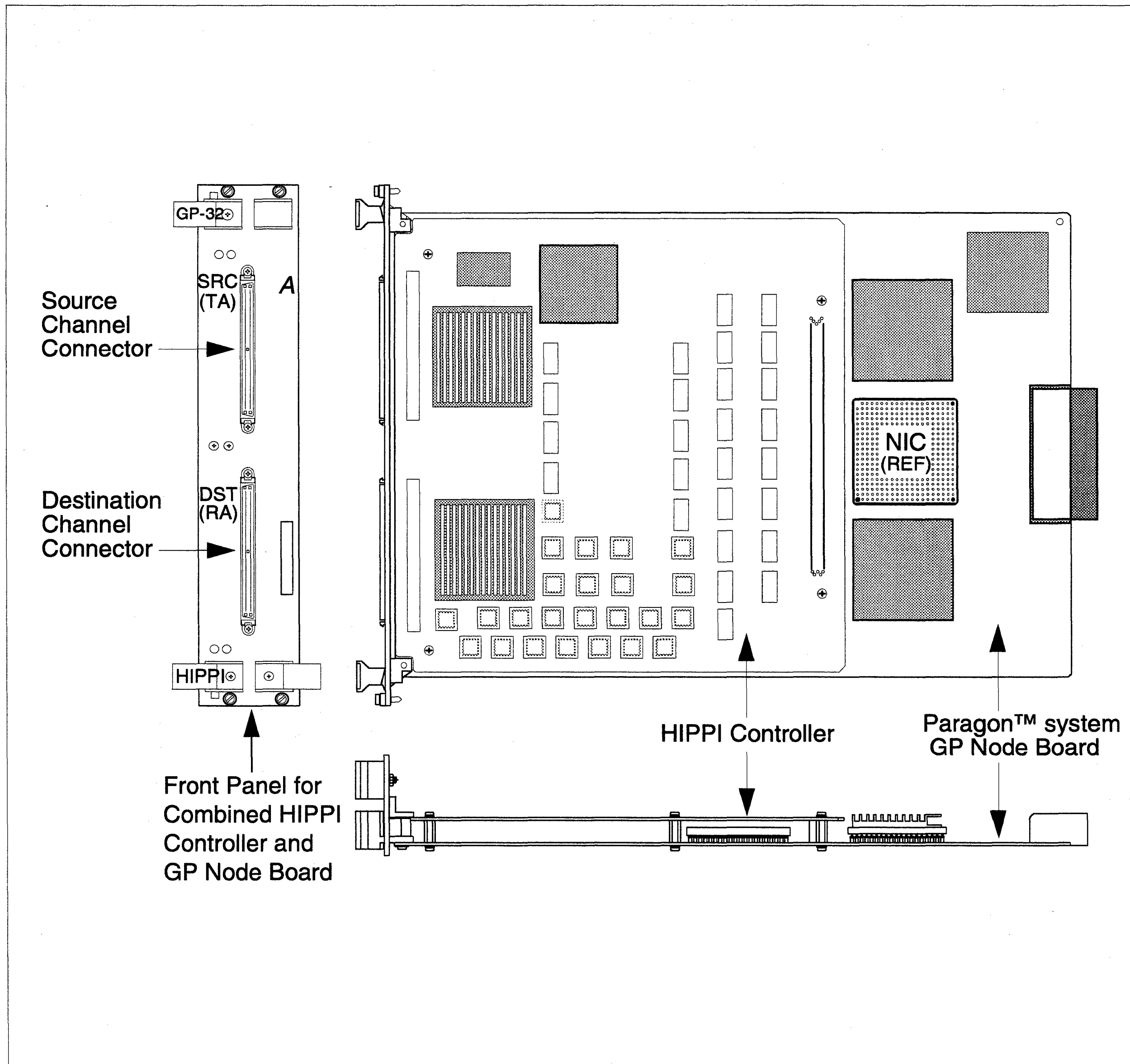

Figure 2-1. The HIPPI Controller Mounted on a Paragon™ System GP Node Board

The HIPPI controller contains the following components:

- AMCC S2020 source interface device.
- AMCC S2021 destination interface device.
- Data first-in-first-out (FIFO) memories.
- 80960CF microcontroller with 128KB of RAM.
- Flash EPROM for 80960CF code, configuration, and diagnostics.
- Local and host expansion bus control logic.
- Control and status registers.
- Direct memory access control logic.
- HIPPI cable connectors.
- Activity light emitting diodes (LEDs).
- HIPPI controller performance monitor.

HIPPI Controller Architecture

The HIPPI controller has two parts: a decision-making unit and a data-transfer unit. The decision-making unit is a microcomputer consisting of an 80960 processor, RAM, flash EPROM, and all status and control registers. The status registers provide the current status of the entire controller hardware. The control registers provide a means to exercise control over the controller hardware and to interrupt the host.

The data-transfer unit consists of DMA logic, source and destination FIFOs, and source and destination HIPPI interfaces. The DMA logic transfers *blocks* of data between the node memory and the FIFO. A block consists of a maximum of 512 64-bit word transfers that are equivalent to four 32-bit word HIPPI bursts.

The host node has complete access to the controller's hardware. The controller, however, can only access the node board's memory. If the node and controller attempt to access each other's hardware simultaneously, the 80960 processor waits until the host access cycle completes.

DMA accesses for node memory are aligned on a cache-line boundary and are not cache-coherent. However, the 80960 processor access of node memory could be from any byte and is cache-coherent. The host should not cache data that it shares with the DMA, but it can cache its shared data with the 80960 processor. Data shared with the DMA includes the HIPPI data that will be sent out or has been received. Data shared with the 80960 includes data structures.

HIPPI Controller Operation

This section discusses the following:

- Basic HIPPI framing protocol.
- Data flow in a HIPPI controller.
- Packet transmission.
- The I-field.
- Switches in the HIPPI environment.

Basic HIPPI Framing Protocol

The HIPPI controller transmits bursts of data using the protocol illustrated in Figure 2-2 on page 2-5. Every time a connection is established, one or more packets are sent across the HIPPI channel, depending upon the HIPPI mode being used. Each packet contains one or more bursts, also depending upon the HIPPI mode used. Bursts contain the actual data and are 1 word wide (32 bits of data, 4 bits of parity) and 256 words long. To accommodate packets that are not exact multiples of 256, the ANSI standard allows either the first burst or the last to be shorter than 256 words. To implement the standard, the Paragon system HIPPI controller sends the short burst last. Refer to Chapter 4 for more information about how the controller transmits data.

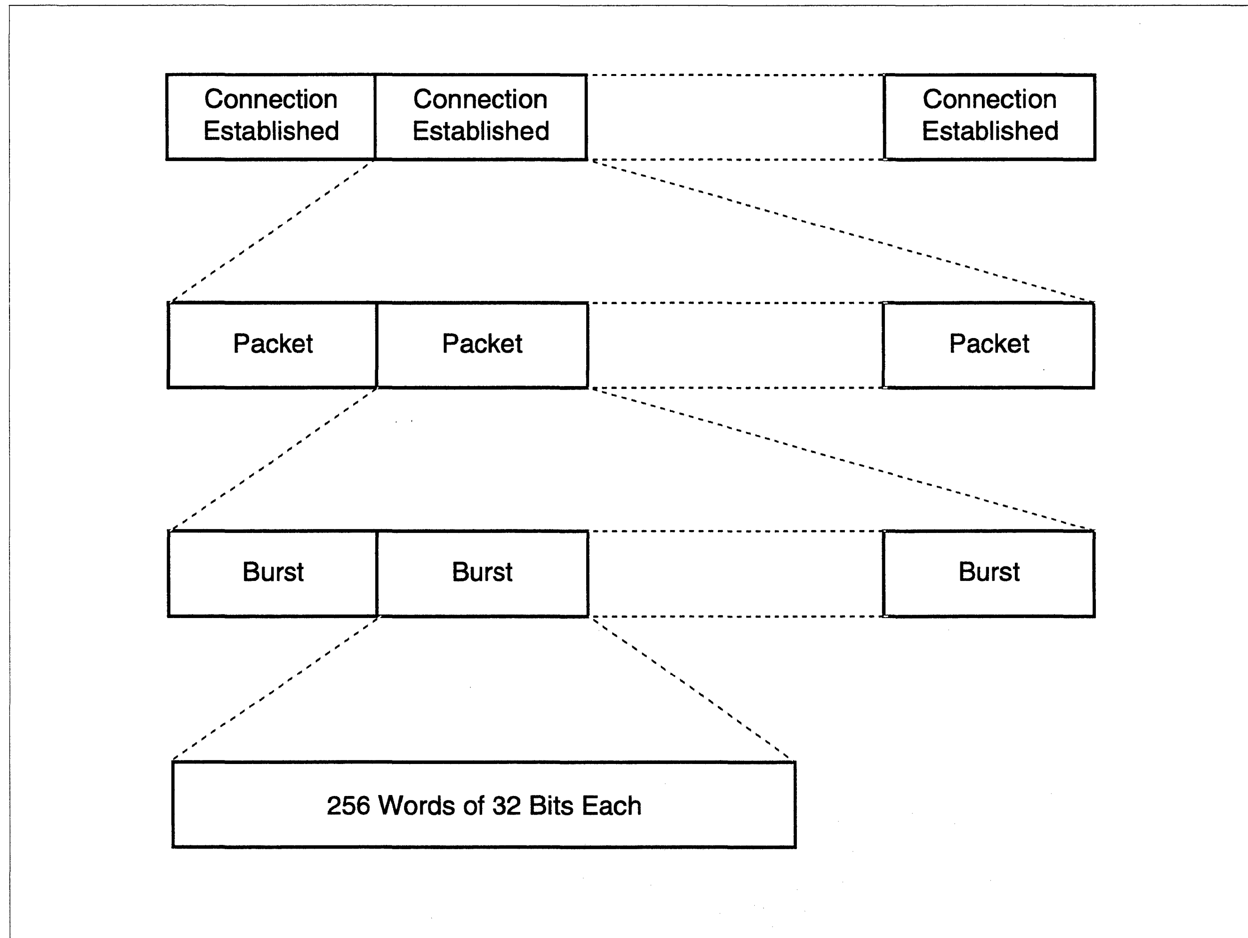

Figure 2-2. HIPPI Framing Protocol

Data Flow

This section describes the data flow in a HIPPI controller (refer to Figure 2-3 on page 2-6).

The HIPPI controller transfers data using direct memory access (DMA). DMA transfers *from* memory are called DMA reads and DMA transfers *to* memory are called DMA writes. DMA transfers occur in a maximum of 4K-byte blocks. Transfers occur when there is room for a block in the source FIFO or when there is a block available in the destination FIFO. Block transfers alternate between channels if both have requests.

Parity is written to the source FIFO during DMA transfers. The HIPPI-PH standard specifies odd parity on outgoing data, so the node bus parity is inverted before writing into the FIFO. Parity is checked by and passed through the S2020 interface circuit.

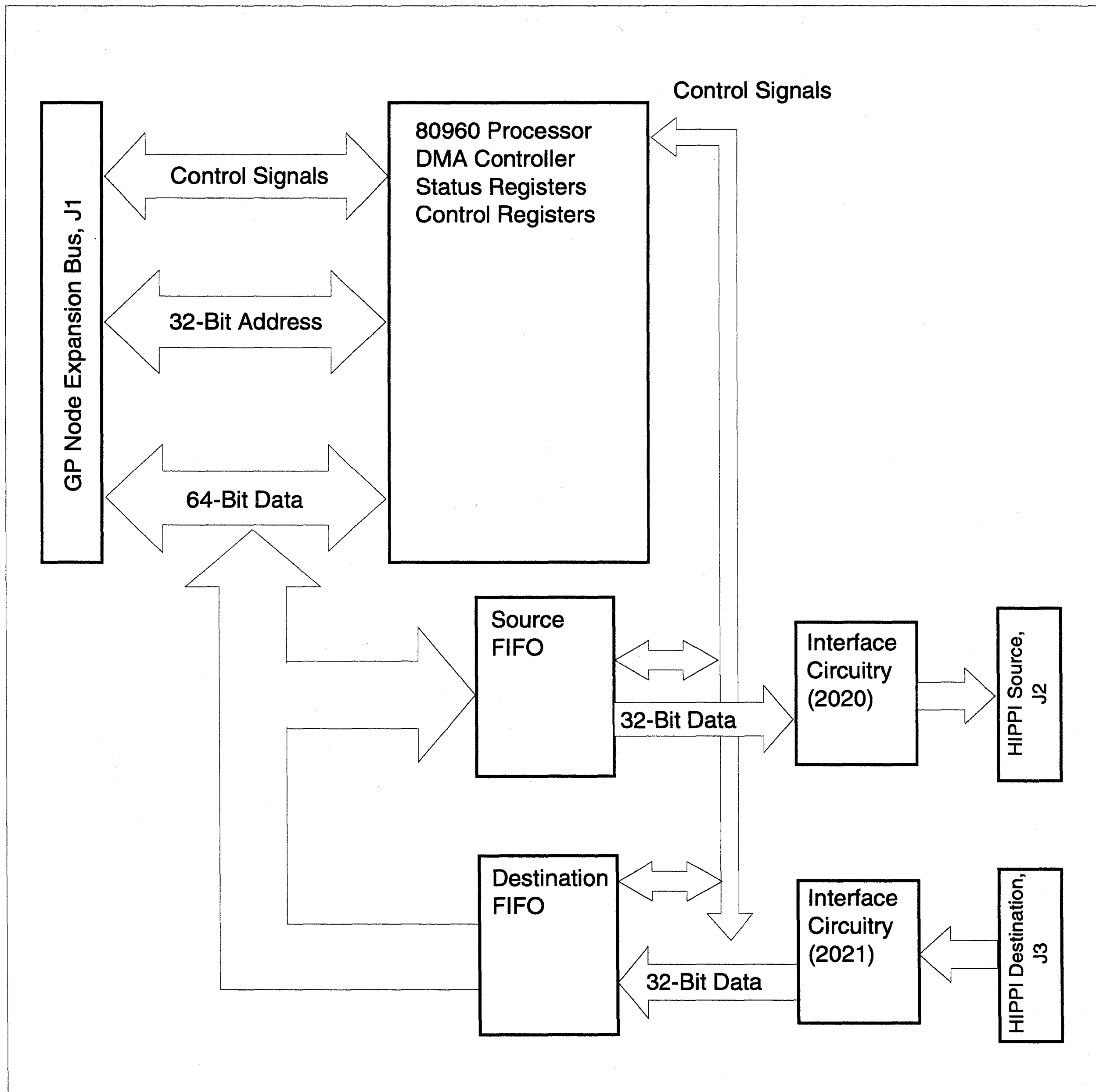

Figure 2-3. Data Flow in a HIPPI Controller

The controller communicates with node memory using DMA at full speed (zero wait states for sequential accesses). This is a peak bandwidth of 400M bytes/sec and does not include the initial transfer latency, memory refresh cycles, DRAM page boundaries, or cache-coherency cycles.

There are 512 transfers made per block (4K bytes), so with overhead, the block takes about 12 micro-seconds. Since a HIPPI burst takes 10.3 ms (258 clocks at 25 MHz), each channel uses about 29% of the node bus bandwidth with equally fast devices at the other ends.

The full 100 MB/s data rate is a theoretical maximum, measured to and from the node memory. There are several factors that decrease the data rate. First, due to inter-burst overhead, the source channel data rate is actually 98.8 MB/s, and the destination is actually 99.2 MB/s. Second, the remote device may be significantly slower than the controller, and HIPPI flow control will throttle the transfers down to the rate of the remote device. Finally, system throughput is limited by inter-node transfers, both by the mesh data rate and the node bus bandwidth required to move data into memory.

Sending a Packet to Its Destination

This section describes how data is transmitted across a simple HIPPI channel. Figure 2-4 shows which signals are active during a simple HIPPI transmission.

An important part of HIPPI transmission is the I-field. The I-field consists of a 32-bit field that contains connection routing information. The I-field precedes each HIPPI connection. Routing tables convert the network addresses to I-fields. This conversion process allows the HIPPI protocol to send and receive data between a wide variety of systems and peripherals.

The I-field is bounded by two events: the REQUEST signal being set to true and the CONNECT signal being set to true. At the highest level, a connection is bounded by both the REQUEST and the CONNECT signals being set to true. At the next highest level, the packet (which contains bursts of data) is bounded by the packet signal being set to true. This means that all bursts of data inside a packet occur between the time when the packet signal is set to true and the time when it is set to false. At the lowest level, each burst of data is bounded by the burst signal being set to true.

The I-field is described in more detail in the following section.

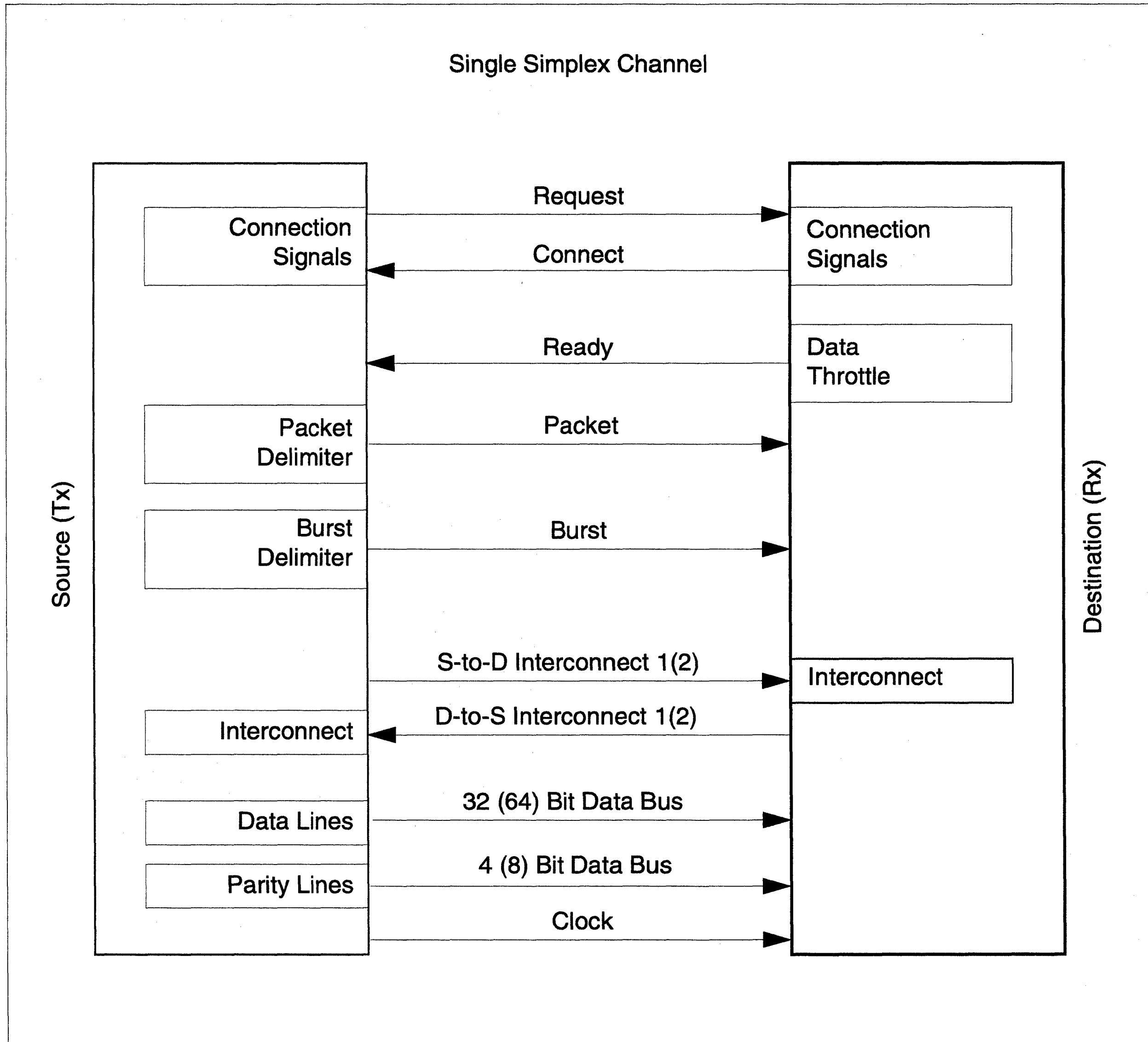

Figure 2-4. The HIPPI Signal Diagram

The I-Field

The I-field is a 32-bit field sent as part of the sequence of the physical layer operations when establishing a connection between a HIPPI source or destination. The I-field contains connection routing information. Routing tables convert network addresses to I-fields, thus allowing the HIPPI protocol to communicate with many different systems. (Refer to "Routing Tables" on page 4-6 for more information.)

Figure 2-5 on page 2-9 shows the I-field format and is followed by a description of the I-field bits. Refer to the HIPPI-SC specification for more information about the I-field.

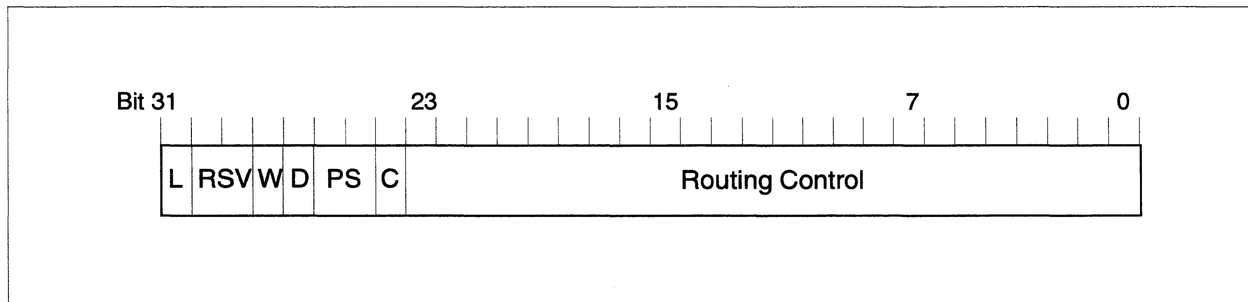

Figure 2-5. I-Field Format

L — Locally administered (bit 31). If L=0, the I-field is defined by the HIPPI-SC standard. If L=1, the entire I-field will fall under the user's local administration, and the HIPPI-SC standard will not apply.

RSV — Reserved (bits 30, 29). The reserved bits are transmitted as zeros.

W — Double-wide (bit 28). If W=0, the Source is using the 800M bits/sec date rate option. The *W* bit is used in conjunction with the INTERCONNECT signals on Cable A and Cable B. The INTERCONNECT signals tell a switch or end point that the cable is physically attached to an active HIPPI port. The *W* bit is used to tell the switch or Destination end point whether or not Cable B is being used in a particular connection.

D — Direction (bit 27). If D=0, the right-hand end of the routing control field is the current sub-field. The right-hand end contains the least-significant bits. See Figure 2-5. If D=1, the left-hand end of the routing control field is the current sub-field. The left-hand end contains the most significant bits. When a reverse path exists, a destination end point may return a reply to a received packet by using the same I-field with the *D* bit complemented.

PS — Path Selection (bits 26, 25). These bits are used as follows:

- 00 = Source routing: A specific route through the switches, with output port numbers specified for each switch.

- 01 = Destination address: Switches select the first route from a list of possible routes.
- 11 = Destination address: Switches select a route.
- 10 = Reserved

C *Camp-on* (bit 24): If C=0, the switch replies with a connection reject sequence if the switch is unable to complete the connection. If C=1, a switch will attempt to establish a connection until either the connection is completed or the source aborts the connection request.

I-Field Source Addressing

When the I-field PS bits are set to 00, the routing control field is split into multiple sub-fields. The size of the sub-fields depends on the size of the switch using it. The number of bits in the sub-field equals $\log_2 N$, where N is the switch size. A 16 x 16 HIPPI switch would use a 4-bit sub-field (Figure 2-6).

When the I-field *D* bit is set to 0, the switch uses the current sub-field (right-most bits of the routing control field) to select the switch output port. The switch right-shifts (end off) the routing control field by the number of bits in the sub-field and inserts the switch port number in the left-most bits of the routing control field (Figure 2-6).

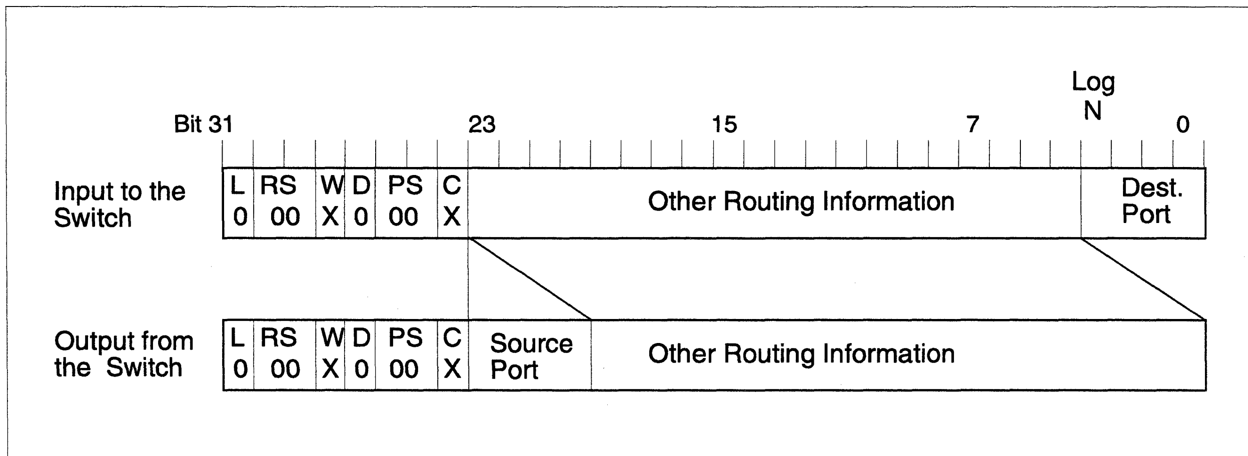

Figure 2-6. I-Field with Source Routing, D = 0

When the I-filed *D* bit is set to 1, the current sub-fields are at the left-end of the routing control field. The routing control field is shifted left, and the port number is inserted at the right end of the routing control field (Figure 2-7 on page 2-11).

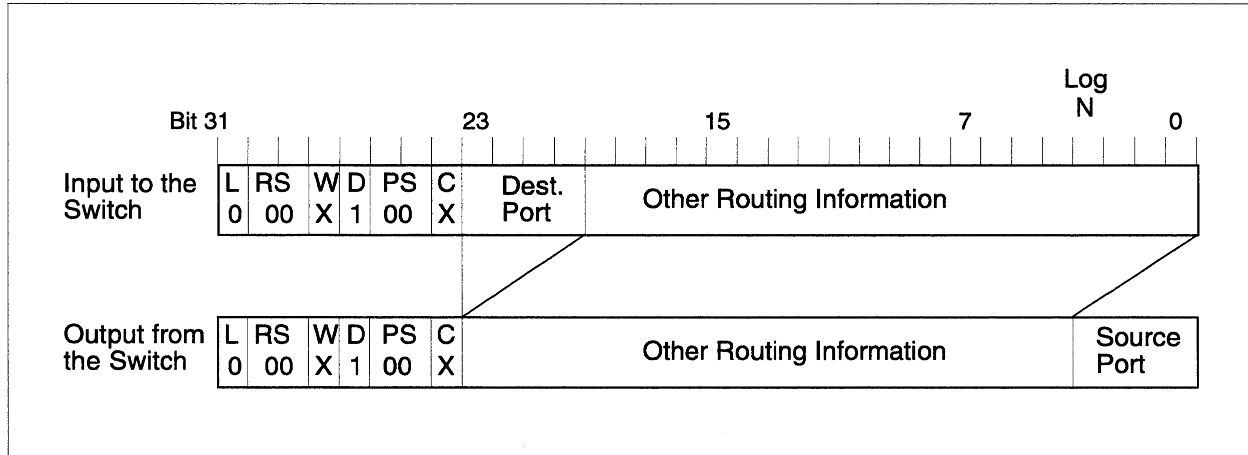

Figure 2-7. I-Field with Source Routing, D = 1

I-Field Destination Addressing

When the I-field *PS* bits are set to 01 or 11, the routing control field is split into two 12-bit fields: one 12-bit field specifies the address of the destination end point and the other specifies the address of the source end point. When the *D* bit (direction) is set to 0, the right-hand 12 bits specify the destination address, and the left-hand 12 bits specify the source address (Figure 2-8). When the *D* bit is set to 1, the right hand 12 bits specify the source address, and the left-hand 12 bits specify the destination address (Figure 2-9).

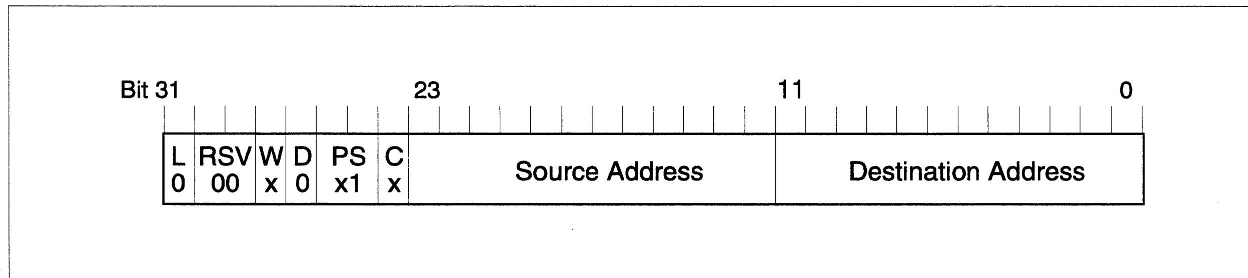

Figure 2-8. I-Field with Destination Address, D = 0

The Use of Switches In the HIPPI Environment

A HIPPI switch is a physical device that accepts input from one port and, after interrogating the I-field, passes all control and data to the receiving port. A HIPPI switch may be compared to an electrical switch in that, once the connection is made between an input port and an output port, it is as if a solid wire were connecting the two ports. The *Camp-on* bit in the I-field provides control of the switch. If the *Camp-on* bit is set, the switches are instructed to attempt a connection until the connection is completed or the source cancels the connection request.

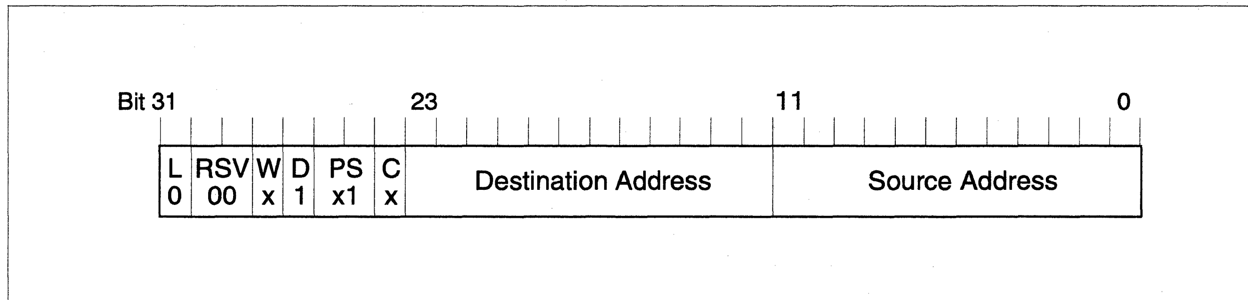

Figure 2-9. I-Field with Destination Address, D = 1

A HIPPI switch is typically identified by the number of input and output ports in the form "A x B." The "A" refers to the number of input ports and "B" to the number of output ports. Thus, an 8 x 8 switch consists of eight input ports and eight output ports

In a practical HIPPI application, a switch may provide switching capability for numerous ports. Figure 2-10 shows an 8 x 8 HIPPI switch. In Figure 2-10, these input and output ports are labeled A1 through A8 and B1 through B8. The switch in Figure 2-10 could be used to control the connection between any A port to any B port, thus allowing numerous connections.

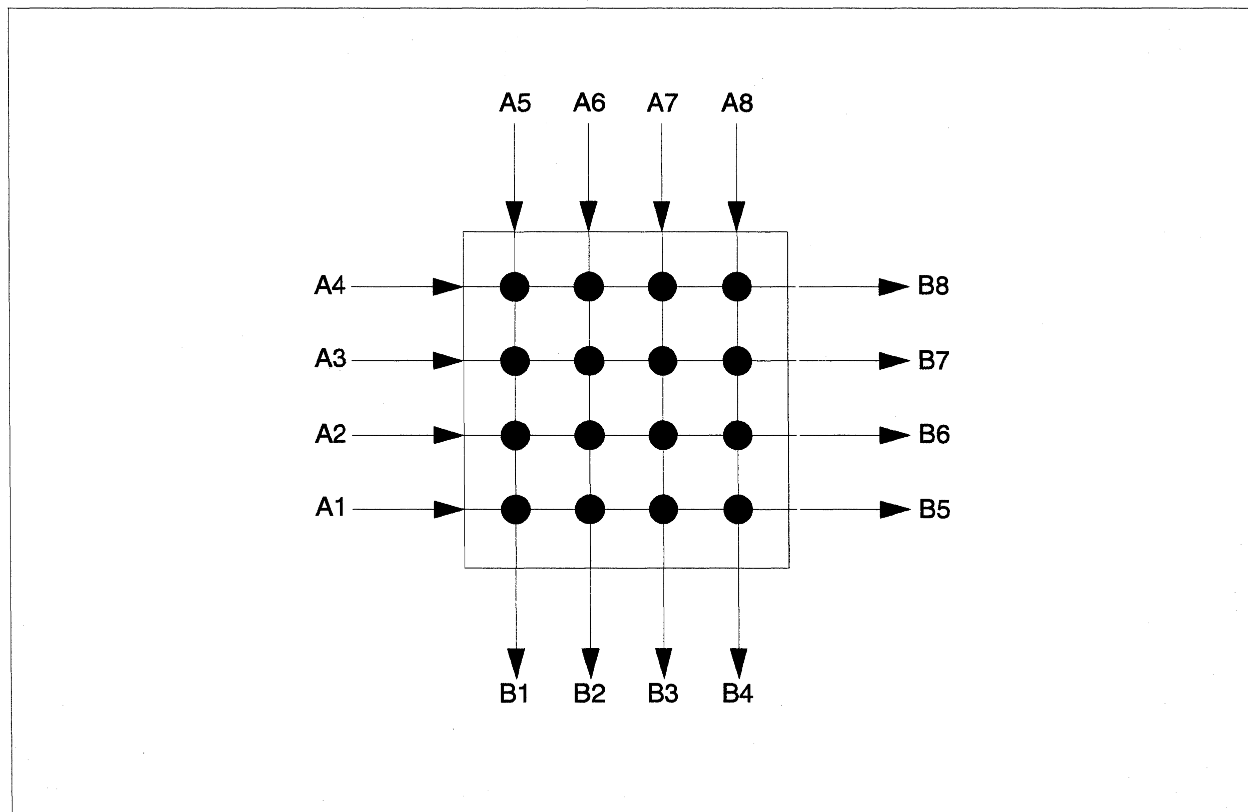

Figure 2-10. An 8 x 8 HIPPI Switch

Installing the HIPPI Controller

3

This chapter explains how to install the HIPPI controller in a Paragon™ system. Refer to the cabling guide in the *Paragon™ System Hardware Maintenance Manual* for information about how to cable the controller to the system and how to run cables out to an external system or peripheral device.

Tools Needed

You need the following tools:

- Small Phillips screwdriver.
- Small flat-bladed screwdriver.
- Antistatic wrist strap.
- Antistatic surface on which to lay the boards while working on them (the antistatic bags used to ship the boards are satisfactory).

The HIPPI Controller and the Node Board

As shown in Figure 2-1 on page 2-2, the system is shipped from the factory with the HIPPI controller already mounted on a node board. The controller is mounted on the expansion interface connector of the node board. When combined, the board and controller (called the HIPPI board) take up two slots in the cardcage.

The primary side of the HIPPI board (the side with most of the active components) faces away from the node board. The HIPPI controller mounts to six metal standoffs on the node with 2-52 screws, #2 washers, and lockwashers. The standoffs provide proper clearance for the expansion connector (J1).

The channel connectors on the HIPPI board are accessible through the HIPPI board's front panel. On the front panel, the channel connectors are labeled "SRC" for the source connector (J2) and "DST" for the destination connector (J3). The board contains a pair of LED's for each channel, one red and one green. Each pair is mounted at the upper end of its associated connector.

Installing the HIPPI Controller

CAUTION

You must shut the system down before installing the controller. Before shutting the system down, make sure that no applications are running on the system. For more information about how to shut the system down, refer to the *Paragon™ System Administrator's Guide*.

Follow these steps to install the controller:

1. Turn off the system's power.

2. Once power is off, open the cabinet door (Figure 3-1 on page 3-3). The door latch is located at the bottom right of the front door. Unlock the door by moving the latch 90° counterclockwise.

3. Attach the antistatic strap to your wrist and connect the other end of the strap to a solid ground.

NOTE

The HIPPI controller/node assembly requires two cardcage slots. The following steps tell how to remove node boards from the cardcage. If you already have two adjacent empty slots in which to install the HIPPI controller, skip steps 4 through 8.

4. Loosen the capture screws at the top and bottom of the node's front panel (Figure 3-2 on page 3-4).

5. Remove the node board by grasping the nylon clips on the top and bottom of the board, and pull the clips toward you. This should dislodge the board.

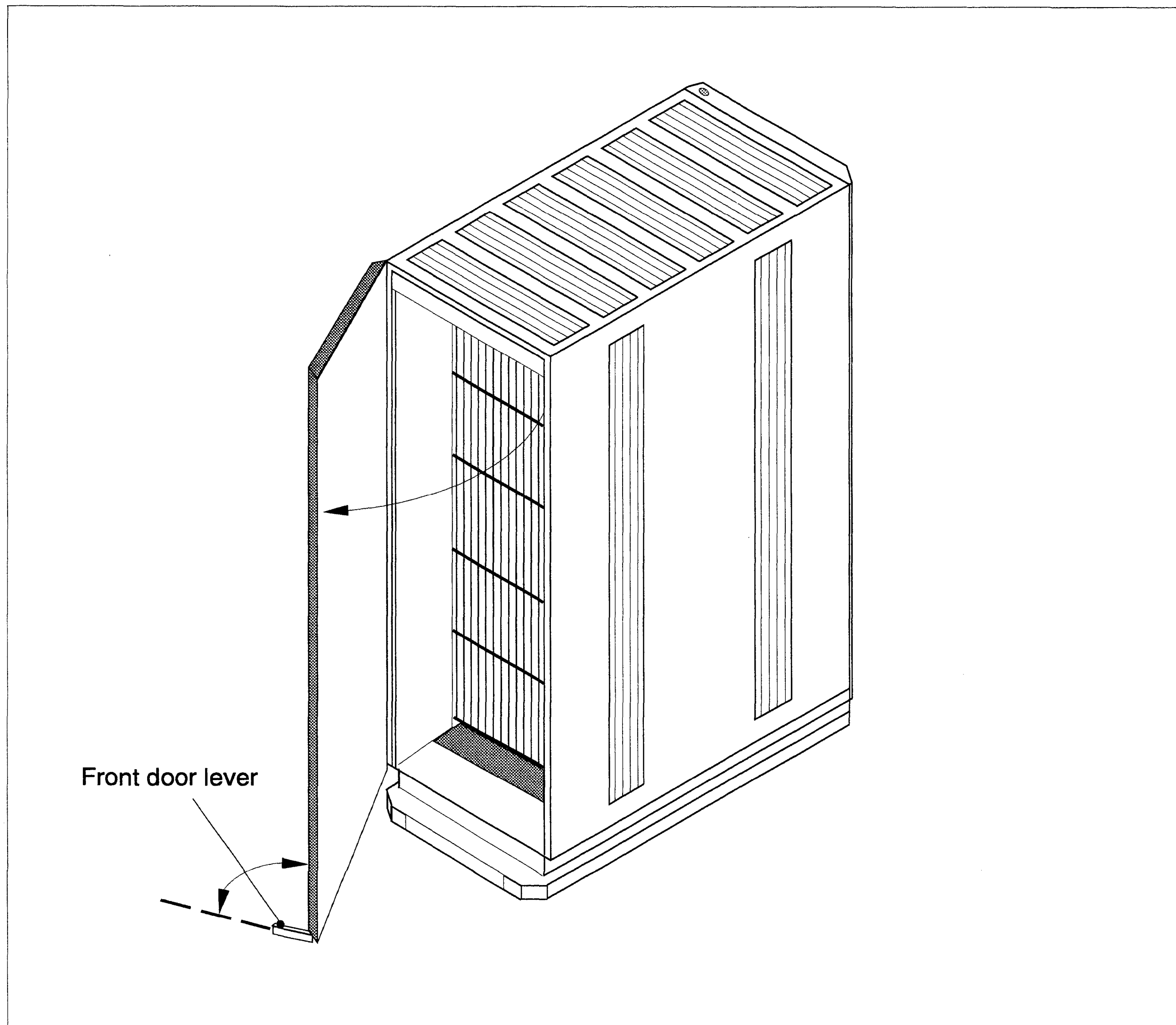

Figure 3-1. Opening the Paragon™ System Cabinet Door

6. Carefully slide the board out of the cardcage. Take care to handle the board only by its edges.

7. Put the node board in an antistatic bag and lay it aside.

8. Repeat steps 4 through 7 for the second node.

9. Remove the combination HIPPI controller/ node from its antistatic bag. Handle the board only by its edges.

10. Align the board edges with the cardguide rails. Slide the board into the cardcage until the board connectors meet the backplane (Figure 3-3 on page 3-5).

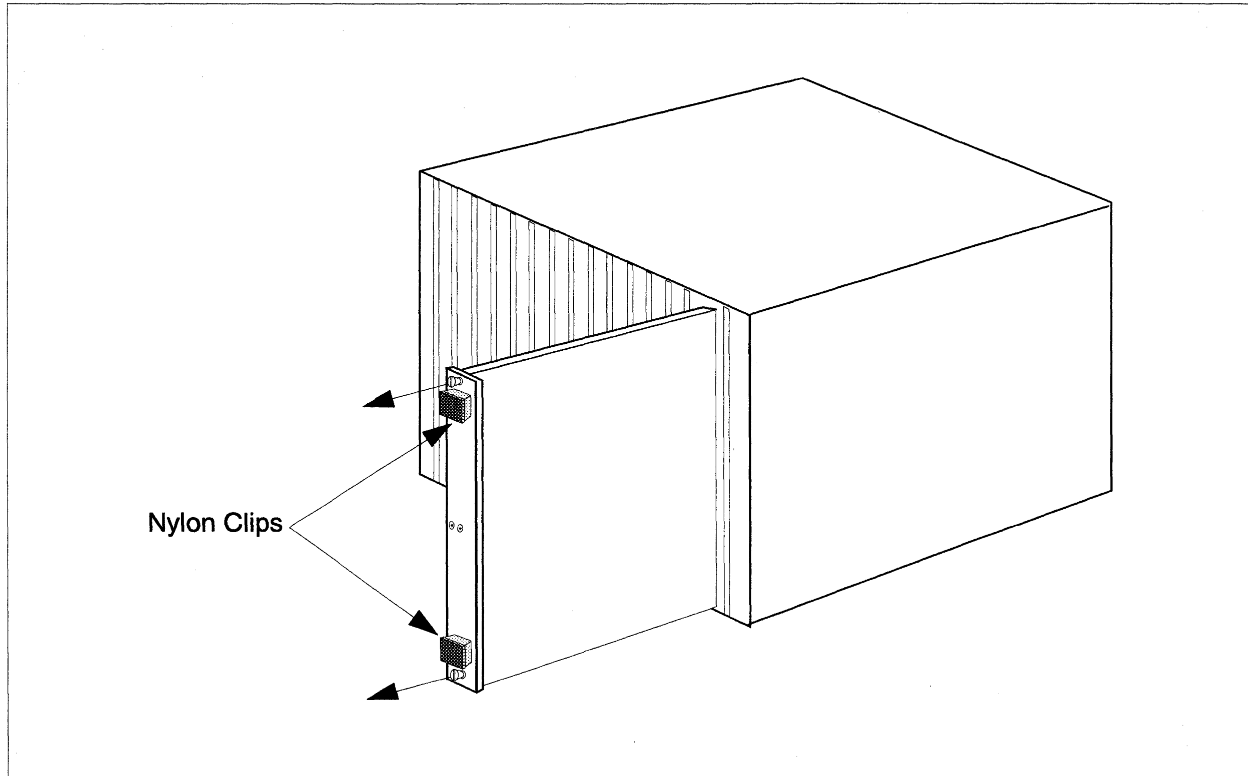

Figure 3-2. Removing a Node Board from the Cardcage

11. Align the node board connector with the backplane and press firmly on the board until the board connector mates with the backplane.

12. When the board is properly seated, the nylon clips straighten out and the board front panel is flush with the frame of the cabinet.

13. Tighten the capture screws at the top and bottom of the board front panel.

Installing the Cables

Each HIPPI controller comes with two identical 6.5-foot, 100-pin "internal" cables that connect to the controller. One cable attaches to the Source Channel Connector and the other attaches to the Destination Channel Connector. Each HIPPI controller also comes with two 25-meter "external" cables that are used to connect to an external system, switch, or peripheral device. The internal and external cable connectors are linked together either at an I/O panel (in early systems) or outside of the cabinet (in later systems). Refer to the *Paragon™ System Hardware Maintenance Manual* for detailed HIPPI cabling procedures.

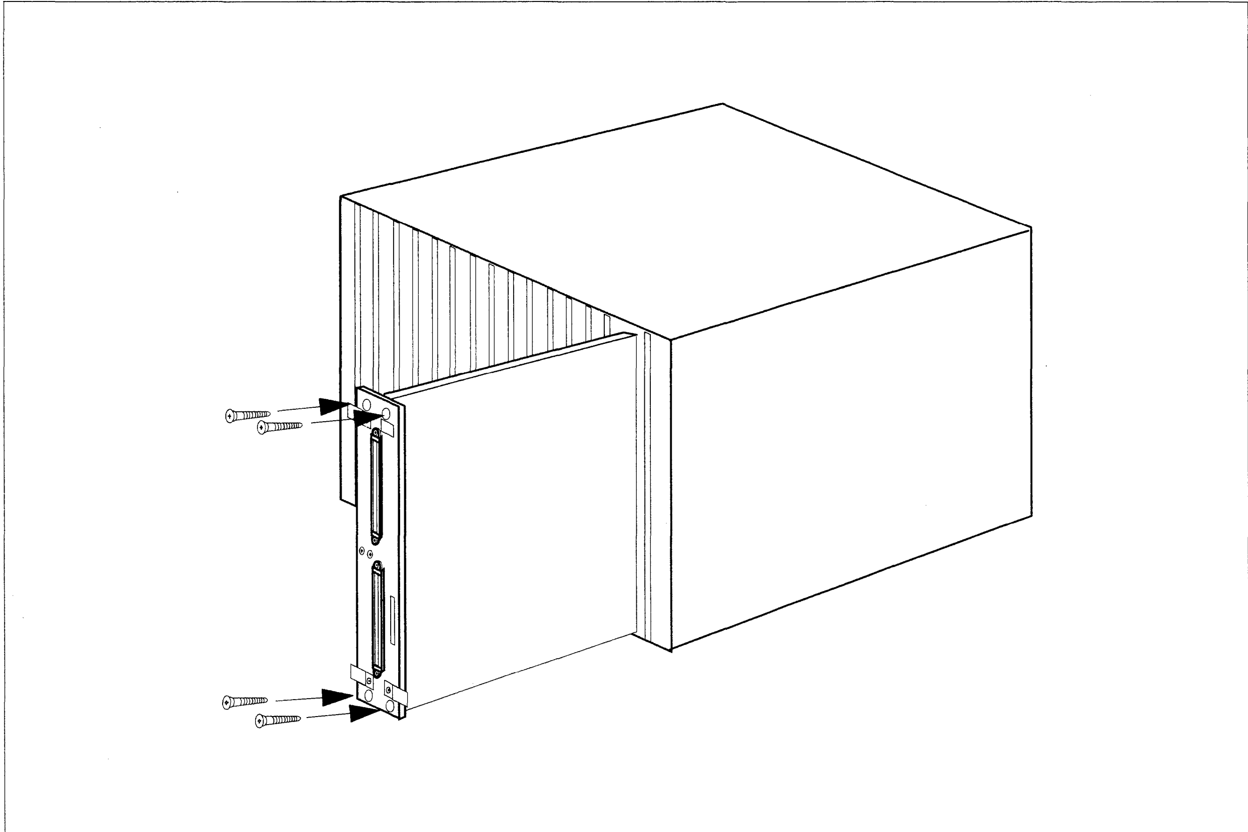

Figure 3-3. Installing the Combined HIPPI/Node Board in the Cardcage

For each system, there is a single loopback cable that is used to perform loopback diagnostic tests. This cable connects to the ends of the source and destination channel connectors on the HIPPI controller to allow you to verify loopback transfers. Refer to Chapter 5, "HIPPI Diagnostics" for more information about using the loopback diagnostic cable.

Intel supplies all these cables for your system. Refer to Chapter 6, "Cable Parts and Specifications," for more information about the cables.

Closing the Cabinet Door

Close and latch the front door to the Paragon system cabinet. Lock the door by moving the latch 90° clockwise. See Figure 3-1 on page 3-3.

Software Configuration 4

This chapter discusses the following software configuration issues:

- Configuring the system for the HIPPI board.
- Configuring a network interface.
- Activating the interface.
- Defining and installing routing tables.
- Building data packets.
- Raw HIPPI.

Network Configuration

The device driver for the HIPPI controller serves as a bridge between the Mach microkernel and the HIPPI controller card. The device driver is integral to the microkernel, but you must configure the network interface for your own system.

Configuring The System For HIPPI

1. On the Diagnostic Station, update the */usr/paragon/boot/DEVCONF.TXT* file to tell the system where the HIPPI board is located. The entry in *DEVCONF.TXT* should be similar to this example, which specifies a HIPPI board in Cabinet 0, Backplane A, Slot 12:

```
HIPPI    00A12    H04
```

   The "H04" is a generic version/rev ID used with HIPPI boards. Refer to the *Paragon™ System Commands Reference Manual* for more information about the *DEVCONF.TXT* file.

2. Once the *DEVCONF.TXT* file is current, reset the system to create the *SYSCONFIG.TXT* and *SYSCONFIG.BIN* files:

```
reset autocfg
```

Configuring the Network Interface

To configure the network interface, you must know the following values:

I-field settings	The I-field is a 32-bit control field defined by the HIPPI-SC specification. For more information, refer to "The I-Field" on page 2-9.
IP address	Each HIPPI board in a system must be assigned its own unique IP address. Your system administrator can provide the IP address.
ULA address	The ULA (Universal LAN Address) for each HIPPI board is unique and is written into the controller's flash memory when the controller is manufactured. Your system administrator can provide the ULA (the ULA number is printed on each HIPPI board). The ULA has the following form: `1:0c:34:65:0:26`.

The following steps show how to configure a HIPPI interface that starts automatically when the Paragon system is rebooted.

1. Log in to the Paragon system as *root.*

2. Back up the files */etc/hosts* and */sbin/init.d/inet.*

3. Edit the */etc/hosts* file to add network address(es) of the HIPPI board(s). The following line is an example of a network address specification for a HIPPI board in */etc/hosts*:

```
123.45.678.910    si123 HIPPI_PARA
```

NOTE

It is recommended to place the HIPPI board(s) on a different subnet than the Ethernet interfaces in the Paragon system to avoid network addressing conflicts.

4. Create the */etc/hippi.map* file, using the format in the following example:

```
# Lines with commments begin with #
# Inet Address            ULA          HIPPI I-field Address
123.45.678.910      00:BA:00:00:00:11       0x000002
123.45.678.911      00:AA:00:06:2C:2B       0x000003
```

- The first column contains the Internet address(es) of the HIPPI board(s).
- The second column contains each board's ULA address—found by looking on the HIPPI board. The ULA may also be found by looking at the HIPPI node's boot output with the Paragon System Debugger.
- The third column contains the HIPPI I-field addresses, which indicate which port of the HIPPI switch that the board is connected to.

NOTE

The I-field addresses must be unique, even if a HIPPI switch is not used.

5. Configure the HIPPI interface as described below under Activating The Network Interface.
6. Reboot the Paragon system to enable the HIPPI interface(s) and to verify that they automatically start at reboot.

Activating the Network Interface

The **ifconfig** command assigns an Internet address and activates the interface. It may be used manually at the operating system prompt, or may be added to the */sbin/init.d/inet* file so that the interface is configured each time the Paragon system is reset.

The following example command assigns and activates a HIPPI interface:

```
# /sbin/ifconfig \<hippinode\> ifhip0 192.9.2.5 netmask
255.255.255.0 up
```

- *hippinode* is the number of the node on which your HIPPI controller is installed (using the root-node numbering scheme). (Refer to the *Paragon™ System Diagnostic Reference Manual* for information about the node-numbering schemes.)
- *192.9.2* is the network number in the example of a class C network address and *5* is the host number. Note that an entry for this IP address and a unique hostname must be in the */etc/hosts* file.

NOTE

All HIPPI devices must be configured after configuring the boot node and prior to configuring the standard network loopback device *lo0*.

It is recommended that the **ifconfig** commands be added to the */sbin/init.d/inet* file, so that the HIPPI interfaces are configured automatically when the Paragon system is reset. The following example shows the lines that need to be added:

```
# THIS COMMAND CONFIGURES THE HIPPI DEVICE
/sbin/ifconfig "<12>ifhip0" 123.45.678.910 netmask 255.255.255.0
-trailers
# Put this *before* configuring lo0

# THIS COMMAND LOADS THE HIPPI NETWORK ROUTING TABLE INTO THE #
HIPPI DRIVER
/usr/sbin/hippi_setmap ifhip0 /etc/hippi.map

# THIS COMMAND SHUTS DOWN THE HIPPI INTERFACE
# Insert this command at the end of the inet file
# as one of the steps to stopping (disabling)
# the network.
/sbin/ifconfig "<12>ifhip0" down
```

Optimizing TCP/IP Performance Over the HIPPI Channel

To maximize the performance of TCP/IP over the HIPPI channel, add the following bootmagic variable definition to the *MAGIC.MASTER* file:

```
TCP_SPACE_SIZE=262144
```

This allocates larger socket buffers for the program and enables the Paragon system to use window scaling with other systems that use it. Reboot the Paragon system to initiate the changes made to the *MAGIC.MASTER* file.

NOTE

In programs, I/O operations done over the sockets should be at least 64K, to maximize Paragon system HIPPI TCP/IP performance.

Determining ULA Addresses for HIPPI Boards

Each HIPPI board is assigned a unique Universal LAN Address (ULA) when it is made. The ULA is printed on the HIPPI board. You can also determine a HIPPI board's ULA with the following procedure:

1. From the Paragon system console, connect to the node board where the HIPPI daughtercard is attached:

```
# ~#
Valid Nodes - node list
New Node:
```

   When prompted for a new node, enter the number of the node where the HIPPI daughtercard is attached.

2. Examine the resulting output (about 30 lines) and locate the line that looks similar to the following example:

```
Hippi 10:0 ULA (MAC) - 00:aa:00:65:0:26, MTU 65535
```

   In this example, the ULA is 00:aa:00:65:0:26.

3. Reconnect to the boot node,

```
# ~#
Valid Nodes - node list
New Node:
```

   When prompted for a new node, enter the number of the boot node.

Routing Tables

HIPPI routing tables provide the server with a mapping from the Internet address and the ULA address to the I-fields. Each table includes the IP address, the ULA, and the I-field. A routing table for all Paragon system HIPPI interfaces is maintained in a file that this manual refers to as a *hippi.map*.

Routing Tables for Simple Networks

The following example shows a *hippi.map* file for the simple, one-switch network shown in Figure 4-1 on page 4-7.

```
# HIPPI Network Routing Map for switch through the HIPPI
# Switches, using Source Routing protocol
#
#      L=0     Conforms to HIPPI-SC specification
#      Rs=00  Reserved
#      W=0     800 Mbit/sec, 32-bits wide
#      D=0     Destination Port in LSB
#      Ps=00   Source Routing protocol
#      C=1     Camp on line until connected or src aborts
#
# Bit 3       2       1       0       0
#     1       3       5       7       0
#    +---------------------------------+
#    |LRsWDPsC<--- Other Info ---->Port|
#    +---------------------------------+
#          I-field Bit Values
#
# Port switches:
#      Jaguar    =0
#      Cheetah   =1
#      Panther   =2
#      Cougar    =3
#
# The I-field for Cougar is:
#
# Bit 3       2       1       0       0
#     1       3       5       7       0
#     00000001000000000000000000000011 = 0x1000003
#            ^                       ^^
#            |                       ||
#         Camp-on bit             Switch 3
#
# IP Address       IEEE              I_field      Comments
#-----------      --------------     ---------   ---------
```

```
192.9.3.1     1:0c:34:65:0:26     0x1000000     #Jaguar
192.9.3.2     1:0c:34:65:0:27     0x1000002     #Panther
192.9.3.3     1:0c:34:65:0:28     0x1000001     #Cheetah
192.9.3.4     1:0c:34:65:0:29     0x1000003     #Cougar
```

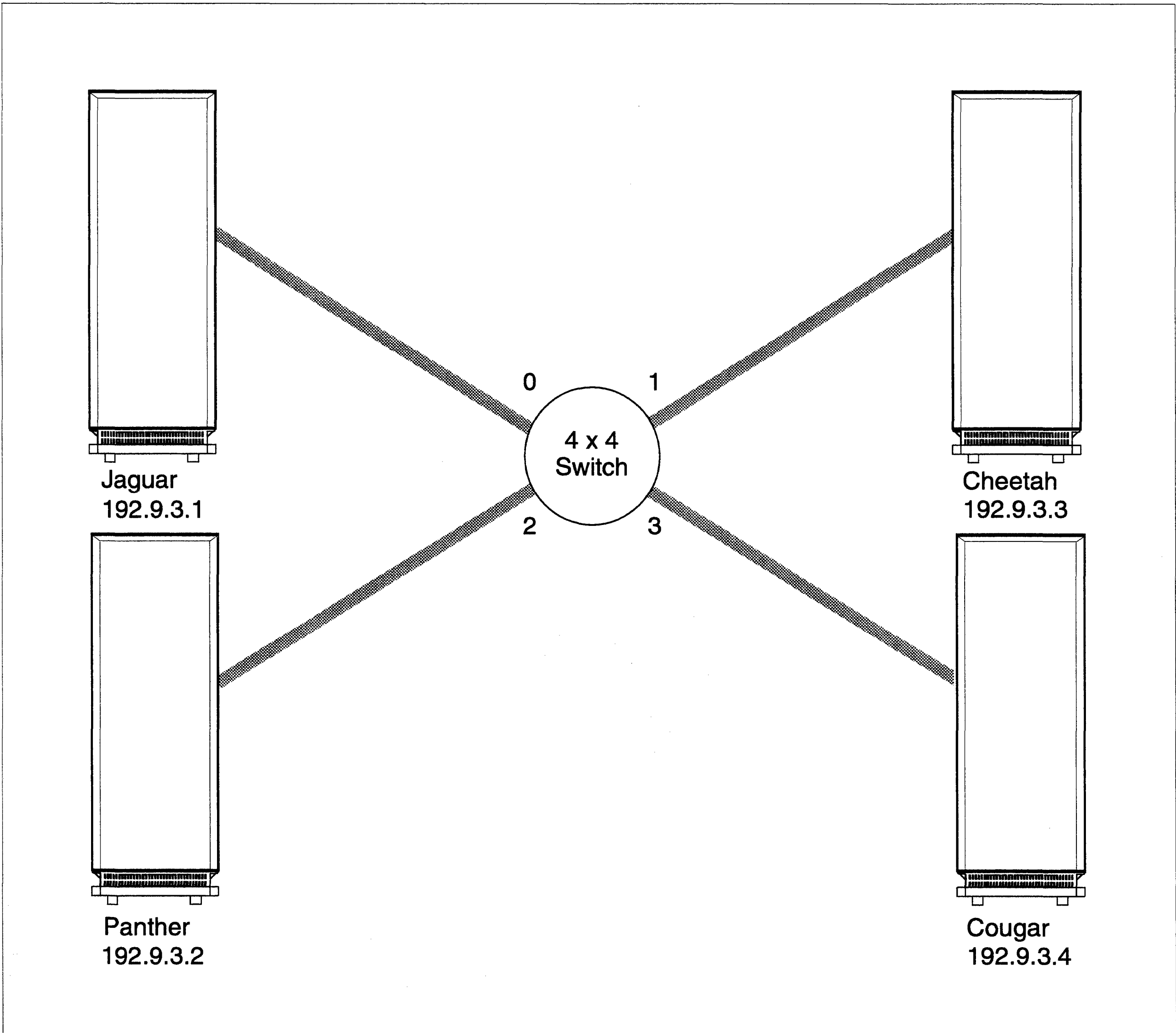

Figure 4-1. Sample Network with One HIPPI Switch

Routing Tables for Complex Networks

In the simple network shown in Figure 4-1, a single routing table supports all hosts on the network, and all hosts go through the same switch to reach their destination. In a more complex network with multiple switches, there may be multiple routing tables, and maintaining them becomes a more

complex job. Each host's routing table must define the interconnecting paths through the switching fabric. If there are failures with the switches, you must modify the routing tables to accommodate a new path.

Routing Table Commands

There are two routing table commands: **hippi_showmap** and **hippi_setmap**.

Use the **hippi_showmap** command to display the current routing table. For example:

```
# hippi_showmap
```

Use the **hippi_setmap** command to load a routing table (formatted like *hippi.map* shown previously) into the device driver. For example:

```
# hippi_setmap ifhip0 hippi.map
```

NOTE

Every time the **hippi_setmap** command is executed, it appends the table to the one in the driver. Use the **hippi_setmap -d** command line switch to delete the table in the driver first.

Refer to Appendix A for more information about the **hippi_setmap** and **hippi_showmap** commands.

Server Interface and Packet Building

The system server supports TCP/IP traffic over HIPPI as well as raw HIPPI frames via the raw HIPPI library, *libhippi.a*. Use of the HIPPI interface for TCP/IP is transparent to the user. As with other interfaces the TCP/IP protocol engine routes data over the HIPPI interface when the network address of the destination dictates.

The server formats each packet of information as specified in Figure 4-2 on page 4-9. Each packet consists of three areas: the FP_Header_Area, the D1_Area, and the D2_Area. The FP_Header_Area contains the Upper Level Protocol (ULP) identification which designates the destination ULP. This identifies where the packet of information is to be delivered. Both D1_Area and D2_Area are data areas. These packet areas are described in the following sections.

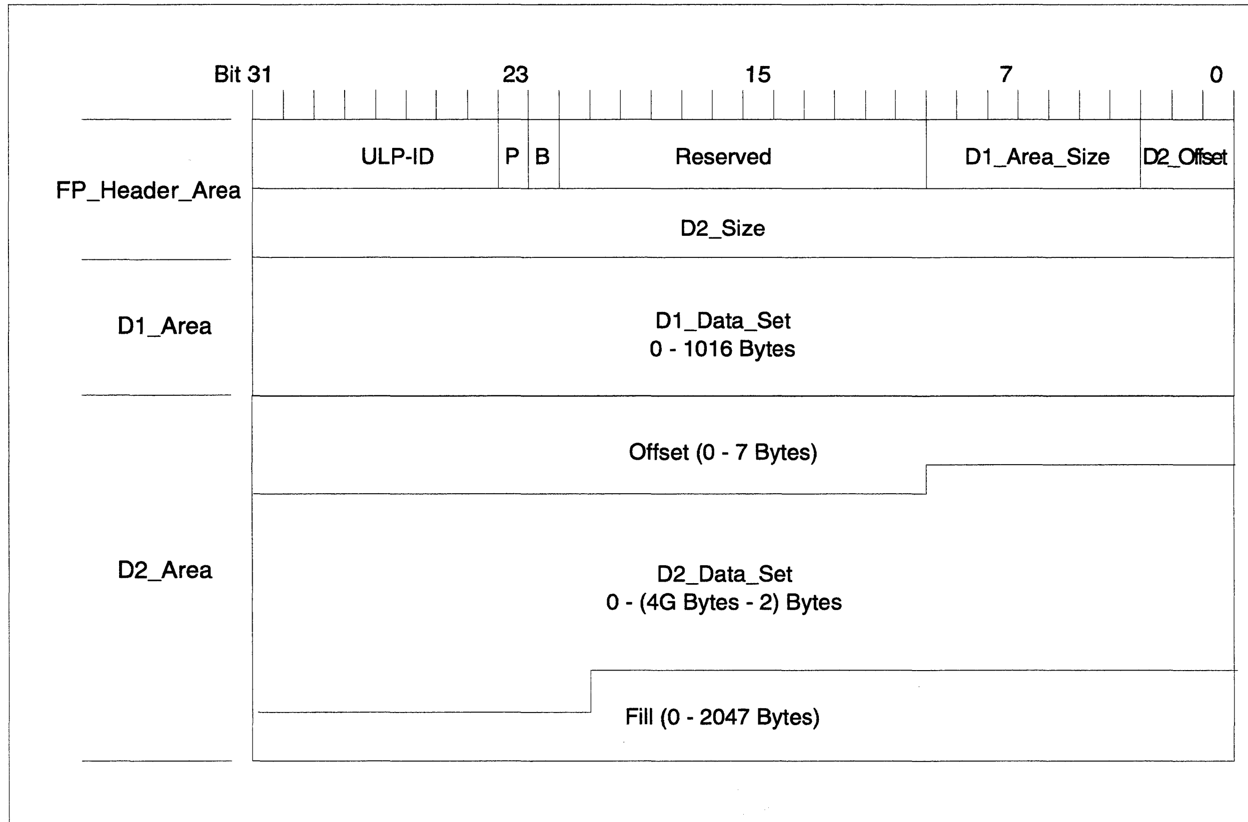

Figure 4-2. HIPPI Packet Format

The FP_Header_Area

The FP_Header_Area comprises the ULP-ID, which is eight bits in length. A value of one in the P bit indicates that a D1_Data_Set is present in this packet. A value of zero in the P bit indicates there is no D1_Data_Set present in this packet. A value of zero in the B bit indicates the D2_Area starts at or before the beginning of the second burst of the data packet. A value of one in the B bit indicates the D2_Area starts at the beginning of the second HIPPI-PH burst of the data packet. The D1_Area_Size designates the number of 64-bit words between the end of the 64-bit Header Area and the start of the D2_Area. The D2_Offset field contains the number of bytes in the Offset field in the D2_Area (the number of "junk" bytes at the beginning of the D2_Area). The D2_Size field contains the number of bytes in the D2_Area (including the Offset field); this number is aligned on an 8-byte boundary.

The D1_Area

The D1_Area follows the FP_Header_Area. If the P bit of the Header Area is equal to zero, then the D1_Data_Set is not present and the contents of the D1_Area are ignored. The D1_Data_Set is the first information in the D1_Area. This area contains control information that may be delivered to the Destination Upper Level Protocol on receipt, without waiting for the arrival of other bursts of the packet. The maximum size of the D1_Data_Set is 127 64-bit words.

The LE_Header

The Link Encapsulation (LE) protocol specification is the currently supported protocol for detailing the header information passed to the D1_Area. The LE protocol envelopes a 802.2 LLC packet for transmission over HIPPI Framing Protocol. The Destination Hub Address, Destination Port, Source Hub Address and Source Port fields in the LE_Header identify the physical address of the hub which this packet has started from or is destined to. Figure 4-3 shows the LE Packet Format.

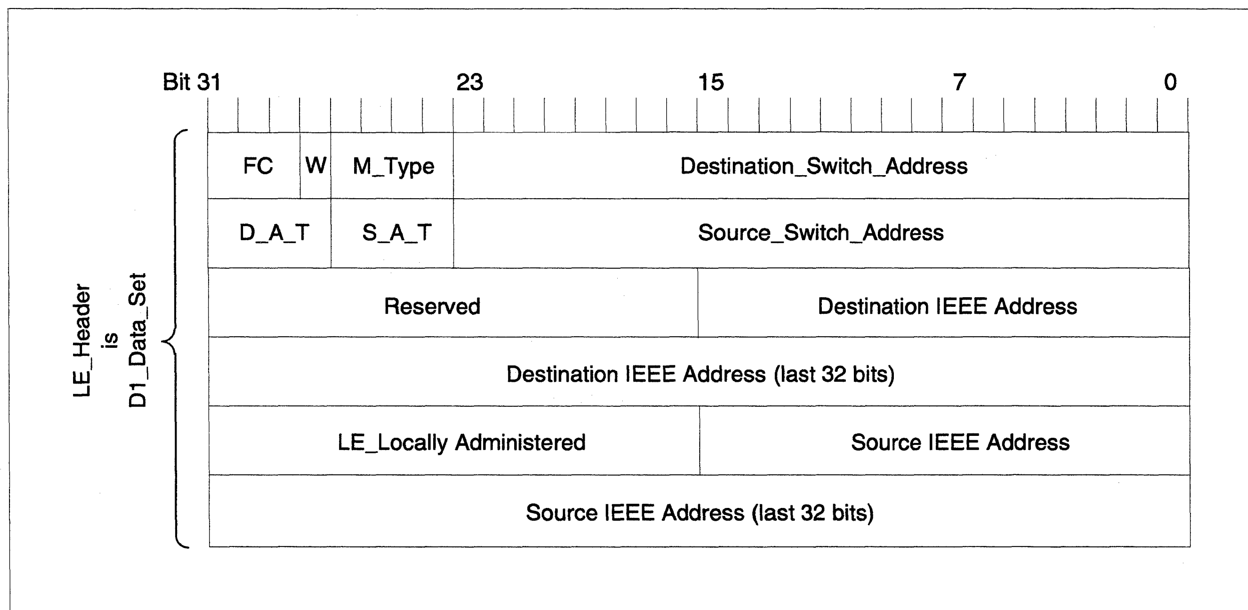

Figure 4-3. Link Encapsulation Packet Format Header

Figure 4-4 on page 4-12 depicts a HIPPI packet. The figure shows the Framing Protocol and the Link Encapsulation Packet Format together. The LE_Header is placed in the D1_Area of the HIPPI FP Packet.

The D2_Area

If the D2_Size is not zero in the Header Area, D2_Area will then immediately follow the D1_Area and will start on a 64-bit boundary and will contain the D2_Data_Set. If the B bit in the Header Area equals one, then the D2_Area will start at the beginning of the second HIPPI burst. The Offset is the unused bytes from the start of the D2_Area to the first byte of the D2_Data_Set.

The D2_Data_Set can range in size from zero to an indeterminate number of bytes. The Fill part of the D2_Area is the unused bytes between the end of the D2_Data_Set and the end of the D2_Area, for example, the end of the packet. If a D2_Size of all binary ones is used, then there is no Fill in the D2_Area.

Inbound Packets

When an incoming HIPPI packet is received, the device driver sends it to the destination channel using standard microkernel network code. The HIPPI packets are filtered using the raw HIPPI Upper Layer Protocol (ULP) and (optionally) a port number that must appear in the first word of D1. If the filter reads an LE_Header in the packet, then that packet is sent to the TCP-IP server. See Figure 4-5.

Raw HIPPI

The raw HIPPI interface is implemented as a user library (*libhippi.a*) that sits on top of the Mach device interface. The library consists of routines that allow you to define HIPPI packet formats and to open, close, write, and read HIPPI channels. To use *libhippi.a*, your code must contain the include file *raw_hippi.h*. For detailed information about the *libhippi.a* functions, refer to the manual pages in Appendix A.

Raw HIPPI Usage Models

The raw HIPPI interface supports two usage models, called HIPPI_RAW and HIPPI_DATA. You select a model when you call the **hippi_open** function (see Appendix A). Whichever model you use, the library maintains state information to ensure that each open HIPPI connection functions correctly.

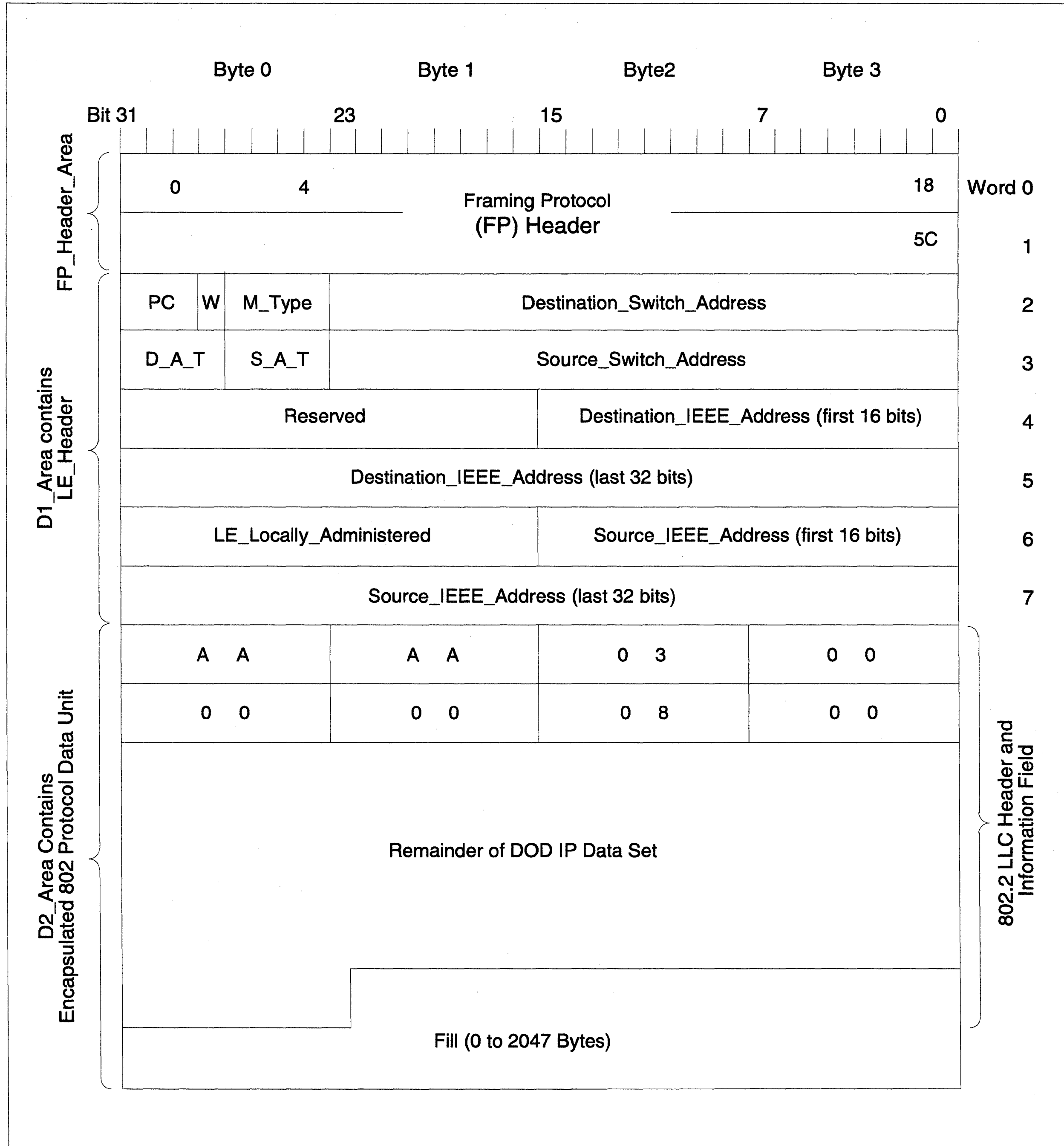

Figure 4-4. A HIPPI Framing Protocol Packet with an LE_Header

The HIPPI_RAW model assumes you are a sophisticated user of HIPPI and therefore *does not* provide HIPPI frame formatting. Instead, you are responsible for formatting the HIPPI frame, allocating and deallocating memory using the functions in *libhippi.a*, and rounding word sizes (per

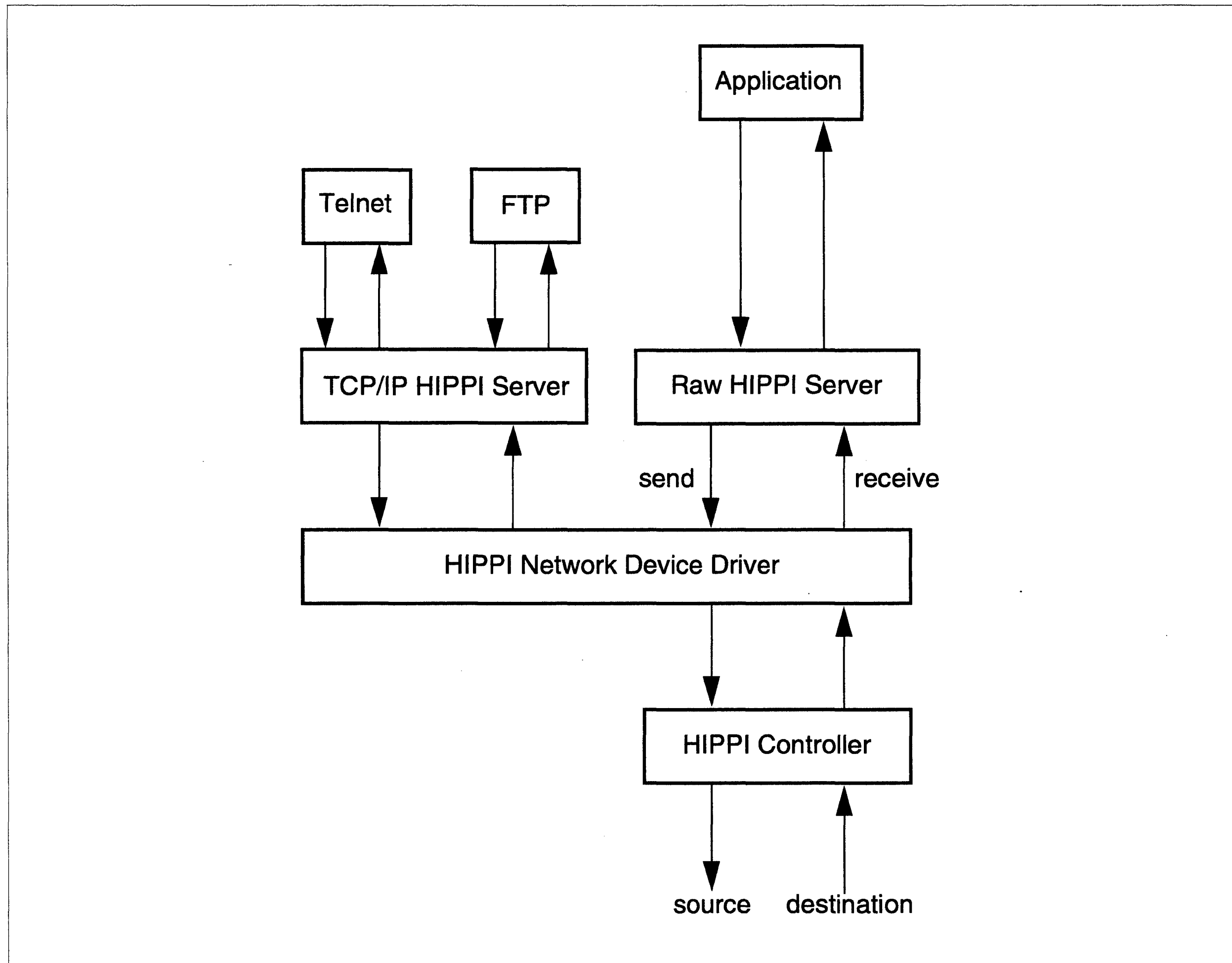

Figure 4-5. HIPPI Packet Incoming and Outgoing Flow

the HIPPI-FP specification).

The HIPPI_DATA model assumes you are not a experienced user of HIPPI and therefore *does* provide HIPPI frame formatting. You are also responsible in this model (as in HIPPI_RAW), for allocating and deallocating memory using the *libhippi.a* functions.When you use these memory functions in the HIPPI_DATA model, memory is allocated in such a way that the library can add HIPPI headers to frames without copying data and the driver is able to process frames efficiently.

Using Raw HIPPI

Before you can use raw HIPPI, your system administrator must use the **rmknod** command (the remote version of **mknod**) to make a special file for each HIPPI board on your system. Normally this is done just after system installation and need not be done again (unless a HIPPI board is changed). As an example, the following command creates a HIPPI device (called hippi.one) having major device number 24, minor device number 0, and node number 5:

```
# rmknod /dev/hippi.one c 24 0 5
```

NOTE

rmknod creates the special file with the permissions set according to your *umask*. You may want to use **chmod** to change to permissions required for your system.

Refer to Appendix C for usage notes and examples of using the raw HIPPI interface.

HIPPI Diagnostics 5

This chapter introduces the HIPPI controller diagnostic tests and explains how to install the diagnostic cable used with the loopback tests.

Diagnostics

The HIPPI controller diagnostic tests consist of a power-on self-test, and tests that are part of the Paragon™ System Diagnostic package. Portions of the power-on test process require a loopback cable to be installed on the HIPPI controller.

Power-On Self Test

The power-on test is an extension of the Paragon System Node Confidence Test (NCT). If the power-on test passes, the system assumes that the HIPPI controller functions correctly. This test is not intended to provide detailed fault isolation beyond the Field Replaceable Unit (FRU) level. The HIPPI power-on test returns a pass/fail indication to the main node-confidence test.

Paragon™ System Diagnostic Program

The Paragon System Diagnostic (PSD) program includes a comprehensive set of tests for evaluating the Paragon system HIPPI interfaces, and to isolate faults to the Field Replaceable Unit (FRU) level.

For more detailed information about the Paragon System Diagnostics, see the *Paragon™ System Diagnostic Reference Manual* and the *Paragon™ System Diagnostic Troubleshooting Guide.*

Loopback Tests

Some parts of the power-on test require a loopback cable. If the loopback cable isn't installed, these tests don't execute, but return "PASS". If you install a loopback cable, the tests transfer data through the cable.

The loopback cable connects the HIPPI controller's source and destination channel connectors together (Figure 5-1).

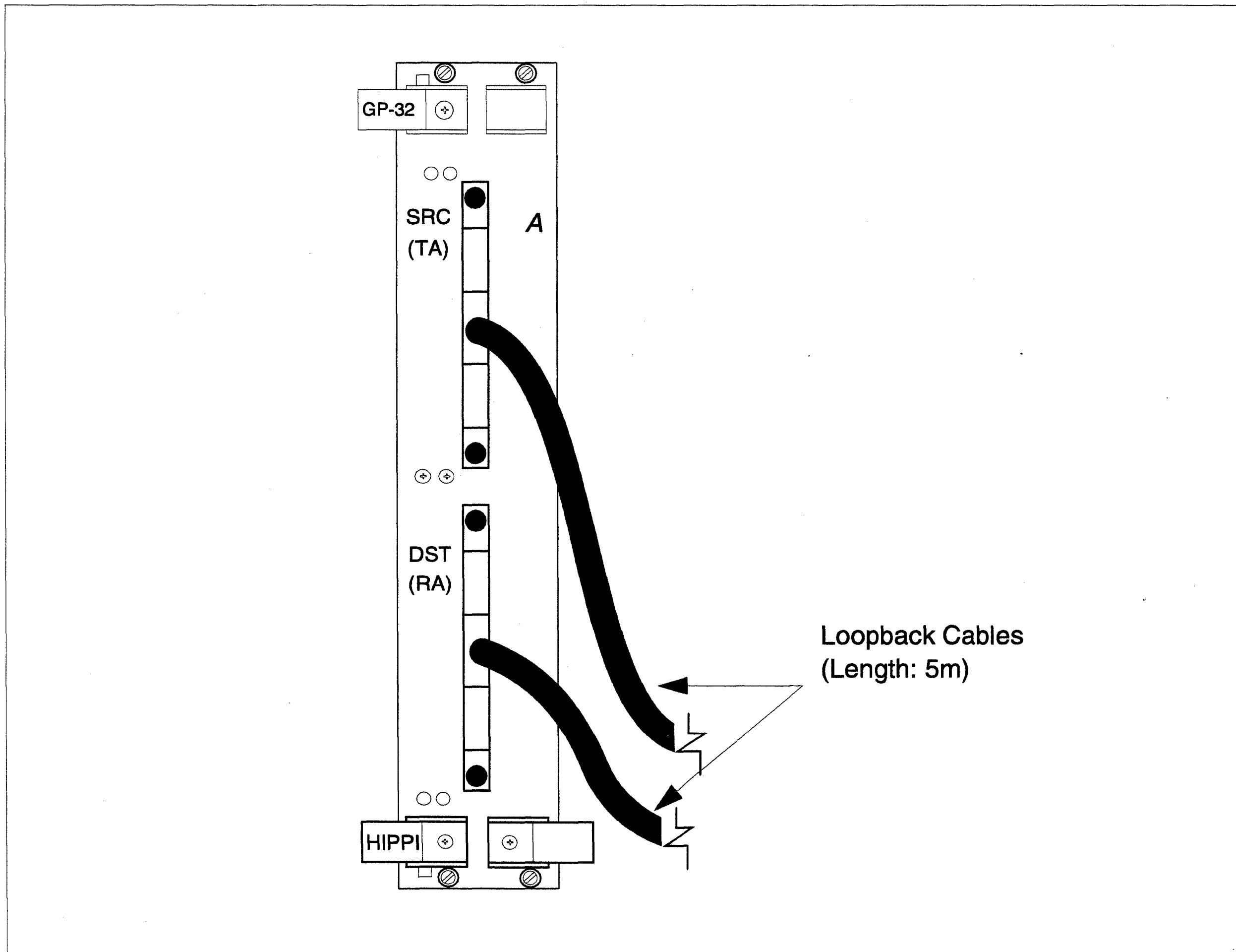

Figure 5-1. Connecting the Loopback Cables

Cable Parts and Specifications 6

This chapter contains the specifications for the internal, external, and loopback cables that connect the HIPPI controller to external devices and that are used to test the controller's operation.

Internal Cables

The two internal cables for each HIPPI controller connect to the front panel of the controller. One cable is for the source channel, one for the destination channel.

Internal Cable Characteristics

The ends of each internal cable are labeled P1 and P2. The P1 end is a right-angle, 100-pin, plug connector that has a metal housing and latching retainer hardware; it mates with J2 and J3 on the HIPPI controller board.

The P2 end of the internal cable is a standard HIPPI connector; it receives the standard *external* HIPPI cable connector (including retaining screws). The standard HIPPI female screw locks are also located on the P2 end of the cable.

Each internal cable is 6.5 feet long.

Internal Cable Implementation

The internal cables are Madison Cable DOSDK00010 or equivalent. The connectors used on this cable must be the functional equivalent of:

- P1 AMP 749611-8 or 749621-9, with latching backshell AMP 74919-1 or 749206-1.
- P2 AMP 749877-9 with female screw locks AMP 749087-1.

External Cables

The two external cables for each HIPPI controller run to an external system, switch, or device. One cable is for the source channel, one for the destination channel.

External Cable Characteristics

The external cables must meet the requirements specified in the ANSI X3T9.3/88-023 specification for connectors, pin assignments, shielding, wire color code, and electrical behavior.

Each external cable is 25 meters long.

External Cable Implementation

The external cables must be the functional equivalent of:

- 1 meter cable - AMP 749755-13.
- 5 meter cable - AMP 749755-2.
- 15 meter cable - AMP 749755-3.
- 25 meter cable - AMP 749755-4.

Loopback Cable

The loopback cable connects the HIPPI controller's source and destination channel connectors together. See Figure 5-1 on page 5-2. The cable allows you to run loopback transfers on the HIPPI controller. Some of the controller's power-on diagnostics require the use of a loopback cable.

Loopback Cable Implementation

The loopback cable is a Madison Cable DOSDO7BTIA or equivalent. The connectors used on this cable must be the functional equivalent of AMP 749070-9 with female screw locks and housing.

HIPPI Calls

A

This appendix contains manual pages for the *libhippi.a* library of routines.

See the *Paragon*™ *System C Calls Reference Manual* for manual pages for system calls unique to the operating system.

The manual pages in this appendix are also available online, using the **man** command.

HIPPI_BIND() HIPPI_BIND()

Selects the incoming data that you want to receive.

Synopsis

```
#include <sys/types.h>
#include <raw_hippi.h>

int hippi_bind(
        int ihandle,
        u_char ulp,
        short port );
```

Parameters

ihandle Specifies the HIPPI connection (ihandle) to bind.

ulp A value greater than or equal to 0x80 that specifies the selected packets. (Values below 0x80 are reserved for use by ANSI.) The maximum value for *ulp* is 0xff. A special value of 0 can be used if the device was opened with O_EXCL during the hippi_open() function, which will result in all HIPPI packets being received by this channel. *ulp* must be set to 0 if running in HIPPI_CNT mode.

port A positive value specifies the port to bind to. A -1 value for *port* indicates that the *ulp* alone should be used to select the incoming packets.

Description

You must call the **hippi_bind** function in order to read data from an open HIPPI channel. The call allows your application to establish a peer-to-peer relationship with another application. Incoming packets for your application are then correctly de-multiplexed based on the given ULP or ULP and port. (For the port to be evaluated, it must be in the first short of the D1_Area of the incoming packet.)

This routine will fail if another process is currently receiving data with the same ULP or ULP and port value.

This function supports receiving RAW HIPPI data by opening the device with O_EXCL during the **hippi_open** routine and setting the ULP=0 in this routine.

HIPPI_BIND() *(cont.)*

HIPPI_BIND() *(cont.)*

Return Value

On successful completion, **hippi_bind** returns 0. On failure, it returns -1 and sets the global variable *errno* to the appropriate value.

Errors

EADDRINUSE	Another process is currently receiving data with the same ULP or ULP and port.
EADDRNOTAVAIL	The given ULP was less than 0x80.
EBADF	The *i-handle* argument is invalid.
EIO	A library operation failed.
ENOBUFS	The device is uninitialized.
EACCES	The ULD requested requires HIPPI_EXC.
EEXIST	The i-handle is already bound.

HIPPI_CLOSE() HIPPI_CLOSE()

Closes a HIPPI connection and cleans up the state information maintained for the connection.

Synopsis

```
#include <sys/types.h>
#include <raw_hippi.h>

int hippi_close(
        int ihandle );
```

Parameters

ihandle Specifies the HIPPI connection (ihandle) to be closed.

Description

The **hippi_close** routine closes the HIPPI connection referred to by the *ihandle* argument and cleans up the library state for this connection.

Return Value

On successful completion, **hippi_close** returns 0; on failure, it returns -1, and sets the global variable *errno* to the appropriate value.

Errors

EBADF	The *ihandle* argument referenced an invalid HIPPI connection.
ENXIO	A library operation failed.
EADDRINUSE	HIPPI_CLOSE encountered an error.
EIO	HIPPI_CLOSE encountered an error.

HIPPI_CONFIG() HIPPI_CONFIG()

Specifies packet framing semantics for HIPPI connections that are opened in the HIPPI_DATA mode.

Synopsis

```
#include <sys/types.h>
#include <raw_hippi.h>

int hippi_config(
        int ihandle,
        u_long ifield,
        u_char ulp,
        u_long b,
        char *d1_data,
        u_short d1_len );
```

Parameters

ihandle Specifies an open HIPPI connection.

ifield The address to which the packet is to be sent. The *ifield* argument must comply with the HIPPI-SC specification.

ulp The upper layer protocol to send the packet to.

b Indicates where the D2_Area starts in the packet:

0	Indicates that D2_Area is not aligned
1	Indicates that D2_Area is burst-boundary aligned

d1_data A pointer to the D1_data area.

d1_len Size of D1 data (the number of 64-bit words in *d1_data*).

Description

The **hippi_config** function should only be used for connections opened in HIPPI_DATA mode; it should not be used for connections opened in HIPPI_RAW mode.

HIPPI_CONFIG() *(cont.)*

HIPPI_CONFIG() *(cont.)*

The **hippi_config** function sets the format characteristics of the first burst of outbound packets. Parameter *b* should be TRUE if D2 must start at a burst boundary. If *d1_data* isn't NULL, *d1_len* bytes are copied into the *d1_data* area of the HIPPI frame. This combination of options result in a small data copy, but enables users to fully control the contents of the first burst.

Return Value

On successful completion, **hippi_config** returns 0; on failure, it returns -1 and sets the global variable *errno* to the appropriate value. (Failure indicates an invalid ihandle; the validity of the other parameters can only be determined at run time.)

Errors

EBADF	The connection referenced by the *ihandle* argument is not a HIPPI_DATA mode connection or is invalid.

HIPPI_MEMFREE() HIPPI_MEMFREE()

Releases the memory acquired by either **hippi_memget**, **hippi_read**, or **hippi_read_request**.

Synopsis

```
#include <sys/types.h>
#include <raw_hippi.h>

int hippi_memfree(
        int ihandle,
        char *ptr,
        u_long size,
        int how );
```

Parameters

ihandle	Specifies the HIPPI connection (ihandle) associated with the memory to free.
ptr	Pointer to the memory to free.
size	Size, in bytes, of the block of memory to free.
how	This parameter should be:

0	If the memory was allocated by **hippi_memget**.
1	If the memory was acquired by **hippi_read** or **hippi_read_request**.

Description

This routine releases memory pointed to by *ptr*. This routine must be called for memory allocated by **hippi_memget** and for memory acquired by **hippi_read** or **hippi_read_request**.

Return Value

On successful completion, **hippi_memfree** returns 0; on failure, it returns -1, and sets the global variable *errno* to the appropriate value.

HIPPI_MEMFREE() *(cont.)*

HIPPI_MEMFREE() *(cont.)*

Errors

EBADF	The *ihandle* argumemt is invalid.
EINVAL	*how* is not valid.

HIPPI_MEMGET()

HIPPI_MEMGET()

Allocates memory for a HIPPI connection.

Synopsis

```
#include <sys/types.h>
#include <raw_hippi.h>

char * hippi_memget(
        int ihandle,
        u_long size );
```

Parameters

ihandle Specifies the HIPPI connection.

size Specifies the number of bytes to allocate.

Description

The **hippi_memget** routine allocates at least *size* bytes. For HIPPI_RAW connections, the value of *size* should be large enough to hold the largest packet the user will send, including the I-field, headers, and data. For HIPPI_DATA connections, *size* only needs to be large enough for the data. The **hippi_memget** routine must be used with HIPPI_DATA connections and can be used with HIPPI_RAW. In HIPPI_DATA mode, you should call **hippi_config** routine before calling **hippi_memget**.

Return Value

On successful completion, **hippi_memget** returns a pointer to the allocated memory. For HIPPI_RAW mode, the pointer points to the start of the HIPPI header area. For HIPPI_DATA mode, the pointer points to the place in the packet where HIPPI expects the user to put data (based on the parameters supplied in the call to **hippi_config**).

On failure, **hippi_memget** returns a null pointer, and sets the global variable *errno* to the appropriate value.

HIPPI_MEMGET() *(cont.)*

HIPPI_MEMGET() *(cont.)*

Errors

EBADF	The *ihandle argumemt* is invalid.
ENOMEM	There is not enough memory on the HIPPI board.

See Also

hippi_config

HIPPI_OPEN() HIPPI_OPEN()

Establishes an open HIPPI connection.

Synopsis

```
#include &lt;fcntl.h&gt;
#include &lt;raw_hippi.h&gt;

int hippi_open(
        char *dev_name,
        u_long hippi_mode,
        u_long mode );
```

Parameters

dev_name The HIPPI device through which connection is to be established. (Use the **rmknod** command to create the device.)

hippi_mode The HIPPI mode must be either HIPPI_RAW or HIPPI_DATA. Optional modes include HIPPI_CNT and HIPPI_MPC. When using HIPPI_CNT or HIPPI_MPC modes, the device must be opened with UNIX I/O mode O_EXCL. Optional modes can ONLY be used with HIPPI_RAW transfers.

mode I/O mode: should be O_RDONLY, O_WRONLY, O_RDWR, or O_EXCL. (Other I/O modes, such as 0_CREAT and O_TRUNC make no sense.) The mode must match the permissions set on the devices by the system administrator (using the **chmod** command).

Description

The **hippi_open** routine opens a HIPPI connection and selects the connection's HIPPI and I/O modes. HIPPI_MPC mode allows you to specify multiple packets per connection for writes and reads via both HIPPI channels. HIPPI_CNT mode allows you to specify multiple I/O writes and reads per connection and packet via both HIPPI channels.

Return Value

On successful completion, **hippi_open** returns a positive integer or zero. This integer, called the *ihandle* argument, is used in subsequent HIPPI operations for the connection. On failure, **hippi_open** returns -1, and sets the global variable *errno* to the appropriate value.

HIPPI_OPEN() *(cont.)*

HIPPI_OPEN() *(cont.)*

Errors

EMFILE	There are too many open files.
EINVAL	The *hippi_mode* argument is invalid.
ENOENT	No such file.
EIO	HIPPI_OPEN encountered an error.
ENXIO	HIPPI_OPEN encountered an error.
EACCES	The MPC node setup failed.

HIPPI_READ() HIPPI_READ()

Reads from an open HIPPI connection.

Synopsis

```
#include <sys/types.h>
#include <raw_hippi.h>

long hippi_read(
        int ihandle,
        char **ptr,
        u_long bytes_wanted);
```

Parameters

ihandle	Specifies the HIPPI connection to read from.
ptr	Address of data read.
bytes_wanted	Number of bytes to read.

Description

The **hippi_read** function attempts to read from the HIPPI connection specified by the *ihandle* argument, the number of bytes specified by the *bytes_wanted* argument. The data returned is referenced by the *ptr* argument, which points to the start of the FP header area.

You must call **hippi_bind** before calling **hippi_read** (otherwise **hippi_read** will fail).

When running in HIPPI_CNT mode, the read length must be at least 2048 bytes.

Return Value

On successful completion, **hippi_read** returns the number of bytes read.

On failure, **hippi_read** returns -1 and sets the global variable *errno* to the appropriate value.

HIPPI_READ() *(cont.)*

HIPPI_READ() *(cont.)*

Errors

EBADF	The *ihandle* argument referenced an invalid HIPPI connection.
EIO	A library operation failed.
ENOMEM	HIPPI cannot allocate enough memory.
EMSGSIZE	HIPPI_CNT mode cannot support requests less than 2K bytes.

See Also

hippi_bind()

HIPPI_READ_COMPLETE()

HIPPI_READ_COMPLETE()

Retrieves a packet on an open HIPPI connection.

Synopsis

```
#include <sys/types.h>
#include <raw_hippi.h>

long hippi_read_complete(
        int ihandle,
        char **ptr);
```

Parameters

ihandle Specifies the HIPPI connection to read from.

ptr Pointer to packet read.

Description

The **hippi_read_complete** function attempts to retrieve a packet from the HIPPI connection specified by the *ihandle* argument. The data returned is referenced by the *ptr* argument, which points to the start of the FP header area. The **hippi_read_complete** function is a non-blocking function which returns either success when a packet has arrived or failure (setting errno) when a packet hasn't. This function can only succeed if there is at least one buffer pending in the driver, which would have been allocated using **hippi_request_request()**.

You must call **hippi_read_request** before calling **hippi_read_complete** (otherwise **hippi_read_complete** will fail).

Return Value

On successful completion, **hippi_read_complete** returns the number of bytes read.

On failure, **hippi_read_complete** returns -1 and sets the global variable *errno* to the appropriate

HIPPI_READ_COMPLETE() *(cont.)*

HIPPI_READ_COMPLETE() *(cont.)*

Errors

EBADF	The *ihandle* argument referenced an invalid HIPPI connection.
EIO	A library operation failed.
EWOULDBLOCK	There was no packet available.
EINVAL	There were no buffers previously allocated via **hippi_read_request()** available.

See Also

hippi_read_request()

HIPPI_READ_REQUEST()

HIPPI_READ_REQUEST()

Posts a read on an open HIPPI connection.

Synopsis

```
#include <sys/types.h>
#include <raw_hippi.h>

int hippi_read_request(
        int ihandle,
        long bytes_wanted);
```

Parameters

ihandle	Specifies the HIPPI connection to read from.
bytes_wanted	Number of bytes to read.

Description

The **hippi_read_request** function posts a request to read from the HIPPI connection specified by the *ihandle* argument, the number of bytes specified by the *bytes_wanted* argument. This function is intended to provide an asynchronous read capability to the application. Each call to this function allocates in the HIPPI device driver a buffer for an incoming packet. The **hippi_read_complete()** function should be called to retrieve the received packet.

You must call **hippi_bind** before calling **hippi_read_request** (otherwise **hippi_read_request** will fail).

When running in HIPPI_CNT mode, the read length must be at least 2048 bytes.

Return Value

On successful completion, **hippi_read_request** returns 0.

On failure, **hippi_read_request** returns -1 and sets the global variable *errno* to the appropriate value.

HIPPI_READ_REQUEST() *(cont.)*

HIPPI_READ_REQUEST() *(cont.)*

Errors

EBADF	The *ihandle* argument referenced an invalid HIPPI connection.
EIO	A library operation failed.
ENOMEM	The function was not able to allocate memory.
EINVAL	The bytes requested were less than 0.

See Also

hippi_read(), hippi_read_complete()

HIPPI_WRITE()

Writes to an open HIPPI connection.

Synopsis

```
#include <sys/types.h>
#include <raw_hippi.h>

int hippi_write(
        int ihandle,
        char *ptr,
        u_long len );
```

Parameters

ihandle	The HIPPI connection to which packet is written.
ptr	Pointer to write buffer.
len	Number of bytes to write.

Description

The **hippi_write**() function writes a packet to the open HIPPI connection referenced by the *ihandle* argument. If HIPPI_RAW mode was selected in the call to **hippi_open**(), the *ptr* argument points to the fully formatted frame (including the I-field), and **hippi_write()** writes the packet starting at the location specified by the *ptr* argument. If HIPPI_DATA mode was selected in the call to **hippi_open**(), **hippi_write()** completes the packet, adjusting the *ptr* argument for copying *D1_data*, *FP header*, and *I-field* as specified in the call to **hippi_config**(), and then writes the packet. For the HIPPI_RAW mode, the number of bytes specified by the *len* argument must include the *I-field*, *FP Header*, *D1_Data*, and *D2_Data*. For the HIPPI_DATA mode, the *len* argument only includes the *D2_Data*.

If HIPPI_CNT or HIPPI_MPC mode was selected with HIPPI_RAW mode in the call to **hippi_open**(), then *ptr* points to the fully formatted frame, including the I-field, only on the first packet. Subsequent packets will not need an I-field header, since the SRC channel connection is asserted until **hippi_close**().

When running in HIPPI_CNT mode the first write, not including the HIPPI I-field (32 bytes), and all subsequent writes must be a multiple of 1024 bytes (HIPPI BURST) to avoid HIPPI channel protocol violations.

HIPPI_WRITE() *(cont.)*

HIPPI_WRITE() *(cont.)*

Return Value

On successful completion, **hippi_write()** returns 0; on failure it returns -1 and sets the global variable *errno* to the appropriate value.

Errors

EBADF	The *ihandle* argument referenced an invalid HIPPI connection.
EIO	An I/O error occurred.
EINVAL	The *ptr* argument is invalid (NULL).
ENOMEM	The HIPPI board has insufficient memory.

See Also

hippi_config(), hippi_open()

HIPPI Commands B

This appendix contains manual pages for the **hippi_setmap** and **hippi_showmap** commands.

See the *Paragon*™ *System Commands Reference Manual* for manual pages for system calls unique to the operating system.

The manual pages in this appendix are also available online, using the **man** command.

HIPPI_SETMAP

HIPPI_SETMAP

Loads a HIPPI network routing table into the HIPPI driver.

Syntax

hippi_setmap [-d] *interface* [*mapfile*]

Arguments

-d Deletes all entries in the routing table.

interface Specifies the HIPPI interface of the board whose routing table is being loaded. You must use **ifconfig** to define the interface before using it in the *interface* argument here.

mapfile Specifies the file that contains the HIPPI network routing table. The **hippi_setmap** command loads this routing table into the HIPPI driver. If a map file is not specified, **hippi_setmap** takes data from the standard input.

Description

The **hippi_setmap** command loads the HIPPI network routing table into the HIPPI server. The routing table lists IP addresses, ULAs, and I-fields and enables **hippi_setmap** to map IP addresses to I-fields.

The specified table is appended to the routing table being used currently. If you want to replace the routing table, you must first delete the existing one using the **-d** command line switch, and then specify the new routing table.

The **hippi_setmap** command reads from the standard input until you press the *End-of-File* key sequence, or it takes its input from the contents of the optional *mapfile*.

You must be a logged in as *root* to use **hippi_setmap**.

HIPPI_SETMAP *(cont.)*

HIPPI_SETMAP *(cont.)*

Examples

The **hippi_setmap** command takes as input a file formatted like the following:

```
#HIPPI Network Routing Table
#
#IP Address         ULA                  I-field
#-----------        --------------       ----------
 192.9.3.1          1:0c:34:65:0:26      0x1000000    #Zombie
 192.9.3.2          1:0c:34:65:0:27      0x1000002    #Jaguar
 192.9.3.3          1:0c:34:65:0:28      0x1000001    #Balu
 192.9.3.4          1:0c:34:65:0:29      0x1000003    #Carlsbad
```

- The IP Address column contains either an internet address or a host name.
- The I-field column is in hex, prefixed with 0x.
- All characters following the pound sign "#" on a line are ignored and serve as comments.

In the above table, the camp-on bit in the I-field is set. This instructs the HIPPI switches to attempt a connection until the connection is completed or the source cancels the connection request.

To append the file *hippi.map* into the server, type:

```
hippi_setmap  ifhip0 hippi.map
```

To delete the existing map from the server and then load the contents of */etc/hippi.map*, type:

```
hippi_setmap -d ifhip0
hippi_setmap ifhip0 /etc/hippi.map
```

See Also

hippi_showmap, ifconfig

HIPPI_SHOWMAP HIPPI_SHOWMAP

Displays the current HIPPI network routing table.

Syntax

hippi_showmap [**-n**]

Arguments

-n Causes the command to search for and display the ASCII names of the hosts in the routing table. These names correspond to IP addresses.

Description

The **hippi_showmap** command displays the current HIPPI network routing table (loaded by the most recent **hippi_setmap** command).

Without the -n argument, **hippi_showmap** displays IP addresses in the first column:

```
#IP Address         ULA                 I-field
#-----------        --------------      ----------
192.9.3.1
192.9.3.2
```

With the -n argument, **hippi_showmap** displays hostnames in the first column:

```
#Hostname           ULA                 I-field
#-----------        --------------      ----------
Tornado
Hurricane
```

Example

To display the current routing table, using hostnames, type the following:

```
hippi_showmap
```

See Also

hippi_setmap

Usage Notes C

Introduction

This Appendix contains practical advice and examples for using the HIPPI interface in both the HIPPI_DATA and HIPPI_RAW modes.

Using the HIPPI_DATA Mode

The HIPPI library provides all of the system calls needed to send and receive data using the HIPPI device. The following notes are a summary of what is necessary to write a program to use HIPPI_DATA mode to exchange data between nodes on a Paragon system.

HIPPI Packets

A HIPPI packet has three main parts. The first 32 bytes contains the *I-field*, the next 8 bytes define the *FP_Heade*r, and the remaining bytes include any *D1_Data* and the *D2_Data*. The *raw_hippi.h* file contains structure definitions that simplify access to the various fields in the header.

In HIPPI_DATA mode, you are not expected to know the size or structure of an outbound packet. The HIPPI library calls automatically format and build outbound packets by using the input parameter that you supply. As a result, there are some fields in the FP header that you have no control over.

FP Header

Here are general rules for that define how the FP header shown in Figure C-1 is built:

- The *ULP* and the *B* bit are set to the values you supply with the **hippi_config()** call.

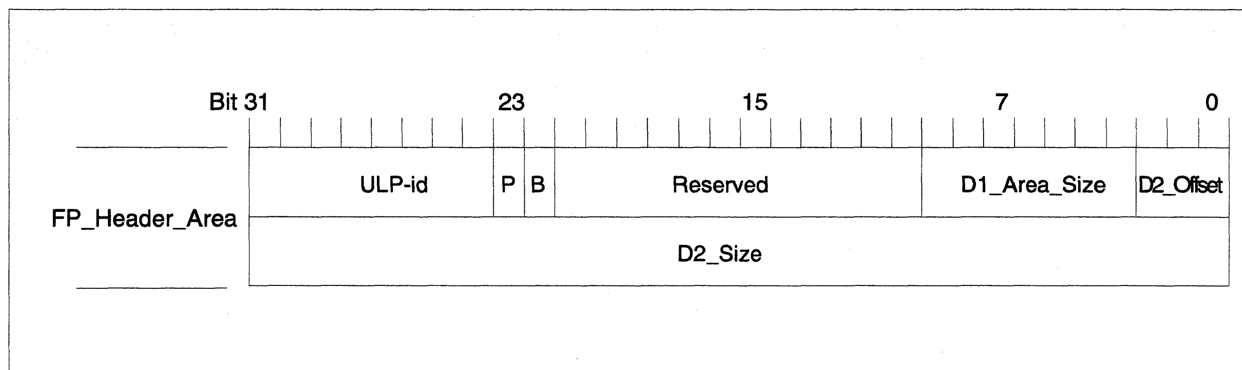

Figure C-1. FP Header

- The *P* bit is set to 1 if you supply a greater-than-zero-length *D1_Data_Set*, otherwise, it is set to zero.
- The 11 reserved bits are always transmitted as zero's.
- If *B* is set to zero, *D1_Area_Size* is set to the size of *D1_Data_Set*. If *B* is set to 1, *D1_Area_Size* should be set to 127 to burst-align the *D2_Data_Set*.
- *D2_Offset* is always set to zero.
- *D2_Size* is set to the size of *D2_Data_Set*.

hippi_open(dev_name, hippi_mode, mode)

hippi_open creates an *I-handle* that is used to access the HIPPI device. You can specify which HIPPI device to use, the mode you are using (HIPPI_RAW or HIPPI_DATA), and the I/O mode (read, write, both...). Make sure that the permissions of the HIPPI device in */dev* are set appropriately.

hippi_bind(ihandle, ulp, port)

hippi_bind binds the *I-handle* to a specific *ulp/port*. The *I-handle* then only receives packets that are sent to this specific *ulp/port*. This function needs to be called only if the *I-handle* is going to receive data. If used, the port must appear in the first 16 bits after the *D2_Size* in the *FP_Header*.

hippi_config(ihandle, ifield, ulp, b, d1_data, d1_len)

hippi_config uses the parameters to format the *I-field*, the *FP_Header_Area* and the *D1_Area* of an outbound packet:

- This function can only be used for a connection opened in *HIPPI_DATA* mode.
- If *d1_data* is *NULL*, *d1_len* is ignored, and the *D1_Data_Set* is considered not present.
- **hippi_config** returns -1 if *d1_len* is greater than the maximum *D1_Area_Size*, 127 64-bit words.

hippi_memget(ihandle, size)

hippi_memget allocates memory for the outbound packet. The behavior of this function depends on which mode you are using.

- In *HIPPI_DATA* mode, the size parameter should be large enough to fit the largest *D2_Data_Set* you will send.
- The pointer returned by **hippi_memget** points to the place in the packet where the *D2_Data_Set* is to be placed. In order to calculate exactly where this point is, **hippi_config** must be called prior to **hippi_memget**. This pointer can then be used for **hippi_write**.

hippi_write(ihandle, ptr, length)

In HIPPI_DATA mode, **hippi_write** uses the *ihandle* to send a packet that contains a *D2_Data_Set* that is *length* bytes long starting from memory location *ptr*. See Figure C-2. The *I-field* and *FP header* information is automatically filled in for you based on the parameters supplied by the *hippi_config* call. The *I-field* is for routing information, the *FP header* indicates how the remaining data is to be interpreted.

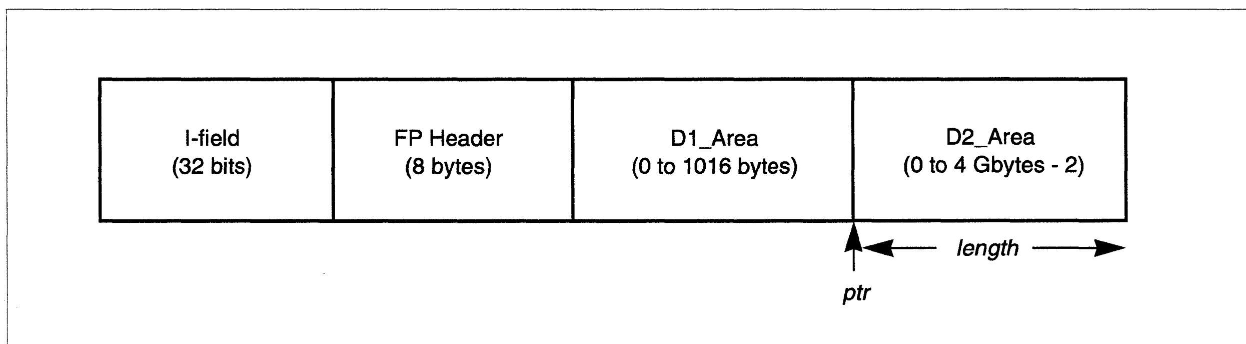

Figure C-2. The *ptr* and *length* Arguments for hippi_write (HIPPI_DATA Mode)

Here is a short and simple example that formats a packet and sends 1024 bytes of data using HIPPI_DATA mode. Note that the *D1_Area* is not present in this example.

```
int ihandle, i;
int how = 0;
char *D2_Data_Set, *tmp_ptr;
u_long length = 1024;
u_long ifield = 0x01000001;
u_long ulp = 128;
u_long b = 0; char *d1_data = NULL;
u_short d1_len = 0;

/* Create HIPPI connection */
ihandle = hippi_open("/dev/hippi", HIPPI_DATA, O_RDWR);

/* Configure Ifield, FP Header and D1_Area */
hippi_config(ihandle, ifield, ulp, b, d1_data, d1_len);

/* Allocate the memory */
D2_Data_Set = hippi_memget(ihandle, length);

/* Copy your data into the buffer returned by hippi_memget().
 * This is a byte copy. If your data is integers, than
 * extra space is needed.
 */
tmp_ptr = D2_Data_Set;

for(int i = 0; i < length; i++) {
*tmp_ptr = your_data[i];
tmp_ptr++; }

/* Send your packet */
hippi_write(ihandle, D2_Data_Set, length);

/* Free the memory */
hippi_memfree(ihandle, D2_Data_Set, length, how):

/* Close HIPPI connection */
hippi_close(ihandle);
```

hippi_read(ihandle, ptr, bytes_wanted)

hippi_read works the same in both *HIPPI_RAW* and *HIPPI_DATA* modes. It is used to read data from an *I-handle* (**hippi_bind** must be called before reading). The arguments to this function include the *I-handle* to read from the reference of a pointer to the received packet (which includes an FP header) and the number of bytes you expect to read. If the incoming packet is larger than the number specified by *bytes_wanted,* **hippi_read()** returns an error.If the incoming packet is equal to or smaller than the number specified by *bytes_wanted*, **hippi_read()** returns the number of bytes actually read. Note that the I-field is not include in this buffer.

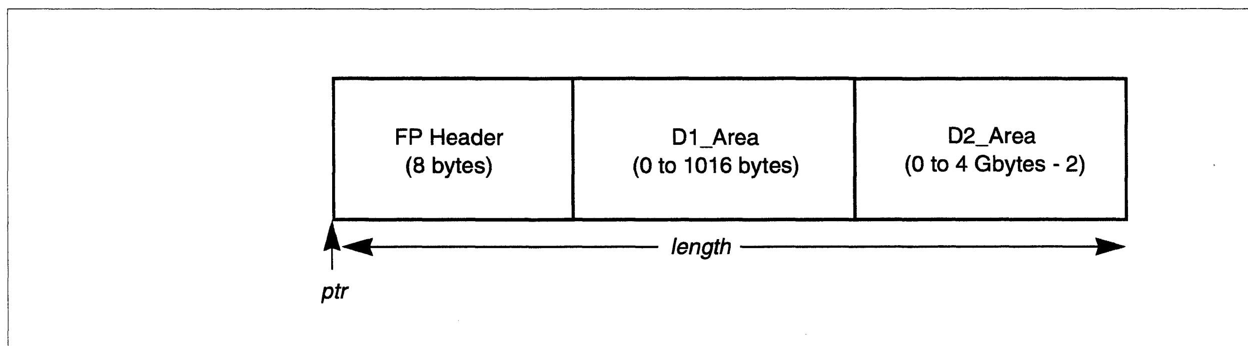

Figure C-3. The *ptr* and *length* Values Returned by hippi_read

To get at the data (assuming there was no *D1_Data* specified in the *FP header*) you must increment the pointer that is returned by **hippi_read** by *sizeof(union hippi_fp_header)*. The length that is returned includes the bytes for the *FP header*. Therefore, the actual data (minus header) that is received starts at *ptr + sizeof(struct hippi_header)* and is *length - sizeof(struct hippi_header)* bytes long.

To get at the *FP header*, cast *ptr* to *(union hippi_fp_header)* and use the field names as defined in *raw_hippi.h.*

hippi_memfree(ihandle, ptr, size, how)

hippi_memfree releases the memory allocated by **hippi_memget** or **hippi_read**. It is very important, and good style, to release the memory you use.

- Use the pointer returned by **hippi_memget**, the length passed into **hippi_memget** and *how = 0* if *ptr* was allocated by **hippi_memget**.

- Use the pointer and length returned by **hippi_read** and *how = 1* if *ptr* was from allocated by **hippi_read**.

hippi_read_request(ihandle, bytes_wanted)

hippi_read_request works the same in both HIPPI_RAW and HIPPI_DATA modes. It allocates buffer space of size *bytes_wanted* for a particular I-handle to provide asynchronous read capability.

You must call **hippi_read_complete** in order to retrieve the received packet. It is the responsibility of the programmer to ensure that the incoming data order and sizes match the order and sizes of the **hippi_read_request**s. For example, if you expect a particular I-handle's second packet to contain 64K of data, your second call to **hippi_read_request** should be posted (with the *bytes_wanted* parameter at least 64K) before the second **hippi_write** occurs or an error will occur.

hippi_read_complete(ihandle, ptr)

hippi_read_complete works the same in both HIPPI_RAW and HIPPI_DATA modes. It is a non-blocking call that retrieves received packets for a particular I-handle. There is a one-to-one correspondence between **hippi_read_request**s and **hippi_read_complete**s—you will get an error if you call **hippi_read_complete** when there are no **hippi_read_request**s pending. If a **hippi_read_request** is pending but the data has not arrived yet from a **hippi_write**, **hippi_read_complete** will return -1 and set *errno* to EWOULDBLOCK.

The following example is one way of using **hippi_read_complete** to continue polling until data arrives.

```
while ((count = hippi_read_complete(ihandle, &buf)) == -1)
  if(errno != EWOULDBLOCK) {
     nx_perror("hippi_read_complete() FAILED");
     exit(1);}
```

hippi_close(ihandle)

hippi_close closes the HIPPI connection. It works the same in both *HIPPI_RAW* and *HIPPI_DATA* modes.

Code Example

There are many ways to format the buffer and allow for different types of data— *char*, *int*, *long*, *float*... This is a working program that shows where and how the pointers need to be manipulated, and illustrates the use of the HIPPI library calls to read and write data in *HIPPI_DATA* mode.

```
/******************************************************************************
 *
 * Title: HIPPI_DATA mode data exchange program.
 *
 * Description:
 * This program tests RAW HiPPI (HIPPI_DATA mode) data exchange for
 * ANSI HiPPI-FP compliance. Node 0, the client, transmits the data and
 * node 1, the server, receives and verifies the data.
 *
******************************************************************************/

#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <setjmp.h>
#include <sys/param.h>
#include <sys/timers.h>
#include <raw_hippi.h>

#define TIMEOUT_VAL 60
#define D1_CHAR    'a'
#define D2_CHAR    'Z'

/* global vars */
int       failed = 0;              /* number of failures */
int       node;                    /* node number */
u_long    cci = 0x01000001;        /* I-field */
u_long    ulp = 0x80;              /* Upper Level Protocol */
u_long    port = -1;               /* Application only binding to ulp,
                                   /*    port value set to -1 */
u_long    b = 0;                   /* Setting b bit to zero indicates D2_Area not
                                    *    burst aligned */
u_short   d1_len = 104;            /* D1_Area_Size=104 64-bit words=832 bytes */
u_long    d2_len = 60*1024;        /* D2_Data_Set_Size = 60 KB of data */

jmp_buf timeout;

main()
{
     node = mynode();
```

```
    if (node > 1) {
        printf("Node %d not need for this test.\n", node);
        fflush(stdout);
        exit(0);
    }

    /*
     * Do the tests
     * Node 0, the client, will transmit the data.
     * Node 1, the server, will receive the data and verify ANSI HiPPI-FP
     * compliance.
     */
    if(node == 0) {
        sleep(2);
        hippi_client();}
    else if(node == 1)
        hippi_server();

    /*
     * Report test results
     */
    if (failed == 0)
        printf("Node %d: HIPPI Data Exchange Done: *** PASSED ***\n", node);
    else
        printf("Node %d: HIPPI Data Exchange Done: *** FAILED ***\n", node);

        exit(failed);
} /* main */


/***************************************************************************
 *
 * alrm_handler()
 *
 * Does a longjmp() if called. Attached to signal SIGALRM
 *
 ***************************************************************************/

void alrm_handler()
{
    printf("Node %d: SIGALRM occured!\n", node);
    fflush(stdout);

    longjmp(timeout, 1);
}
```

```
/***************************************************************************
 *
 * hippi_client()
 *
 * This procedure uses a HIPPI_DATA-mode connection to
 * build and transmit a HIPPI packet to the server.
 *
 ***************************************************************************/

hippi_client()
{
    int ihandle;  /* ihandle corresponding to the HIPPI connection */
    int   err;
    char *d1_data;
    char *d2_data;

    /* print the parameter values to be tested */
    printf("Node %d: HIPPI_CLIENT starting test case\n", node);
    printf(" I-field = 0x%x\n ulp = 0x%x\n port = %d\n", cci, ulp, port);
    printf(" b = %d\n d1_len = %d\n d2_len = %d\n", b, d1_len, d2_len);
    fflush(stdout);

    /* create HIPPI connection */
    if((ihandle = hippi_open("/dev/hippi", HIPPI_DATA, O_RDWR)) < 0) {
        nx_perror("HIPPI_CLIENT: FAILED hippi_open()");
        failed++;}

    /* get memory for d1_data area. D1_len is multiplied by 8
     * to convert from 64-but words to bytes */
    if((d1_data = (char *) malloc(d1_len*8)) < 0) {
        nx_perror("HIPPI_CLIENT: FAILED malloc()");
        failed++;}

    /* fill d1_data with fill_char */
    memset(d1_data, D1_CHAR, d1_len*8);

    /* configure I-field, FP Header and D1_Area */
    if(hippi_config(ihandle, cci, ulp, b, d1_data, d1_len) < 0) {
        nx_perror("HIPPI_CLIENT: FAILED hippi_config()");
        failed++;}

    /* allocate memory for d2_data area */
    if((d2_data = hippi_memget(ihandle, d2_len)) == NULL) {
        nx_perror("HIPPI_CLIENT: FAILED hippi_memget()");
        failed++;}
    else
        /* fill d2_data with fill_char */
        memset(d2_data, D2_CHAR, d2_len);
```

```
    if(setjmp(timeout) == 0) {
        /* attach signal to alarm handler */
        signal(SIGALRM, alrm_handler);

        alarm(TIMEOUT_VAL);

        /* write data */
        errno = 0;
        if(hippi_write(ihandle, d2_data, d2_len) == -1) {
            nx_perror("HIPPI_CLIENT: FAILED hippi_write()");
            failed++;}

        alarm(0);}

    else {
        printf("Node %d: HIPPI_CLIENT: FAILED hippi_write timed out\n", node);
        failed++;}

    /* free memory allocated by hippi_memget */
    hippi_memfree(ihandle, d2_data, d2_len, 0);

    /* close HIPPI_DATA-mode connection */
    hippi_close(ihandle);
    return;

} /* hippi_client */

/************************************************************************
 *
 * hippi_server()
 *
 * Read the HIPPI packet sent by the server and verify ANSI HIPPI-FP
 * compliance.
 *
 ************************************************************************/

hippi_server()
{
    int  i;                         /* Temporary loop counter */
    int  ihandle; /* ihandle corresponding to the HIPPI
                                     * connection */
    int  bad_cnt = 0;/* Data mismatch counter */

    char     *rx_ptr;/* Pointer to the FP Header of the packet */
    char     *tmp_ptr;/* temporary pointer */
    u_long   rx_size;/* Number of bytes to read by hippi_read() */
       longcount;/* Number of bytes actually read by hippi_read */
    union    hippi_fp_header *fp_ptr; /* FP Header structure */
```

```
    /* create HIPPI_DATA-mode connection */
    if((ihandle = hippi_open("/dev/hippi", HIPPI_DATA, O_RDWR)) < 0) {
         nx_perror("HIPPI_SERVER: FAILED hippi_open()");
         failed++;}

    /* bind for read from client */
    if(hippi_bind(ihandle, ulp, port) == -1) {
         nx_perror("HIPPI_SERVER: FAILED hippi_bind()");
         failed++;}

    if(setjmp(timeout) == 0) {
         /* attach signal to alarm handler */
         signal(SIGALRM, alrm_handler);

         alarm(TIMEOUT_VAL);

         /* read data */
         if((count = hippi_read(ihandle, &rx_ptr, rx_size)) == -1) {
              nx_perror("HIPPI_SERVER: FAILED hippi_read()");
              failed++;}

         alarm(0);

         /* Check received data for ANSI HIPPI-FP compliance */

         /* convert fp header to host order */
         fp_ptr = (union hippi_fp_header *)rx_ptr;
         fp_ptr->words.w1 = ntohl(fp_ptr->words.w1);
         fp_ptr->words.w2 = ntohl(fp_ptr->words.w2);

         /* check ULP */
         if(fp_ptr->fields.ulp != ulp) {
              printf("Node %d: HIPPI_SERVER: FAILED ULP wrong, expected 0x%x,
got 0x%x\n", node, ulp, fp_ptr->fields.ulp);
              failed++;}

         /* Check P bit - it should be 1 since the D1_Area is present */
         if(fp_ptr->fields.p != 1) {
              printf("Node %d: HIPPI_SERVER: FAILED p bit wrong, expected 0x%x,
got 0x%x\n", node, 1, fp_ptr->fields.p);
              failed++;}

         /* Check B bit - it should be the same as the outbound packet */
         if(fp_ptr->fields.b != b) {
              printf("Node %d: HIPPI_SERVER: FAILED b bit wrong, expected 0x%x,
got 0x%x\n", node, b, fp_ptr->fields.b);
              failed++;}
```

```
        /* Check reserved bits - always zeros */
        if(fp_ptr->fields.res != 0) {
             printf("Node %d: HIPPI_SERVER: FAILED res bits wrong, expected
0x%x, got 0x%x\n", node, 0, fp_ptr->fields.res);
             failed++;}

        /* Check d1_size - it should be the size of D1_Data-Set*/
        if(fp_ptr->fields.d1_size != d1_len) {
             printf("Node %d: HIPPI_SERVER: FAILED d1_size wrong, expected %d,
got %d\n", node, d1_len, fp_ptr->fields.d1_size);
             failed++;}

        /* Check d2_offset - always zero in the HIPPI_DATA mode */
        if(fp_ptr->fields.d2_off != 0) {
        printf("Node %d: HIPPI_SERVER: FAILED d2_off wrong, expected 0x%x, got
0x%x\n", node, 0, fp_ptr->fields.d2_off);
             failed++;}

        /* Check d2_size - it should be the size of D2_Data_Set */
        if(fp_ptr->fields.d2_size != d2_len) {
             printf("Node %d: HIPPI_SERVER: FAILED d2_size wrong, expected %d,
got %d\n", node, d2_len, fp_ptr->fields.d2_size);
             failed++;}

        /* Check d1_data_set.  The pointer returned by hippi_read() points
         * to the beginning of the FP Header.  The D1_Data_Set (if it
         * exists) immediately follows the FP Header.  To get to the
         * D1_Data_Set, increment the pointer returned by hippi_read()
         * sizeof(union hippi_fp_header) bytes. */
        tmp_ptr = rx_ptr + sizeof(union hippi_fp_header);
        for (i = 0; i < d1_len*8; i++) {
             if (*tmp_ptr != D1_CHAR)
                  bad_cnt++;
                  tmp_ptr++;}
        if (bad_cnt > 0) {
             printf("Node %d: HIPPI_SERVER: FAILED d1_data wrong, %d bytes did
not match\n", node, bad_cnt);
             failed++;}

        /* Check d2_data_set.  The pointer returned by hippi_read() points
         * to the beginning of the FP Header.  The D2_Data_Set (if it
         * exists) immediately follows the d2_offset.  To get to the
         * D2_Data_Set, increment the pointer returned by hippi_read()
         * sizeof(union hippi_fp_header) + fp_ptr->fields.d1_size*8 +
         * fp_ptr->fields.d2_off bytes. */
        bad_cnt = 0;
        tmp_ptr = rx_ptr + sizeof(union hippi_fp_header) +
fp_ptr->fields.d1_size*8 + fp_ptr->fields.d2_off;
```

```
          for (i = 0; i < d2_len; i++) {
               if (*tmp_ptr != D2_CHAR)
                    bad_cnt++;
                    tmp_ptr++;}
          if (bad_cnt > 0) {
               printf("Node %d: HIPPI_SERVER: FAILED d2_data wrong, %d bytes did
not match\n", node, bad_cnt);
               failed++;}
     }
     else {
          printf("Node %d: HIPPI_SERVER: FAILED hippi_read timed out\n", node);
          fflush(stdout);
          failed++;}

     /* free memory allocated by hippi_read() */
     hippi_memfree(ihandle, rx_ptr, count, 1);

     /* close connection */
     hippi_close(ihandle);
     return;

} /* hippi_server */
```

Using The HIPPI_RAW Mode

The HIPPI library provides all of the system calls needed to send and receive data using the HIPPI device. The following notes are a summary of what is necessary to write a program that uses *HIPPI_RAW, HIPPI_CNT,* or *HIPPI_MPC* modes to exchange data between nodes on a Paragon system.

HIPPI Packets

A HIPPI packet has three main parts. The first 32 bits contain the *I-field*, the next 8 bytes define the *FP header*, and the remaining bytes include any *D1_Data* and the *D2_Data*. The *raw_hippi.h* file contains structure definitions to simplify access to the various fields in the header. Always cast your data buffers to *struct hippi_header* when filling in the header fields.

hippi_open(dev_name, hippi_mode, mode)

hippi_open creates an *I-handle* that is used to access the HIPPI device. You can specify which HIPPI device to use, the mode you are using (HIPPI_RAW, HIPPI_CNT, HIPPI_MPC, or HIPPI_DATA), and the I/O mode (read, write, both...). Make sure that the permissions of the HIPPI device in */dev* are set appropriately.

```
ihandle = hippi_open("/dev/hippi", HIPPI_RAW, O_ROWR);
```

hippi_bind(ihandle, ulp, port)

hippi_bind binds the *I-handle* to a specific *ulp/port*. The *I-handle* then only receives packets that are sent to this specific *ulp/port*. This function needs to be called only if the *I-handle* is going to receive data.

hippi_config(ihandle, ifield, ulp, b, d1_data, d1_len)

This function cannot be used for a connection opened in the *HIPPI_RAW*, *HIPPI_CNT*, or *HIPPI_MPC* modes.

hippi_memget(ihandle, size)

The behavior of this function depends on which mode you are using. For an *I-handle* that was opened in *HIPPI_RAW*, *HIPPI_CNT*, or *HIPPI_MPC* modes, this function returns a pointer to the beginning of a chunk of memory *size* bytes big. Make sure that the *size* parameter is large enough to fit the *I-field*, *FP header*, *D1_Area* and *D2_Area* for the packet you will send.This pointer can then be used for **hippi_write**.

hippi_write(ihandle, ptr, length)

Using the *ihandle*, this function sends a packet *length* bytes long starting from memory location *ptr*. You must fill in the header information manually in *HIPPI_RAW*, *HIPPI_CNT*, or *HIPPI_MPC* modes. All outbound packets must contain a full *hippi_header*, which consists of the *I-field* and the *FP header*. Cast your buffer to *struct hippi_header* to access the appropriate fields. See the

raw_hippi.h file for the definition of the structures and unions. The buffer that you allocate with **hippi_memget** must be large enough to include the data, the *FP header* and the *I-field.* The *I-field* is for routing information, and the *FP header* indicates how the remaining data is to be interpreted.

I-field (32 bits)	FP Header (8 bytes)	D1_Area (0 to 1016 bytes)	D2_Area (0 to 4 Gbytes - 2)

Figure C-4. The *ptr* and *length* Arguments for hippi_write (HIPPI_RAW Mode)

The following example formats a packet and sends *n* bytes of data.

```
int       n;
char      *ptr;
struct    hippi_header *header_ptr;

/* allocate the memory */
ptr = hippi_memget(ihandle, n + sizeof(struct hippi_header));

    /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    *                             N O T E                               *
    *                                                                   *
    * hippi_memget's size parameter = bytes of data plus the header    *
    * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/* cast the buffer to a hippi_header */
header_ptr = (struct hippi_header *)ptr;

/* set the I-field, in network order */
header_ptr->cci.Ifield = htonl(0x01000001);

/* set the FP fields for w1 */
header_ptr->fp.fields.d2_off = 0;
header_ptr->fp.fields.d1_size = 0;
header_ptr->fp.fields.res = 0;
header_ptr->fp.fields.b = 0;
header_ptr->fp.fields.p = 0;
header_ptr->fp.fields.ulp = 0x80;
```

```
/* set the FP fields for w2 */
header_ptr->fp.fields.d2_size = 0; /* this field is not used by HIPPI_RAW
                                    * mode, in RAW mode it is up to the
                                    * receiving platform to interprete it */
/*
 * put the header in network order
 * Remember that the hippi_fp_header is a union and can be accessed using
 * either words or fields
 * w1 contains: d2_off, d1_size, res, b, p, and ulp
 * w2 contains: d2_size
 */
header_ptr->fp.words.w1 = htonl(header_ptr->fp.words.w1);

/*
 * copy the data into the buffer after the header, this
 * is a byte copy, if the data is integers, than extra space is needed
 */
tmp_ptr = ptr + sizeof(struct hippi_header); /* new pointer: points to data
                                              * area directly after the
                                              * header */
for(int i = 0; i < n; i++) {
 *tmp_ptr = your_data[i];
 tmp_ptr++;
}

/* now send the packet!! */
hippi_write(ihandle, ptr, n + sizeof(struct hippi_header));
```

hippi_read(ihandle, ptr, bytes_wanted)

hippi_read works the same in the *HIPPI_RAW, HIPPI_CNT, HIPPI_MPC* and *HIPPI_DATA* modes. It is used to read data from an *I-handle* (**hippi_bind** must be called before reading). The arguments to this function include the *I-handle* to read from the reference of a pointer to the received packet (which includes an FP header) and the number of bytes you expect to read. If the incoming packet is larger than the number specified by *bytes_wanted,* **hippi_read()** returns an error.If the incoming packet is equal to or smaller than the number specified by *bytes_wanted*, **hippi_read()** returns the number of bytes actually read. Note that the I-field is not include in this buffer.

To get at the data (assuming there was no *D1_Data* specified in the *FP header*) you must increment the pointer that is returned by **hippi_read** by *sizeof(union hippi_fp_header).* The length that is returned includes the bytes for the *FP header*. Therefore, the actual data (minus header) that is received starts at *ptr + sizeof(struct hippi_header)* and is *length - sizeof(struct hippi_header)* bytes long.

To get at the *FP header*, cast *ptr* to *(union hippi_fp_header)* and use the field names as defined in *raw_hippi.h.*

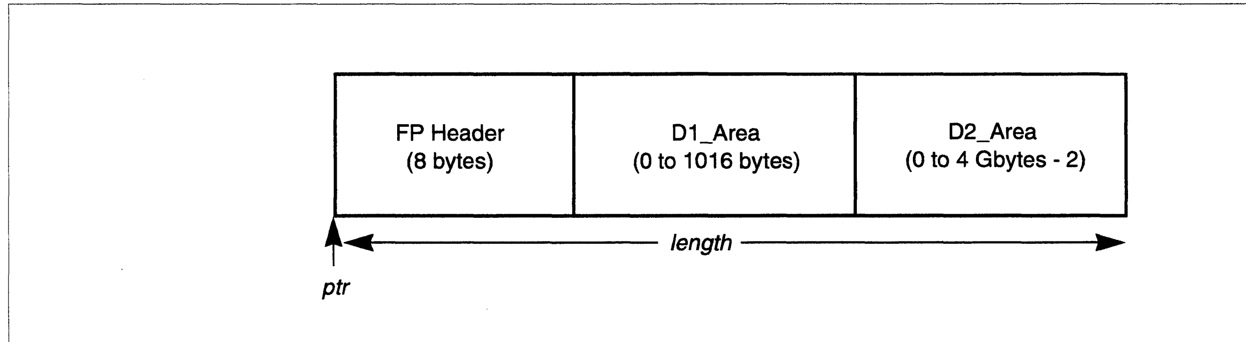

Figure C-5. The *ptr* and *length* Values Returned by hippi_read

hippi_memfree(ihandle, ptr, size, how)

hippi_memfree releases the memory allocated by **hippi_memget** or **hippi_read**. It is very important, and good style, to release the memory you use.

- Use the pointer returned by **hippi_memget**, the length passed into **hippi_memget** and *how = 0* if *ptr* was allocated by **hippi_memget**.
- Use the pointer and length returned by **hippi_read** and *how = 1* if *ptr* was from allocated by **hippi_read**.

hippi_read_request(ihandle, bytes_wanted)

hippi_read_request works the same in the *HIPPI_RAW*, *HIPPI_CNT, HIPPI_MPC* and *HIPPI_DATA* modes. It allocates buffer space of size *bytes_wanted* for a particular I-handle to provide asynchronous read capability.

You must call **hippi_read_complete** in order to retrieve the received packet. It is the responsibility of the programmer to ensure that the incoming data order and sizes match the order and sizes of the **hippi_read_request**s. For example, if you expect a particular I-handle's second packet to contain 64K of data, your second call to **hippi_read_request** should be posted (with the *bytes_wanted* parameter at least 64K) before the second **hippi_write** occurs or an error will occur.

hippi_read_complete(ihandle, ptr)

hippi_read_complete works the same in the *HIPPI_RAW, HIPPI_CNT, HIPPI_MPC* and *HIPPI_DATA* modes. It is a non-blocking call that retrieves received packets for a particular I-handle. There is a one-to-one correspondence between **hippi_read_request**s and **hippi_read_complete**s—you will get an error if you call **hippi_read_complete** when there are no **hippi_read_request**s pending. If a **hippi_read_request** is pending but the data has not arrived yet from a **hippi_write**, **hippi_read_complete** will return -1 and set *errno* to EWOULDBLOCK.

The following example is one way of using **hippi_read_complete** to continue polling until data arrives.

```
while ((count = hippi_read_complete(ihandle, &buf)) == -1)
  if(errno != EWOULDBLOCK) {
     nx_perror("hippi_read_complete() FAILED");
     exit(1);}
```

hippi_close(ihandle)

hippi_close closes the HIPPI connection. It works the same in the *HIPPI_RAW, HIPPI_CNT, HIPPI_MPC,* and *HIPPI_DATA* modes.

Code Example

There are many ways to format the buffer and allow for different types of data— *char*, *int*, *long*, *float*, etc. This example is a working program that shows where and how the pointers need to be manipulated, and illustrates the use of the HIPPI library calls to read and write data in *HIPPI_RAW* mode.

This example program does not use the *D1* data area. Also, it uses the asynchronous calls **hippi_read_request** and **hippi_read_complete** rather than the synchronous call **hippi_read** to receive the data. The program runs with a loop-back connector installed on the HIPPI node.

Compile the example using the following command:

```
# cc -nx -o hippi_node -g node.c -lhippi
```

The HIPPI interface must be set up as raw HIPPI (see page 4-14).

Create a link to your HIPPI device as follows:

```
# ln -s /dev/hippinum /dev/hippi
```

(*num* represents the HIPPI device number.)

The code is meant to run on a partition of two compute nodes, it then sends to the HIPPI node. The HIPPI node does not have to be in the compute partition. The following shows how to create the partition and run the example, and illustrates a typical output:

```
# mkpart -sz 2 hippitest
# ./hippi_node -pn hippitest
Node 0: HIPPI Data Exchange: STARTED
Node 0: Loopback Mode
Node 0: CLIENT - Ifield = 0x1000001, ULP = 0x80, port = -1
Node 1: HIPPI Data Exchange: STARTED
Node 1: Loopback Mode
Node 0: SERVER - Ifield = 0x1000001, ULP = 0x81, port = -1
Node 1: HIPPI_SERVER started
Node 1: HIPPI_SERVER reading data
Node 0: HIPPI_CLIENT started
Node 0: HIPPI_CLIENT writing data
Node 0: HIPPI_CLIENT reading data
Node 1: HIPPI_SERVER writing data
Node 1: check_buf(): 0 mismatches
Node 0: HIPPI_CLIENT: 2679856 bytes in 0.02 s = 127776823.59 bytes/sec
Node 0: check_buf(): 0 mismatches
Node 1: check_buf(): 0 mismatches
Node 0: check_buf(): 0 mismatches
Node 1: check_buf(): 0 mismatches
Node 0: check_buf(): 0 mismatches
Node 1: check_buf(): 0 mismatches
Node 0: check_buf(): 0 mismatches
Node 1: HIPPI Data Exchange Done: *** PASSED ***
Node 0: HIPPI Data Exchange Done: *** PASSED ***
#
```

The example code that follows is located in the */usr/share/examples/c/hippi* directory on the Paragon system.

```
/**************************************************************************
 *
 * Title: hippi/exchange/node.c
 *
 * Description:
 * Using the HIPPI interface, exchange data between nodes. Node 0 is
 * the "client" Node 1 is the "server."
 * Each node creates four buffers for writing. Each buffer
 * is a different length (defined by the Bx_SIZE constants), formated
 * to be sent to the other node, and filled with filler data.
 * The client writes its buffers to the server and expects to receive
 * four packets from the server. The server receives the packets from
 * the client and writes its buffers to the client. After all data is
 * sent and received, both nodes compare the received data with the
 * sent data and report any discrepancies. The client also reports
 * the rate of the data exchange.
 *
 *
 **************************************************************************/

#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <setjmp.h>
#include <sys/param.h>
#include <sys/timers.h>
#include <raw_hippi.h>

#define TIMEOUT_VAL 60

/* buffer sizes: must have 8 as a factor so that there are no partial words */
#define B1_SIZE 1000
#define B2_SIZE 88888
#define B3_SIZE 250016
#define B4_SIZE 1000024

/* global vars */
int failed = 0;
int node;
int tcp_debug;

jmp_buf timeout;

u_long cci_client = 0x01000001;
```

```
u_long cci_server = 0x01000001;
u_long ulp_client = 0x80;
u_long ulp_server = 0x81;
u_long port_client = -1;
u_long port_server = -1;


/**************************************************************************
 *
 * main(): gets command line arguments, calls either client or
 * server function based on node number, and reports results
 *
 **************************************************************************/

main (argc, argv)
int argc;
char *argv[];
{

     tcp_debug = 1;
     node = mynode();

     printf("Node %d: HIPPI Data Exchange: STARTED\n", node);
     fflush(stdout);

     switch(argc) {
          case 1:
               printf("Node %d: Loopback Mode\n", node);
               fflush(stdout);
               break;

          case 7:
               cci_client = atol(argv[1]);
               ulp_client = atol(argv[2]);
               port_client = atol(argv[3]);
               cci_server = atol(argv[4]);
               ulp_server = atol(argv[5]);
               port_server = atol(argv[6]);
               break;

          default:
               printf("Node %d: Usage - %s [client_ifield client_ulp client_port
server_ifield server_ulp server_port]\n", node, argv[0]);
               fflush(stdout);
               exit(1);
     }

     /*
```

```
     * Do the tests
     * Node 0 is the client
     * Node 1 is the server that echos the data sent by the client
     */
    if(node == 0) {
        printf("Node %d: CLIENT - Ifield = 0x%x, ULP = 0x%x, port = %d\n",
node, cci_client, ulp_client, port_client);
        fflush(stdout);
        printf("Node %d: SERVER - Ifield = 0x%x, ULP = 0x%x, port = %d\n",
node, cci_server, ulp_server, port_server);
        fflush(stdout);
        sleep(5);
        hippi_client();}
    else if(node == 1) {
        hippi_server();}
    else {
        printf("Node %d: Node Not Needed - Good Bye!\n", node);
        fflush(stdout);
        exit(0);}

    /*
     * Report test results
     */
    if (failed == 0) {
        printf("Node %d: HIPPI Data Exchange Done: *** PASSED ***\n", node);
        fflush(stdout);}
    else {
        printf("Node %d: HIPPI Data Exchange Done: *** FAILED ***\n", node);
        fflush(stdout);}

    exit(failed);} /* main */


    /***********************************************************************
     *
     * alrm_handler()
     *
     * Prevents infinite blocking reads from hanging program. Does a
     * longjmp()if called. Attached to signal SIGALRM
     *
     ***********************************************************************/

    void alrm_handler(){
        if(tcp_debug) {
            printf("Node %d: SIGALRM occured!\n", node);
            fflush(stdout);}

        longjmp(timeout, 1);}
```

```
/***********************************************************************
 *
 * get_buf(ihandle, size, header_ptr, fill_char)
 *
 *   ihandle: specifies the HIPPI connection
 *   size: the size of the desired buffer including space for
 *        the header
 *   head_ptr: a pointer to the header structure that will be
 *            copied into this buffer
 *   fill_char: the character to fill the remaining space in the buffer
 *
 * Returns a pointer to a buffer of `size' bytes, formatted with the header
 * and fill character.
 *
 ***********************************************************************/

char *get_buf(ihandle, size, header_ptr, fill_char)
int ihandle;
u_long size;
struct hippi_header *header_ptr;
char fill_char;

{
char *ptr;

/* allocate memory */
if((ptr = hippi_memget(ihandle, size)) == NULL) {
     nx_perror("get_buf(): FAILED hippi_memget()");}
else { /* copy header into buffer */
     memcpy(ptr, header_ptr, sizeof(struct hippi_header));

     /* fill rest of buffer */
     memset(ptr + sizeof(struct hippi_header), fill_char,
     size - sizeof(struct hippi_header));}

/* return pointer to buffer */
return(ptr);}


/***********************************************************************
 * check_buf(buf1, buf2, ulp, size)
 *
 * Compare the data in HIPPI frame buf1 (a frame that was written) to buf2
 * (a frame that was read). buf2 is checked for the correct ULP and size.
 * Return 0 if everything matches, otherwise return the number of
 * discrepancies.
```

```
 *
 * Remember that a buffer formatted for writing includes the i-field while
 * a buffer returned from a hippi_read() does not contain an i-field. Also
 * headers are in "network" order and need to converted using the
 * ntoh*() functions. The Paragon bit ordering matches network order,
 * but good style and the demands of portable code dictate their use.
 * See raw_hippi.h for the definition of the header structures.
 *
 *********************************************************************/

int check_buf(buf_w, buf_r, ulp, size_w, size_r)
char *buf_w;
char *buf_r;
u_long ulp;
u_long size_w;
u_long size_r;

{
char *ptr_w, *ptr_r;
int i, bad_cnt = 0;
union hippi_fp_header *fp_ptr;

/* convert fp header to host order */
fp_ptr = (union hippi_fp_header *)buf_r;
fp_ptr->words.w1 = ntohl(fp_ptr->words.w1);

/* check size, read buffer does not contain the I-field */
if(size_r != (size_w - sizeof(struct hippi_ifield))) {
     printf("Node %d: check_buf(): FAILED wrong number of bytes read\n",
node);
     fflush(stdout);
     bad_cnt++;}

/* check ULP of buf_r */
if(fp_ptr->fields.ulp != ulp) {
     printf("Node %d: check_buf(): FAILED ULP wrong, 0x%x != 0x%x\n",
node, fp_ptr->fields.ulp, ulp);
     fflush(stdout);
     bad_cnt++;}

/*
 * check data
 *
 * This check compares the read buffer to the write buffer. A read
 * buffer only has an FP header while the write buffer has both the I-field
 * and an FP header. Therefore, to get at the data in the read buffer, you
 * must skip sizeof(union hippi_fp_header) bytes and to get to the
```

```
 * data in the write buffer you must skip sizeof(struct hippi_header) bytes
 */
ptr_w = buf_w + sizeof(struct hippi_header);
ptr_r = buf_r + sizeof(union hippi_fp_header);

for(i = 0; i < (size_w - sizeof(struct hippi_header)); i++) {
     if(*ptr_w != *ptr_r) {
          bad_cnt++;}
     ptr_w++;
     ptr_r++;}

/* restore fp header to network order, non-destructive compare */
fp_ptr->words.w1 = htonl(fp_ptr->words.w1);

if(tcp_debug) {
     printf("Node %d: check_buf(): %d mismatches\n", node, bad_cnt);
          fflush(stdout);}

return(bad_cnt);}


/*********************************************************************
 *
 * hippi_client()
 *
 * Fill in a HIPPI header with values to send data to the server.
 * Open the HIPPI device. Use get_buf() to create some buffers. Write
 * the buffers to the server, read what the server sends back, and check
 * for discrepancies. Report any errors and the data rate. Also, free
 * the memory allocated for for the device.
 *
 ********************************************************************/

hippi_client()
{
int ihandle, total_bytes;
long buf1_r_len, buf2_r_len, buf3_r_len, buf4_r_len;
double elap_time;
char *buf1_w, *buf2_w, *buf3_w, *buf4_w;
char *buf1_r, *buf2_r, *buf3_r, *buf4_r;

/*
 * See hippi_raw.h for the definition of a hippi_header. It contains two
 * other structures, an I-field and a FP_Header.
 */
struct hippi_header hh_w;
struct timespec t1, t2;
```

```
printf("Node %d: HIPPI_CLIENT started\n", node);
fflush(stdout);

/*
 * header for writes to server
 *
 * Note: zeroing this structure zeros the P, B, D1_Area_Size,
 * and the D2_Offset fields of the FP_HEADER_AREA. This turns "off"
 * the use of the D1_Area and makes the D2_Area not aligned.
 * This simplies things and avoids any offset computation.
 * Remember that this is "RAW" mode that allows you to do anything!
 *
 * Also, the header must be in "network" order--use the hton*()
 * functions to avoid problems.
 */
memset(&hh_w, 0, sizeof(hh_w));

/*
 * The I-field for outbound packets contains the routing information.
 * This is the destination of the outbound data.  The hippi.map file
 * (or equivalent) contains the I-fields for other hosts connected to
 * the HIPPI network
 */
hh_w.cci.Ifield = htonl(cci_server);

/*
 * Since the D1_Area is not being used, the only value that needs to
 * be put into the FP_Header is the destination ULP.
 */
hh_w.fp.fields.ulp = ulp_server;
hh_w.fp.words.w1 = htonl(hh_w.fp.words.w1);

 /* create HIPPI connection */
if((ihandle = hippi_open("/dev/hippi", HIPPI_RAW, O_RDWR)) < 0) {
    nx_perror("HIPPI_CLIENT: FAILED hippi_open()");
    failed++;}

/*
 * bind for reads from server
 *
 * To receive data the I-handle must be bound to a specific ULP. In this
 * case, the server is sending data to the client. Therefore the client
 * must bind its ULP to the I-handle. This I-handle will then be able to
 * to receive data that is sent to this ULP.
 */
if(hippi_bind(ihandle, ulp_client, port_client) == -1) {
```

```
        nx_perror("HIPPI_CLIENT: FAILED hippi_bind()");
        failed++;}

    /* Initialize buffers */
    if((buf1_w =get_buf(ihandle, B1_SIZE, &hh_w, 'f')) == NULL) {
        printf("Node %d: HIPPI_CLIENT: FAILED get_buf(B1_SIZE)\n", node);
        fflush(stdout);
        failed++;}

    if((buf2_w =get_buf(ihandle, B2_SIZE, &hh_w, 'A')) == NULL) {
        printf("Node %d: HIPPI_CLIENT: FAILED get_buf(B2_SIZE)\n", node);
        fflush(stdout);
        failed++;}

    if((buf3_w =get_buf(ihandle, B3_SIZE, &hh_w, 's')) == NULL) {
        printf("Node %d: HIPPI_CLIENT: FAILED get_buf(B3_SIZE)\n", node);
        fflush(stdout);
        failed++;}

    if((buf4_w =get_buf(ihandle, B4_SIZE, &hh_w, 'T')) == NULL) {
        printf("Node %d: HIPPI_CLIENT: FAILED get_buf(B4_SIZE)\n", node);
        fflush(stdout);
        failed++;}

    if(tcp_debug) {
        printf("Node %d: HIPPI_CLIENT writing data\n", node);
        fflush(stdout);}

    /* Allocate buffer space for receiving data from client */
    if(hippi_read_request(ihandle, B1_SIZE) < 0) {
        nx_perror("HIPPI_CLIENT: FAILED hippi_read_request(buf1)");
        exit(1);}

    if(hippi_read_request(ihandle, B2_SIZE) < 0) {
        nx_perror("HIPPI_CLIENT: FAILED hippi_read_request(buf2)");
        exit(1);}

    if(hippi_read_request(ihandle, B3_SIZE) < 0) {
        nx_perror("HIPPI_CLIENT: FAILED hippi_read_request(buf3)");
        exit(1);}

    if(hippi_read_request(ihandle, B4_SIZE) < 0) {
        nx_perror("HIPPI_CLIENT: FAILED hippi_read_request(buf4)");
        exit(1);}
```

```
if(setjmp(timeout) == 0) {
    /* attach signal to alarm handler */
    signal(SIGALRM, alrm_handler);
    alarm(TIMEOUT_VAL);
    getclock(TIMEOFDAY, &t1);

    /* write data */
    if(hippi_write(ihandle, buf1_w, B1_SIZE) == -1) {
        nx_perror("HIPPI_CLIENT: FAILED hippi_write(buf1)");
        failed++;}

    if(hippi_write(ihandle, buf2_w, B2_SIZE) == -1) {
        nx_perror("HIPPI_CLIENT: FAILED hippi_write(buf2)");
        failed++;}

    if(hippi_write(ihandle, buf3_w, B3_SIZE) == -1) {
        nx_perror("HIPPI_CLIENT: FAILED hippi_write(buf3)");
        failed++;}

    if(hippi_write(ihandle, buf4_w, B4_SIZE) == -1) {
        nx_perror("HIPPI_CLIENT: FAILED hippi_write(buf4)");
        failed++;}

    if(tcp_debug) {
        printf("Node %d: HIPPI_CLIENT reading data\n", node);
        fflush(stdout);}

    /* read data */
    while((buf1_r_len = hippi_read_complete(ihandle, &buf1_r)) == -1)
    if(errno != EWOULDBLOCK) {
         nx_perror("HIPPI_CLIENT: FAILED hippi_read(buf1)");
         exit(1);}

    while((buf2_r_len = hippi_read_complete(ihandle, &buf2_r)) == -1)
        if(errno != EWOULDBLOCK) {
            nx_perror("HIPPI_CLIENT: FAILED hippi_read(buf2)");
            exit(1);}

    while((buf3_r_len = hippi_read_complete(ihandle, &buf3_r)) == -1)
        if(errno != EWOULDBLOCK) {
            nx_perror("HIPPI_CLIENT: FAILED hippi_read(buf3)");
            exit(1);}

    while((buf4_r_len = hippi_read_complete(ihandle, &buf4_r)) == -1)
        if(errno != EWOULDBLOCK) {
            nx_perror("HIPPI_CLIENT: FAILED hippi_read(buf4)");
            exit(1);}
```

```
        getclock(TIMEOFDAY, &t2);
        alarm(0);

        /* report bytes per second */
        elap_time = t2.tv_sec+t2.tv_nsec/10.0e9 - t1.tv_sec-t1.tv_nsec/10.0e9;
        total_bytes = 2*(B1_SIZE + B2_SIZE + B3_SIZE + B4_SIZE);

        printf("Node %d: HIPPI_CLIENT: %d bytes in %.2f s = %.2f bytes/sec\n",
 node, total_bytes, elap_time, total_bytes/elap_time);
        fflush(stdout);

        /* Check received data */
        if(check_buf(buf1_w, buf1_r, ulp_client, B1_SIZE, buf1_r_len)) {
            printf("Node %d: HIPPI_CLIENT: FAILED buf1 bad data\n", node);
            fflush(stdout);
            failed++;}

        if(check_buf(buf2_w, buf2_r, ulp_client, B2_SIZE, buf2_r_len)) {
            printf("Node %d: HIPPI_CLIENT: FAILED buf2 bad data\n", node);
            fflush(stdout);
            failed++;}

        if(check_buf(buf3_w, buf3_r, ulp_client, B3_SIZE, buf3_r_len)) {
            printf("Node %d: HIPPI_CLIENT: FAILED buf3 bad data\n", node);
            fflush(stdout);
            failed++;}

        if(check_buf(buf4_w, buf4_r, ulp_client, B4_SIZE, buf4_r_len)) {
            printf("Node %d: HIPPI_CLIENT: FAILED buf4 bad data\n", node);
            fflush(stdout);
            failed++;}
        }
    else {
        printf("Node %d: HIPPI_CLIENT: FAILED write or read timed out\n",
node);
        fflush(stdout);
        failed++;}

    /*
     * free memory and close connection
     *
     * See the man page for hippi_memfree() about the last parameter
     */
    hippi_memfree(ihandle, buf1_w, B1_SIZE, 0);
    hippi_memfree(ihandle, buf2_w, B2_SIZE, 0);
```

```
hippi_memfree(ihandle, buf3_w, B3_SIZE, 0);
hippi_memfree(ihandle, buf4_w, B4_SIZE, 0);
hippi_memfree(ihandle, buf1_r, buf1_r_len, 1);
hippi_memfree(ihandle, buf2_r, buf2_r_len, 1);
hippi_memfree(ihandle, buf3_r, buf3_r_len, 1);
hippi_memfree(ihandle, buf4_r, buf4_r_len, 1);
hippi_close(ihandle);
return;
} /* hippi_client */

/**********************************************************************
 *
 * hippi_server()
 *
 * Fill in a HIPPI header with values to send data to the client.
 * Open the HIPPI device. Use get_buf() to create some buffers. Read what
 * the client sends, write the buffers to the client and check
 * for discrepancies. Report any errors and free the memory allocated
 * for for the device.
 *
 *********************************************************************/

hippi_server()
{
int ihandle;
long   buf1_r_len, buf2_r_len, buf3_r_len, buf4_r_len;
char *buf1_w, *buf2_w, *buf3_w, *buf4_w;
char *buf1_r, *buf2_r, *buf3_r, *buf4_r;
struct hippi_header hh_w;

printf("Node %d: HIPPI_SERVER started\n", node);
fflush(stdout);

/*
 * header for writes to client
 *
 * See the comments in hippi_client() about filling in the HIPPI
 * header fields. For the server, the destination ULP is the client
 */
memset(&hh_w, 0, sizeof(hh_w));
hh_w.cci.Ifield = htonl(cci_client);
hh_w.fp.fields.ulp = ulp_client;
hh_w.fp.words.w1 = htonl(hh_w.fp.words.w1);

/* create HIPPI connection */
if((ihandle = hippi_open("/dev/hippi", HIPPI_RAW, O_RDWR)) < 0) {
     nx_perror("HIPPI_SERVER: FAILED hippi_open()");
     failed++;}
```

```
/*
 * bind for reads from client to receive data sent to the server ULP
 *
 * See the comments in hippi_client() about binding I-handles
 */
if(hippi_bind(ihandle, ulp_server, port_server) == -1) {
     nx_perror("HIPPI_SERVER: FAILED hippi_bind()");
     failed++;}

/* Initialize buffers */
if((buf1_w =get_buf(ihandle, B1_SIZE, &hh_w, 'f')) == NULL) {
     printf("Node %d: HIPPI_SERVER: FAILED get_buf(B1_SIZE)\n", node);
     fflush(stdout);
     failed++;}

if((buf2_w =get_buf(ihandle, B2_SIZE, &hh_w, 'A')) == NULL) {
     printf("Node %d: HIPPI_SERVER: FAILED get_buf(B2_SIZE)\n", node);
     fflush(stdout);
     failed++;}

if((buf3_w =get_buf(ihandle, B3_SIZE, &hh_w, 's')) == NULL) {
     printf("Node %d: HIPPI_SERVER: FAILED get_buf(B3_SIZE)\n", node);
     fflush(stdout);
     failed++;}

if((buf4_w =get_buf(ihandle, B4_SIZE, &hh_w, 'T')) == NULL) {
     printf("Node %d: HIPPI_SERVER: FAILED get_buf(B4_SIZE)\n", node);
     fflush(stdout);
     failed++;}

if(tcp_debug) {
     printf("Node %d: HIPPI_SERVER reading data\n", node);
     fflush(stdout);}

/* Allocate buffer space for receiving data from client */
if(hippi_read_request(ihandle, B1_SIZE) < 0) {
     nx_perror("HIPPI_SERVER: FAILED hippi_read_request(buf1)");
     exit(1);}

if(hippi_read_request(ihandle, B2_SIZE) < 0) {
     nx_perror("HIPPI_SERVER: FAILED hippi_read_request(buf2)");
     exit(1);}

if(hippi_read_request(ihandle, B3_SIZE) < 0) {
     nx_perror("HIPPI_SERVER: FAILED hippi_read_request(buf3)");
     exit(1);}
```

```
if(hippi_read_request(ihandle, B4_SIZE) < 0) {
     nx_perror("HIPPI_SERVER: FAILED hippi_read_request(buf4)");
     exit(1);}

if(setjmp(timeout) == 0) {
     /* attach signal to alarm handler */
     signal(SIGALRM, alrm_handler);

     alarm(TIMEOUT_VAL);

     /* read data */
     while((buf1_r_len = hippi_read_complete(ihandle, &buf1_r)) == -1)
     if(errno != EWOULDBLOCK) {
           nx_perror("HIPPI_SERVER: FAILED hippi_read(buf1)");
           exit(1);}

     while((buf2_r_len = hippi_read_complete(ihandle, &buf2_r)) == -1)
          if(errno != EWOULDBLOCK) {
               nx_perror("HIPPI_SERVER: FAILED hippi_read(buf2)");
               exit(1);}

     while((buf3_r_len = hippi_read_complete(ihandle, &buf3_r)) == -1)
          if(errno != EWOULDBLOCK) {
               nx_perror("HIPPI_SERVER: FAILED hippi_read(buf3)");
               exit(1);}

     while((buf4_r_len = hippi_read_complete(ihandle, &buf4_r)) == -1)
          if(errno != EWOULDBLOCK) {
               nx_perror("HIPPI_SERVER: FAILED hippi_read(buf4)");
               exit(1);}

     if(tcp_debug) {
          printf("Node %d: HIPPI_SERVER writing data\n", node);
          fflush(stdout);}

     /* write data */
     if(hippi_write(ihandle, buf1_w, B1_SIZE) == -1) {
          nx_perror("HIPPI_SERVER: FAILED hippi_write(buf1)");
          failed++;}

     if(hippi_write(ihandle, buf2_w, B2_SIZE) == -1) {
          nx_perror("HIPPI_SERVER: FAILED hippi_write(buf2)");
          failed++;}

     if(hippi_write(ihandle, buf3_w, B3_SIZE) == -1) {
          nx_perror("HIPPI_SERVER: FAILED hippi_write(buf3)");
          failed++;}
```

```
        if(hippi_write(ihandle, buf4_w, B4_SIZE) == -1) {
            nx_perror("HIPPI_SERVER: FAILED hippi_write(buf4)");
            failed++;}

        alarm(0);

        /* Check received data */
        if(check_buf(buf1_w, buf1_r, ulp_server, B1_SIZE, buf1_r_len)) {
            printf("Node %d: HIPPI_SERVER: FAILED buf1 bad data\n", node);
            fflush(stdout);
            failed++;}

        if(check_buf(buf2_w, buf2_r, ulp_server, B2_SIZE, buf2_r_len)) {
            printf("Node %d: HIPPI_SERVER: FAILED buf2 bad data\n", node);
            fflush(stdout);
            failed++;}

        if(check_buf(buf3_w, buf3_r, ulp_server, B3_SIZE, buf3_r_len)) {
            printf("Node %d: HIPPI_SERVER: FAILED buf3 bad data\n", node);
            fflush(stdout);
            failed++;}

        if(check_buf(buf4_w, buf4_r, ulp_server, B4_SIZE, buf4_r_len)) {
            printf("Node %d: HIPPI_SERVER: FAILED buf4 bad data\n", node);
            fflush(stdout);
            failed++;}

        }
    else {
        printf("Node %d: HIPPI_SERVER: FAILED write or read timed out\n",
node);
        fflush(stdout);
        failed++;}

        /* free memory and close connection */
        hippi_memfree(ihandle, buf1_w, B1_SIZE, 0);
        hippi_memfree(ihandle, buf2_w, B2_SIZE, 0);
        hippi_memfree(ihandle, buf3_w, B3_SIZE, 0);
        hippi_memfree(ihandle, buf4_w, B4_SIZE, 0);
        hippi_memfree(ihandle, buf1_r, buf1_r_len, 1);
        hippi_memfree(ihandle, buf2_r, buf2_r_len, 1);
        hippi_memfree(ihandle, buf3_r, buf3_r_len, 1);
        hippi_memfree(ihandle, buf4_r, buf4_r_len, 1);
        hippi_close(ihandle);
        return;
        } /* hippi_server */
```

Using IPI Devices With Paragon™ Systems

D

Using The IPI-3 Interface on Paragon™ Systems

This section describes how to use the HIPPI IPI-3 interface on Paragon systems. It includes information about IPI protocol, system requirements, addressing, and information about configuring the HIPPI interface for use with IPI-3 transfers.

The IPI Protocol

The Intelligent Peripheral Interface (IPI) is a protocol developed by the International Standards Organization (ISO) and the International Electrotechnical Commission (IEC). The specification is in four parts:

Part 1	Physical Interface.
Part 2	Device-specific command set for magnetic disk drives.
Part 3	Device-generic command set for magnetic and optical disk drives.
Part 4	Device-generic command set for magnetic disk drives.

The Part 3 interface (IPI-3) drivers are available to connect IPI devices to Paragon™ systems via the HIPPI interface. Data is transferred between the HIPPI master and IPI-3 slave devices using "read" and "write" commands like any disk drive.

Using the IPI-3 Interface

The IPI-3 subsystem provides a high-speed interface for transferring large amounts of data between a Paragon system and a remote IPI-3 mass-storage device (such as an external magnetic/optical disk farm or a tape unit) via a HIPPI connection. The HIPPI IPI-3 driver provides three types of transfers:

Raw Block I/O	Provides for reading and writing directly to the IPI-3 device without going through the file system. Raw block I/O to an IPI-3 magnetic disk array provides high-speed remote temporary storage for moving large data sets (using block sizes as large as 4 MB) into and out of a Paragon system without going through a file system.
File System I/O	Provides for communicating with the IPI-3 device through the file system, just as if it were part of the local disk arrays. File system I/O is only available for parallel file systems (PFS) and not for Unix file systems (UFS). File system I/O to a mounted magnetic disk provides high-speed remote-access file systems that can be configured with block sizes as large as 512 KB.

System Setup Overview

A system administrator must perform the following steps to set up the IPI-3 interface on a Paragon™ system. The steps are described in detail in this appendix:

1. Create new bootmagic variables to configure the IPI-3 driver to a HIPPI channel *or* design a series of **ioctl()** function calls within your application to configure the interface.

2. Reboot the Paragon system to enable the variables in the *bootmagic* file, if you are using that method to configure the interface.

3. Create device special files in */dev*.

4. Create a disk label in */etc/disktab* (if necessary).

5. Write the label to the IPI device.

The process is complete at this point if you are configuring the system for raw I/O transfers. The remaining steps are necessary for using the system with file system I/O.

6. Create new file systems on the IPI device.

7. Mount the new file systems for use by the Paragon system.

Device Requirements

IPI devices must meet the following requirements to be used with a Paragon system:

- It must be a magnetic disk using IPI-3 protocol over HIPPI channels.

- It must use a block size of 64k bytes.

- It is *desirable* that it use Full First Burst mode for better performance.

IPI Addressing

An IPI-3 device is addressed by four components:

Slave	Up to eight slaves may be assigned to each HIPPI master.
Facility	Up to 16 physical facilities are available per slave.
Partition	There may be up to 256 partitions per facility. (Note that this is not the same as a Unix partition.)
Unix Partition	Up to 16 Unix partitions are available on each facility partition.

Setting Up An IPI-3 Interface on Paragon™ Systems

Connecting the IPI-3 Interface to a HIPPI Channel

The IPI-3 interface connections must be specified, using either bootmagic variables in the *MAGIC.MASTER* file to initialize the connections during system start-up, or by using **ioctl()** function calls within your application to specify or change the connections. Both processes are described in the following paragraphs.

Using Bootmagic Variables

The IPI-3 interface is assigned to a HIPPI channel by setting variables in the *MAGIC.MASTER* file on the diagnostic station. That file is used when a Paragon system boots to create the *bootmagic* file, which is downloaded to the supercomputer and used to configure hardware and software.

NOTE

Never modify the *bootmagic* file directly.

Bootmagic variables allow you to control the operation of the HIPPI IPI device driver. These configuration variables operate on individual nodes, allowing multiple IPI interfaces to be configured differently within a Paragon™ system.

The syntax of a bootmagic string is:

name=[<*node_list*>]*value*[:<*node_list*>*value*]...

name — The name of the bootmagic variable.

node_list — The node_list may have the following form:

node[,*node*]...

node — Either an individual node ID or a range of nodes (*low_node..high_node*).

If *node_list* is not specified, the bootmagic variable setting applies to all nodes in the Paragon™ system.

value — The initialization value for the bootmagic variable.

Refer to the *Paragon™ System Software Release Notes* for more information about the bootmagic variables.

Setting IPI Slave Connection Control Variables

The HIPPI interface defines a protocol for controlling physical layer switches. This protocol includes a parameter called an I-field to establish a connection from a source to a destination. A bootmagic variable that contains the connection control information may be defined for each IPI slave device. The available bootmagic variables are:

IPI_SLAVE_0_IFIELD
IPI_SLAVE_1_IFIELD
IPI_SLAVE_2_IFIELD
IPI_SLAVE_3_IFIELD
IPI_SLAVE_4_IFIELD
IPI_SLAVE_5_IFIELD
IPI_SLAVE_6_IFIELD
IPI_SLAVE_7_IFIELD

If not defined, the driver defaults to a value of the slave ID. The following example specifies that the I-field routing control value for the Slave 0 device will be “8”:

```
IPI_SLAVE_0_IFIELD=8
```

Refer to Chapter 4 for more information about the I-field and routing through the HIPPI network.

Setting IPI Slave Facility Variables

Each IPI device can support multiple facilities. These facilities are unique to each type of hardware and allow a single IPI slave to control multiple sub-devices. While there are several ways in which a facility address may logically be used by an IPI slave, a maximum of sixteen physical facilities may exist. To overcome this logical versus physical device limitation, a set of *bootmagic* variables are available to define the base facility number that is added to the facility number offset encoded in the device minor number. The *bootmagic* variables are:

IPI_SLAVE_0_FACILITY
IPI_SLAVE_1_FACILITY
IPI_SLAVE_2_FACILITY
IPI_SLAVE_3_FACILITY
IPI_SLAVE_4_FACILITY
IPI_SLAVE_5_FACILITY
IPI_SLAVE_6_FACILITY
IPI_SLAVE_7_FACILITY

If not defined, the driver defaults to a facility base value of 0. The following example sets the base facility number for the Slave 0 device to "16":

```
IPI_SLAVE_0_FACILITY=16
```

Setting IPI Slave Partitions

IPI devices can support multiple partitions within each facility. These partitions are controlled by the IPI-3 device and are not related to the Unix file system partitions encoded in the minor number. Multiple partition support allows you to take advantage of the performance characteristics of each partition depending on the device configuration and usage model. The bootmagic variables are:

IPI_SLAVE_0_PARTITION
IPI_SLAVE_1_PARTITION
IPI_SLAVE_2_PARTITION
IPI_SLAVE_3_PARTITION
IPI_SLAVE_4_PARTITION
IPI_SLAVE_5_PARTITION
IPI_SLAVE_6_PARTITION
IPI_SLAVE_7_PARTITION

If not defined, the driver will default to partition 0, the default data partition of the device. The following example specifies that the Slave 0 device will use partition 16

```
IPI_SLAVE_0_PARTITION=16
```

Setting IPI Command Reference Numbers

IPI commands include a command reference number field that identifies each individual command. The slave echoes the command reference number in a response packet to identify the associated command. For slaves capable of queuing multiple commands, the master is responsible for ensuring that all active commands have a unique identification. When the command is no longer outstanding, the master may reuse the command reference number. Because a slave may be shared by several masters, such as when two systems are sharing different partitions on the same disk array, a unique range of command reference numbers must be assigned to each master. The bootmagic variable pairs used are:

IPI_SLAVE_0_CMD_REF_MIN	IPI_SLAVE_0_CMD_REF_MAX
IPI_SLAVE_1_CMD_REF_MIN	IPI_SLAVE_1_CMD_REF_MAX
IPI_SLAVE_2_CMD_REF_MIN	IPI_SLAVE_2_CMD_REF_MAX
IPI_SLAVE_3_CMD_REF_MIN	IPI_SLAVE_3_CMD_REF_MAX
IPI_SLAVE_4_CMD_REF_MIN	IPI_SLAVE_4_CMD_REF_MAX
IPI_SLAVE_5_CMD_REF_MIN	IPI_SLAVE_5_CMD_REF_MAX
IPI_SLAVE_6_CMD_REF_MIN	IPI_SLAVE_6_CMD_REF_MAX
IPI_SLAVE_7_CMD_REF_MIN	IPI_SLAVE_7_CMD_REF_MAX

If not defined, the driver defaults to command reference numbers 0x0000 through 0x000F inclusive. To provide compatibility with NSL client implementations on the Paragon™ system, the maximum command reference number is limited to 0x7FFF. The following example defines the range of command reference numbers that will apply to Slave 0 as those between 16 and 31.

```
IPI_SLAVE_0_CMD_REF_MIN=16
IPI_SLAVE_0_CMD_REF_MAX=31
```

Note that the range of command reference numbers specified limits the amount of command queuing the HIPPI IPI device driver performs on each slave and the amount of memory allocated for command queues. Each command buffer allocated consumes 1024 bytes of wired-down node memory.

NOTE

If you are using bootmagic variables to configure the IPI-3 interface, you must reboot the Paragon system at this point for the *bootmagic* variables to take effect.

Dynamic Control of IPI-3 Connections With IOCTL Functions

The IPI control functions may either be set with *bootmagic* variables as described previously, or changed dynamically with **ioctl()** functions within the application code. The **ioctl()** functions may be used to set or read information about a particular slave device.

The **ioctl()** functions are defined in the header file */usr/include/device/ipi_status.h.* Refer to the online manual page for **ioctl()** for general information about using those functions, and to the *ipi_status.h* file for detailed information about the arguments for each function.

The following code segment contains examples of how to configure an IPI-3 interface using **ioctl()** functions:

```
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <device/ipi_status.h>

#define IPI_MASTER_DEV "/dev/ipi/ripm0"

main()
{
   int fd;      /* file descriptor of an IPI Master Device */
   unsigned int ifield;    /* ifield */
   unsigned int facility;  /* facility */
   unsigned int partition; /* partition */
   struct ipi_cmd_ref cmd_ref; /* min & max command */
                               /* reference numbers */

   if ((fd = open(IPI_MASTER_DEV, O_RDONLY)) < 0) {
      perror("open failed");
      exit(1);
   }

   ifield = 8;
   if (ioctl(fd, IPISIFIELD, &ifield) < 0) {
      perror("ioctl() IPISIFIELD failed\n");
      exit(1);
   }

   facility = 16;
   if (ioctl(fd, IPISFACILITY, &facility) < 0) {
      perror("ioctl() IPISFACILITY failed\n");
      exit(1);
   }
```

```
   partition = 16;
   if (ioctl(fd, IPISPART, &partition) < 0) {
      perror("ioctl() IPISPART failed\n");
      exit(1);
   }

   cmd_ref.min = 16;
   cmd_ref.max = 31;
   if (ioctl(fd, IPISCMDREF, &cmd_ref) < 0) {
      perror("ioctl() IPISCMDREF failed\n");
      exit(1);
   }

   close(fd);
} exit(0);
```

Creating IPI-3 Device Entries

After configuring the IPI-3 interface, create device entries in the */dev* directory using the **rmknod** command. The syntax of the **rmknod** command is:

rmknod *<name>* <**b**|**c**> *<major>* *<minor>* *<node>*

name	The name of the device being created.
b	Defines a block-oriented device.
c	Defines a character-oriented device.

major The major number specifies the device type and slave. The available device types are:

Master Device: Character devices only. The master device is an optional control interface for the IPI-3 device driver and does not use the Unix partition defined in the minor number.

Master devices are specified with the major number 25, and offset by the number of the slave (0 - 7). Slave 0 is 25, Slave 1 is 26, etc.

Magnetic Disk: Either block or raw character devices.

Block disk devices are specified with the major number 23, and offset by the number of a slave (0 - 7). Slave 0 is 23, Slave 1 is 24, etc.

Raw character disks are specified with the major number 33, and offset by the number of a slave (0 - 7). Slave 0 is 33, Slave 1 is 34, etc.

NOTE

If you are using an application to configure the device (as opposed to *bootmagic* variables) you must open the master device to perform the configuration. You cannot open a magnetic disk if it has not been configured. Once they are properly configured, either a master device or a magnetic disk may be used to modify the configuration, with some restrictions.

minor The minor number is an 8-bit value that specifies the facility and the Unix partition.

Bits 0 - 3: Defines the Unix partition. Four bits allow for sixteen Unix partitions (a - p). These bits must all be set to "0" for a Master Device, which doesn't use Unix partitions.

Bits 4 - 7: Defines the IPI facility number offset. This value is added to the base facility value defined by the *bootmagic* variable IPI_SLAVE_*x*_FACILITY. Four bits allow for sixteen device facilities.

node The remote HIPPI I/O node to be used.

The following example creates a block disk device special file "imd (IPI Magnetic Disk) slave 0 partition a on remote node 7".

```
> rmknod /dev/ipi/imd0a b 23 0 7
```

The next example creates a raw disk device special file "rimd (Raw IPI Magnetic Disk) slave 7 partition d, again on remote node 7":

```
> rmknod /dev/ipi/rimd7d c 40 3 7
```

The third example creates a raw master device special file "ripm (Raw Intelligent Peripheral Master) slave 1 on remote node 7"

```
> rmknod /dev/ipi/ripm1 c 26 0 7
```

Creating a Disk Label

After the interface is connected, the next step in setting up an IPI-3 device for use in a Paragon system is to create a label for the new disk in the */etc/disktab* file. The label describes the disk geometry, and divides the disk into Unix partitions. It is dependent on the specifications of the hardware. Refer to the manual page for the *disktab* file for information about the file format. Refer to the manufacturer's specifications for details about a specific IPI device.

NOTE

The disk label for the Maximum Strategy GEN-4 device is included in the standard */etc/disktab* file for the Paragon™ system. The disk label may need modification, depending upon the exact configuration of your GEN-4 system.

The following example is the disk label for the Maximum Strategy GEN-4 device:

```
gen4|GEN-4|Maximum Strategy GEN-4:\
        :ty=winchester:dt=IPI:\
        :ns#15:nt#9:nc#2626:\
        :sc#135:su#354510:se#65536:sf#0:\
        :rm#5400:sk#0:cs#0:hs#0:ts#1700:il#1:\
        :bs#65536:sb#65536:\
        :oa#0:pa#32670:ba#524288:fa#65536:ta=4.2BSD:\
        :ob#32670:pb#32670:bb#524288:fb#65536:tb=4.2BSD\
        :oc#0:pc#354510:tc=all:\
        :od#65340:pd#32670:bd#524288:fd#65536:td=4.2BSD:\
        :oe#98010:pe#32670:be#524288:fe#65536:te=4.2BSD:\
        :of#130680:pf#32670:bf#524288:ff#65536:tf=4.2BSD:\
```

```
:og#163350:pg#32670:bg#524288:fg#65536:tg=4.2BSD:\
:oh#196020:ph#32670:bh#524288:fh#65536:th=4.2BSD:\
:oi#228690:pi#32670:bi#524288:fi#65536:ti=4.2BSD:\
:oj#261360:pj#32670:bj#524288:fj#65536:tj=4.2BSD:\
:ok#294030:pk#32670:bk#524288:fk#65536:tk=4.2BSD:\
:ol#326700:pl#27810:bl#524288:fl#65536:tl=4.2BSD:
```

Labeling the IPI Device

After defining the disk label, you must use the operating system's **disklabel** command to write the label information on to the external IPI device. The following example shows how to write the disk label for on to a Maximum Strategy GEN-4 device that's connected to the raw (character-mode) device "rimd0a".

```
> disklabel -r -w /dev/ipi/rimd0a gen4
```

Refer to the *Paragon™ System Administrator's Guide* or the online manual page for the disklabel(8) command for more information.

Creating New File Systems

If you are using file system I/O, you must create the file systems on the IPI device, using the **newfs** command.

The following example creates a new file system on the device with the block-device name "imd0" and partition "j", using the default file-system parameters:

```
> newfs /dev/ipi/imd0j
```

Refer to the *Paragon™ System Administrator's Guide* or the online manual page for the **newfs** command for more information about creating new file systems and the default parameters.

Adding IPI Entries to Mount Tables

Add entries for the file systems on IPI devices to the */etc/fstab* and */etc/pfstab* files to make those file systems available during multi-user sessions on the Paragon™ system.

NOTE

While the IPI-3 interface does not support I/O with Unix file systems (UFS), you need to mount an IPI-3 device as a UFS file system to use it as a PFS stripe directory.

The following example defines partition "j" on an IPI-3 device with the block name "imd0" to be mounted in the directory */home/.sdirs/vol0,* to be used as a PFS stripe directory for the PFS filesystem mounted at */pfs*.

```
/dev/ipi/imd0j /home/.sdirs/vol0   ufs rw 0 4
/dev/io0/rz0d  /pfs                pfs rw, stripegroup=one 0 5
```

Refer to the *Paragon™ System Administrator's Guide* or the online manual page for the *fstab* and *pfstab* file for more information about the file-system-table files.

Mounting The File Systems

Use either the ***mount -a*** command reboot the Paragon system to mount the file systems defined in the */etc/fstab* file.

Optimizing IPI-3 PFS Performance

The performance of IPI-3 PFS file systems is dependent on many variables. Included among these is the application usage model, layout of the file system, device partitioning, PFS striping factor, I/O request size, and the device itself. With such a large numbe of variables, it is difficult to suggest a configuration that will satisfy the I/O requirements of every user. However, the following recommendations may help to increase the file system performance. This is not a complete list of available options—you may find others as you gain experience.

newfs Option
: The **newfs** command offers many options that may be used to control the layout of the file system and the manner in which the file system accesses the device. For example, increasing the number of cylinders per group (**-c** option) may be beneficial, especially when combined with an increase in the maximum blocks per group (**-e** option).

PFS Configuration
: The PFS file system has many unique configuration options that may affect the I/O performance. For example, increasing the stripe unit size from the default of 64K may result in an increase the I/O performance. The performance of IPI-3 devices is usually optimized for large data transfers (512k and greater). It is important to understand how the device capabilities map to the application usage model.

Application I/O Requests
: The I/O requests generated by an application have a significant impact on the performance. In general, large sequential accesses out perform small or random accesses. It may be necessary to tune the application to take full advantage of the performance potential.

IPI-3 Device Options

The performance of IPI-3 devices is usually optimized for large data transfers (512k or greater). It is important to understand how the device capabilities map to the application and file system usage model. For example, creating a separate facility partition may allow you to customize the device for a particular application, perhaps by changing the RAID level on that partition.