IRIS® HIPPI API Programmer's Guide

Document Number 007-2227-005

CONTRIBUTORS

Originally written by Carlin Otto and Thomas Skibo
Revised by Carlin Otto and Jim Pinkerton
Illustrated by Carlin Otto, Dan Young, and Cheri Brown
Production by Carlos Miqueo
Engineering contributions by Irene Kuffel, Jim Pinkerton and Thomas Skibo
St Peter's Basilica image courtesy of ENEL SpA and InfoByte SpA. Disk Thrower image courtesy of Xavier Berenguer, Animatica.

IRIS® HIPPI API Programmer's Guide
Document Number 007-2227-005

Contents

List of Figures

List of Tables

About This Guide

This document describes the programmatic interface to IRIS HIPPI 3.2, the Silicon Graphics implementation of the High-Performance Parallel Interface: HIPPI-800 and HIPPI-Serial protocols. This information is intended for use by developers of upper-layer applications that use HIPPI as their communication medium.

Support for Upper Layer Applications

IRIS HIPPI supports the following upper layer applications:

- customer-developed applications using one of the IRIS HIPPI APIs:
  The application programming interfaces (APIs), described in this document, enable customers to develop or port their upper-layer applications for operation over IRIS HIPPI. Chapter 2 describes the character-device (**ioctl()**) API for direct access to the IRIS HIPPI subsystem (HIPPI-PH) or for data exchange using the Framing Protocol (HIPPI-FP). Chapter 3 describes IRIX' sockets-based API for the Internet Protocol (IP) network-layer stack.
- standard IRIX applications:
  For Internet (IP) communication, IRIS HIPPI supports IP over HIPPI-LE in conformance with RFC 1374 guidelines. All standard IRIX IP applications can use the IP-over-HIPPI interface (*hip*#), much as they would IP over Ethernet or FDDI. See the online InSight document *IRIS HIPPI Administrator's Guide* (shipped with the IRIS HIPPI software) for configuration details.
- IRIS HIPPI utilities:
  IRIS HIPPI includes utilities for monitoring, maintaining, and testing the IRIS HIPPI subsystem.

Style Conventions

This guide uses the following stylistic conventions:

`code sample`
Indicates the text for code, exactly as you must type it.

variable
Indicates generic, place-holding variable names within the code, where you must replace the variable with text that you select. For example, you might replace *hexvalue* with 0x3C.

file name
Indicates file names and file name suffixes.

[ ]
Encloses optional arguments.

...
Denotes omitted material or indicates that the preceding items may appear more than once in succession.

Product Support

Silicon Graphics, Inc., provides a comprehensive product support and maintenance program for its products. If you are in North America and would like support for your Silicon Graphics-supported products, contact the Technical Assistance Center at 1-800-800-4SGI. If you are outside North America, contact the Silicon Graphics subsidiary or authorized distributor in your country.

Obtaining Updated or Paper-copy Versions of This Document

Silicon Graphics maintains a World Wide Web page from which you can retrieve the latest versions of many of the company's documents, and from which you can obtain instructions for ordering printed (paper-copy) versions of online documents. Using your Web browser, open the following URL:

http://techpubs.sgi.com/library/

To locate the latest versions of IRIS HIPPI documents (including this one), make the following selections:

1. Click on the "Library Search" option.
2. Enter **`hippi`** to search for all titles that contain this string.
3. Click on the document that you want to view, download and print it, or purchase it in bound/hardcopy format.

Chapter 1

Description of IRIS HIPPI Implementation

This chapter describes the design and architecture of IRIS HIPPI. The functionality described in this chapter is directly available to programs that use the character device API. For programs using the IRIX sockets API (for example, to use the services of the IP protocol stack), this functionality is hidden by the network stack.

Overview of IRIS HIPPI Implementation

This section provides an overview of IRIS HIPPI.

Conformance with HIPPI Standards

The Silicon Graphics implementations of HIPPI provide the services and conform to the protocols described in the HIPPI standards for HIPPI-PH, HIPPI-Serial (when appropriate), HIPPI-FP, the host portion of HIPPI-SC, and HIPPI-LE. Also see sections, "How HIPPI Protocol Items Are Handled With the HIPPI-PH Access Method" on page 12, and "How HIPPI Protocol Items Are Handled With the HIPPI-FP Access Method" on page 18.

Basic Architecture

IRIS HIPPI supports transmission and reception as separate channels. Because of this design, it is possible for a system to have applications that only receive, applications that only send, and/or applications that do both. In addition, each channel can be accessed with a different access method. (The access methods are described in section, "The Device File and Access Methods" on page 6.)

Note: Due to the nature of HIPPI-Serial, simultaneous use of the channels to different endpoints requires the presence of a HIPPI switch between each source/destination pair. For each HIPPI-PH connection over HIPPI-Serial, the switch demultiplexes the backflowing control signals (for example, **READY**s).

For transmission, IRIS HIPPI supports four different functionality scenarios, summarized in Table 1-1 and described in the text below the table. Each functionality scenario is a combination of one connection control method (the rows of Table 1-1) and one packet control method (the columns of Table 1-1). The bullets describe these two control methods:

- Connection control: A connection can be single-packet or many-packet. A single-packet connection is when one packet is sent and then the HIPPI subsystem automatically closes the connection. A many-packet connection is a connection that is kept open for as long as the application wants. In the latter case, the application must indicate when it wants the connection closed.
- Packet control: A packet can be single-write or multiple-write. A single-write packet is a HIPPI packet that is created by the HIPPI subsystem from an application's single *write()* call. A multiple-write packet is created from two or more *write()* calls. In the latter case, the application must indicate the start of each packet.

Table 1-1 Transmission Functionality Scenarios for IRIS HIPPI

		Packet Control	
		single-write = 1 packet	**multiple-writes = 1 packet**
Connection Control	**single-packet connection**	scenario 1	scenario 3
	many-packet or long-term connection	scenario 2	scenario 4

1. Single-packet connection, single-write packet:
    - The application does one *write()*, which causes the connection to be opened.
    - One packet is sent. It consists of the data in the *write()* call.
    - The HIPPI subsystem automatically closes the connection.

2. Many-packet connection, single-write packets:
    - The application asks for a long-term connection.
    - The application does the first *write()* call and the connection is opened.
    - One packet is sent. It consists of the data in the *write()* call.
    - The application does any number of *write()* calls.
    - One packet is created and sent from each *write()*.
    - The connection remains open until the application closes it.
3. Single-packet connection, multiple-write packet:
    - The application indicates the length of a packet.
    - The application does the first *write()* call, which causes the connection to be opened.
    - The application continues to do *write()* calls.
    - A packet is sent. Data from the *write()* calls, up to the specified packet length, are sent as one packet. The packet length can be indeterminate.
    - The HIPPI subsystem closes the connection.
4. Many-packet connection, multiple-write packets:
    - The application asks for a long-term connection.
    - The application indicates the length of a packet. This length may be indeterminate or a specified bytecount.
    - The application does the first *write()* call and the connection is opened.
    - The application does any number of *write()* calls.
    - A packet is sent. Data from the *write()* calls, up to the specified packet length, are sent as one packet. The packet length can be indeterminate.
    - The application may indicate the end of the packet at any time.
    - The application can send any number of packets by indicating a length for each new packet.
    - The connection remains open until the application closes it.

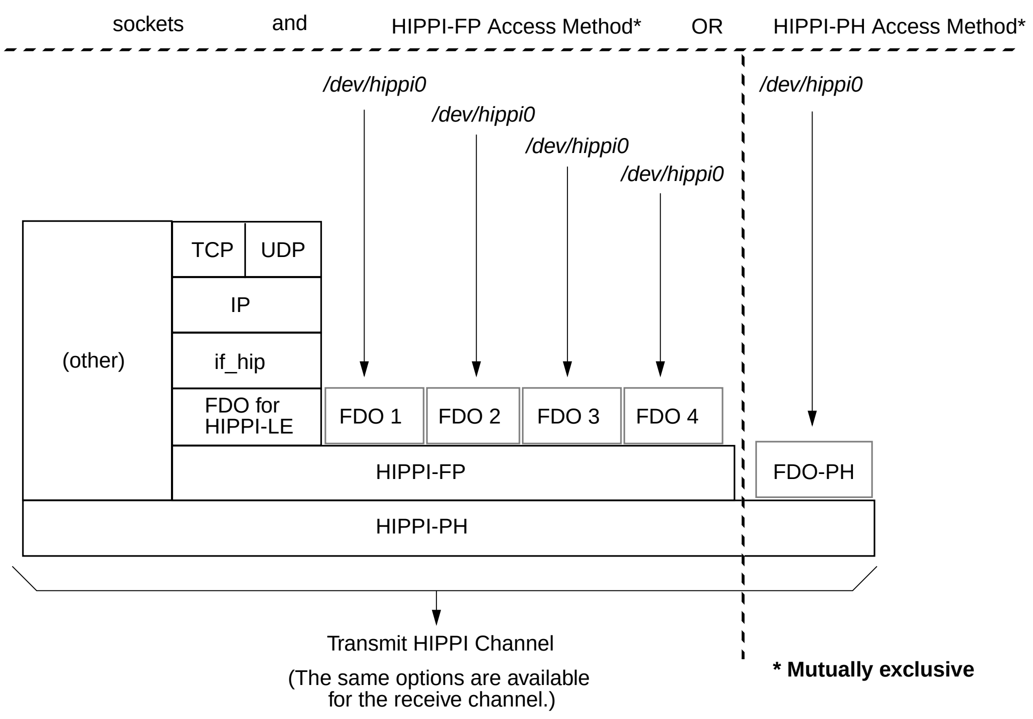


Figure 1-1 Interfaces for Controlling One Channel of the HIPPI Subsystem

The IRIS HIPPI implementation consists of 4 main components:

- the device file (*/dev/hippi#*) and access methods for controlling the HIPPI subsystem, illustrated in Figure 1-1
- transmission-related information objects (FDOs) for each open device file descriptor
- reception-related information objects (ULPOs) for different upper-layer protocol applications (ULPs)
- the application programming interface (API)

Each of these components is described in a section below.

The Transmission Information Object for File Descriptors

A portion of the IRIS HIPPI subsystem, within the UNIX kernel, maintains transmission information objects, referred to as *file descriptor objects* (FDOs). An FDO is maintained for each open file descriptor. The HIPPI subsystem uses this information for generating HIPPI packets on transmission. Applications change the FDO values through the IRIS HIPPI API. The values are persistent, so they can be used on sequentially sent packets, without resetting.

- the ULP's identification number (that is, the ULP-id)
- access mode (read-only, write-only, read and write)
- the I-field used when establishing a connection
- the size of the first burst for each packet
- the setting for the B-bit in the FP header (used by HIPPI-FP only)
- the setting for the P-bit of the FP header (used by HIPPI-FP only)
- the size of the D1 area (used by HIPPI-FP only)

The Reception Information Object for Upper Layer Protocols

A portion of the IRIS HIPPI subsystem, within the UNIX kernel, maintains reception information objects, called *upper-layer protocol objects* (ULPOs). A ULPO is maintained for each ULP-id. Each ULPO consists of a set of information that the HIPPI subsystem uses when receiving HIPPI packets. Each application must have a ULPO associated with (bound to) it. With HIPPI-FP, the information in one ULPO can be shared among a group of applications. Applications use the HIPPI API (*ioctl* calls) to change their ULPO values.

Note: HIPPI-LE (over which TCP/IP runs) is an example of a ULP.[1] ULPs that customers may develop include IPI-3 and "raw" protocols.

Each ULPO maintains the following information:

- the ULP's identification number (that is, the ULP-id), used only by HIPPI-FP
- the number of applications using this ULP-id, used only by HIPPI-FP
- the received bytecount for the packet currently being read

[1] Currently the information in the HIPPI-LE's ULPO cannot be shared with other network-layer applications.

The APIs

The IRIS HIPPI product includes 2 application programming interfaces (APIs): a character device API composed of *ioctl()* calls and an IRIX sockets API that uses the standard BSD-socket calls.

The character device API allows customer-developed applications to change the information in their ULPO and FDO and to control the HIPPI subsystem. This API is through a UNIX "character special" device file. By invoking different IRIS HIPPI API commands, customer-developed programs define their access method, data flow (packet) control, connection control, and HIPPI protocol processing.

Details on the character device API are provided in Chapter 2, "Programming Notes for Character Device API." Details for the socket API are provided in Chapter 3, "Programming Notes for Sockets-based API."

The Device File and Access Methods

The IRIS HIPPI implementation provides the */dev/hippi#* device file for accessing and controlling the HIPPI subsystem. Two different access methods (described below) are provided. Both use the */dev/hippi#* device file. The access method is defined when the device file is bound.

IRIS HIPPI offers two mutually exclusive methods for accessing the HIPPI subsystem: HIPPI-FP and HIPPI-PH. The HIPPI-FP access method requires the use of FP headers and provides automatic processing of that header. The HIPPI-PH method does not require use of the FP header, thus allowing an application to bypass the HIPPI-FP layer.

Besides the difference (discussed above) in the point of access, the main differences between the two access methods revolve around coexistence with other ULPs (including HIPPI-LE and the TCP/IP stack), as listed below:

- For HIPPI-FP, received packets are demultiplexed using ULP-ids. For HIPPI-PH, all packets are placed on a single input queue.
- For HIPPI-FP, access to the HIPPI subsystem is shared among two or more ULPs (of which the IP protocol stack is one). With HIPPI-FP, the HIPPI device is blocked when a ULP is accessing the device, thus giving each ULP exclusive access for the duration of its access. With the HIPPI-PH access method, no blocking occurs because only one ULP is allowed to be bound to the HIPPI device at any time.

Further details are provided in sections, “The HIPPI-PH Access Method” on page 8, and “The HIPPI-FP Access Method” on page 14.

Implementation Details

The Silicon Graphics IRIS HIPPI Product (HIPPI board, device driver, interface, and firmware) has been designed to meet the ANSI X3T11 Standards Committee’s standards for the High-Performance Parallel Interface.[1] The HIPPI board design includes the following implementation details that are either not mentioned in the ANSI documentation for the HIPPI standard, or are considered by the design team to be ambiguously defined in the standards documentation:

- Once a receiving channel has been created and configured to accept connections, all incoming connection requests are accepted by the HIPPI board. The HIPPI subsystem does not wait for upper-layer input. (That is, the HIPPI board does not generate the service primitive PH_RING.Indicate and does not allow the application to respond with a PH_ANSWER.Request for each connection). The upper layers may discard data that has been received from undesirable connections.
- Each instance of a ULPO must be assigned a ULP-id that is unique within the ULPOs for a specific HIPPI board. A valid ULP-id is a number between 0 and 255 decimal (inclusive); 4 is reserved for and used by the IRIX module implementing 8802.2 Link Encapsulation (HIPPI-LE); 7 is reserved for IPI-3 implementations.

Coexistence With the IP Stack

IRIS HIPPI includes a ULP module (HIPPI-LE) for servicing the IP stack. This module can coexist with customer-developed applications; however, IP network traffic has requirements that, in some situations, conflict with full usage of some features of the HIPPI subsystem. These requirements are:

- IP applications need short but frequent access to the HIPPI subsystem.
- IP requires bi-directional flow of data (for example, data packets in one direction require acknowledgment packets in the other direction).

[1] HIPPI-LE, HIPPI-FP, and HIPPI-PH.

Customer-developed applications can coexist well with IP applications if they follow these guideline:

- They must not create (open) long-term connections. Since only one connection at a time can use a HIPPI channel, if any one ULP monopolizes the channel, the IP performance degrades significantly.
- They must limit their *write()*s to small-sizes (for example, 128kbytes).
- They must *read()* their input queue in a timely fashion so that host buffers are always available to retrieve packets from the board. Arriving packets are enqueued in the IRIS HIPPI board's memory and are transferred from the board into host memory on a first-arrived-first-transferred basis. As long as a packet in the queue is not read, all subsequently arriving packets queue up behind it. IP performance degrades when packet latencies become excessive. Additionally, the onboard memory can fill up and arriving packets can be dropped.
- The physical connections for both channels must attach to the same node. For example, with a copper-based product, both cables must be attached to the same switch or HIPPI node.

Note: In addition to degrading the performance of local IP applications, failure to follow these guidelines may cause wide performance variations or hangs at intermediate switches or at the endpoints with which the station is exchanging data.

The HIPPI-PH Access Method

The HIPPI-PH access method controls the HIPPI protocol stack at the HIPPI-PH signal layer. This access method is independent of the physical layer, so it works for both copper-based and fiber optic-based hardware. With this access method, the HIPPI-FP protocol is bypassed; the IRIS HIPPI subsystem does no checking for or processing of HIPPI-FP protocol items. Accessing the HIPPI subsystem in this manner is well-suited for applications requiring full or almost full use of the HIPPI device, and situations where (for other reasons) the application does not wish to use the HIPPI-FP protocol.

Description of HIPPI-PH

HIPPI-PH supports the following functions for reception:

- Accepts all incoming HIPPI packets. Does not reject any packet, and does not demultiplex using the ULP-id.
- Enqueues the entire packet on the input queue for retrieval by the application. Does not interpret anything in the packet (not an FP header or D1 data).
- The HIPPI subsystem maintains a received bytecount value (offset) that can be used by applications to identify packet boundaries

HIPPI-PH supports the following functions for transmission:

- Provides two choices for setting up the HIPPI connection: a single-packet connection (where the HIPPI subsystem creates and tears down a connection for each packet), and a long-term connection (where the HIPPI subsystem keeps the connection up across one, many, or all packets). In long-term connections, the application controls the timing of the disconnect.
- Provides two methods for creating packets: “multiple-write” packets (where the **PACKET** signal is asserted across multiple *write()* calls) and single-write packets (where the **PACKET** signal is deasserted when the data from one *write()* has been transmitted). The maximum bytecount for any *write()* is 16 megabytes, so a single-write packet cannot be larger than 16 megabytes.
- Allows an application to send an “infinite” sized packet.
- Allows an application to specify that the first burst of any packet be a short burst.
- Allows an application to terminate a multiple-write packet before its bytecount is transmitted.
- Allows an application to use the HIPPI subsystem as a “raw” data pipeline. For example, the FP header is not required and the I-field can be set to any value.
- HIPPI-PH supports the four different functionality scenarios for transmission, summarized in Table 1-1.

Applications using the HIPPI-PH access method can open the device for transmit only (illustrated in Figure 1-2), receive only (also illustrated in Figure 1-2), or for both (illustrated in Figure 1-3). Once the device is open and bound, any of the functionality scenarios in Table 1-1 can be used.

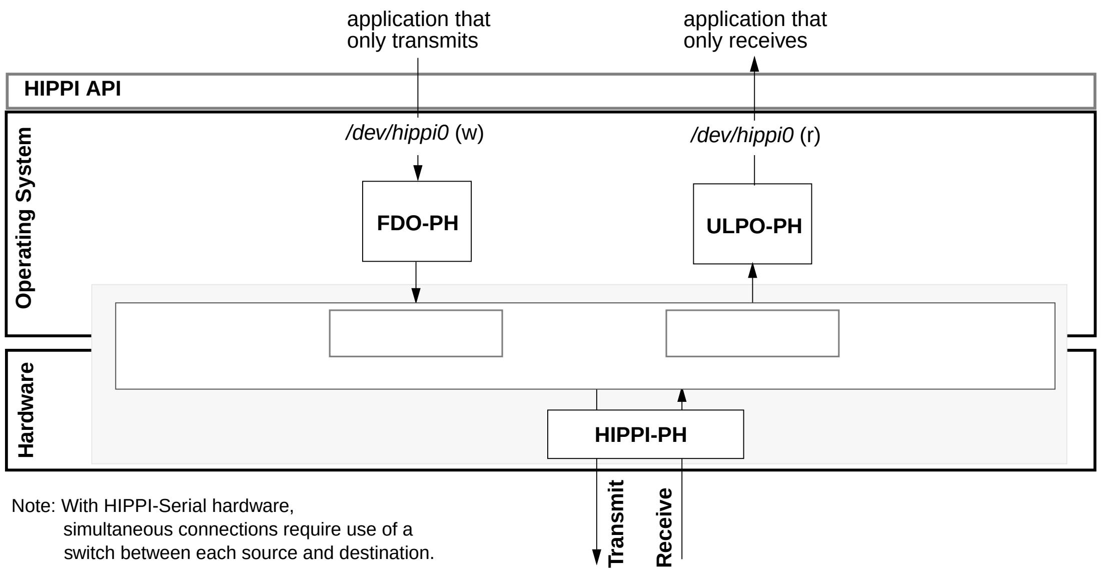

application that only transmits
application that only receives
HIPPI API
Operating System
/dev/hippi0 (w)
/dev/hippi0 (r)
FDO-PH
ULPO-PH
Hardware
HIPPI-PH
Transmit
Receive
Note: With HIPPI-Serial hardware, simultaneous connections require use of a switch between each source and destination.

Figure 1-2 Block Diagram for Two Applications Using HIPPI-PH: One Receive-Only and One Transmit-Only

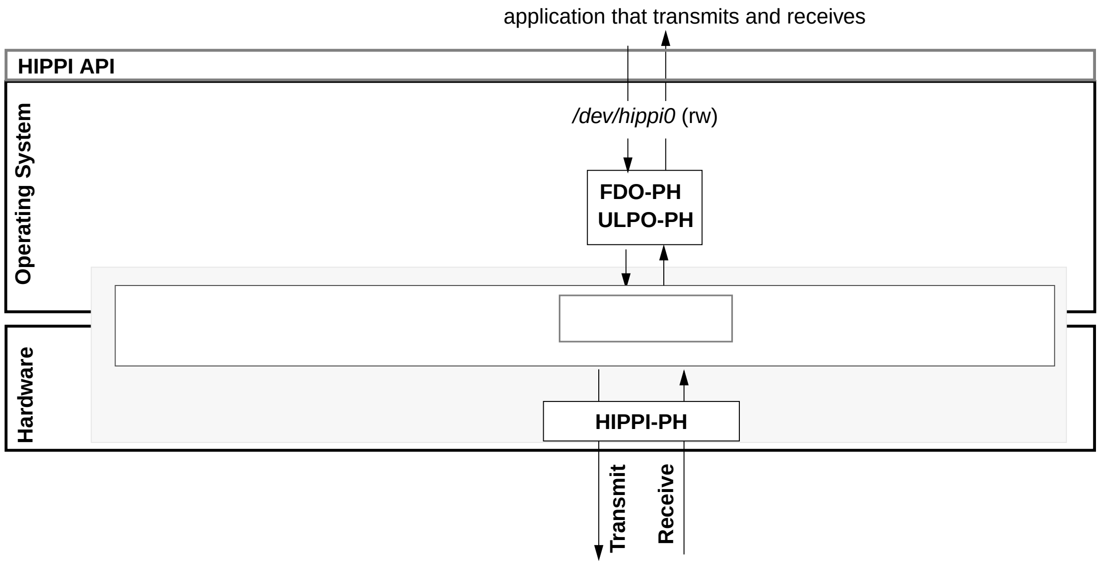


Figure 1-3 Block Diagram for One Application Using HIPPI-PH: Receive and Transmit

There are certain constraints associated with using the HIPPI-PH access method, as listed below. However, with HIPPI-PH overall, there are fewer constraints on what the application can do with the interface than with HIPPI-FP.

- If more than one application operates a channel (transmit or receive) of the HIPPI board, an arbitration and synchronization mechanism between the applications must be developed to prevent race conditions.
- The HIPPI network interface cannot be *ifconfig*'ed up, which means that the TCP/IP protocol stack cannot use the HIPPI board.
- For receiving, if a HIPPI-FP header exists in the packet, it is not interpreted (and not demultiplexed) by the HIPPI subsystem.
- All *read()*s and *write()*s must specify buffers that are 8-byte word-aligned. This is because direct memory access (DMA) occurs directly to/from user application space and the HIPPI device only handles word-aligned DMAs.
- The data lengths for all *read()*s and *write()*s must be multiples of 8 bytes up to a maximum of 16 megabytes.

HIPPI-PH Output

When a device is opened for writing and bound with the HIPPI-PH access method, the HIPPI subsystem transmits only data that the application passes to it. No additional data, encapsulation, or HIPPI protocols are added by the HIPPI subsystem. The only information used from the application’s FDO is the I-field and the short burst setting.

The application can define the first burst as short within each packet.

HIPPI-PH Input

When a device is opened for reading and bound with the HIPPI-PH access method, the HIPPI subsystem receives **all** inbound data; all packets are enqueued on the reading queue. The HIPPI subsystem does not attempt to interpret an FP header; therefore, if an FP/D1 header exists, these are passed to the application as part of the data stream. No demultiplexing is performed on the ULP-id. No special handling features are available.

The application can retrieve a packet bytecount (offset) value that simplifies identification of packet boundaries.

How HIPPI Protocol Items Are Handled With the HIPPI-PH Access Method

This section describes how the HIPPI-PH access method handles the HIPPI I-field and the HIPPI Framing Protocol.

How I-Fields Are Handled on Transmission

The FDO maintains a value for the I-field. The HIPPI subsystem uses the value each time it sets up a connection. Applications use an *ioctl()* call to set the I-field value to a new one whenever desired. The HIPPI subsystem does not interpret or alter the I-field in any way during transmission.

How I-Fields Are Handled on Reception

The HIPPI subsystem does not interpret or alter the I-field in any way for reception.

How the Framing Protocol and D1 Data Are Handled on Transmission

The HIPPI-PH access method does not generate an FP header or D1 data area for packets on transmission. An application may utilize the HIPPI Framing Protocol by generating its own FP header/D1 data and transmitting these just as it does all other packet data (with *write* calls), as illustrated in Figure 1-4. An application may invoke an *ioctl()* call to define the bytecount for the first (short) burst; the data for that first burst is taken from the first *write()* call, as illustrated in Figure 1-5.

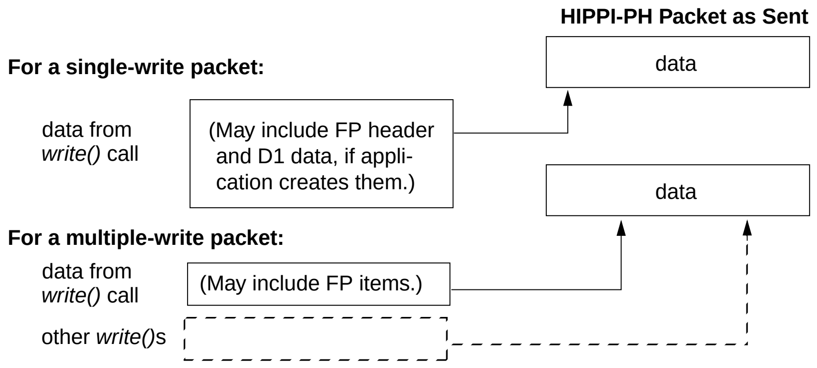

HIPPI-PH Packet as Sent
For a single-write packet:
data
data from write() call
(May include FP header and D1 data, if application creates them.)
data
For a multiple-write packet:
data from write() call
(May include FP items.)
other write()s

Figure 1-4 Creation of HIPPI-PH Packet

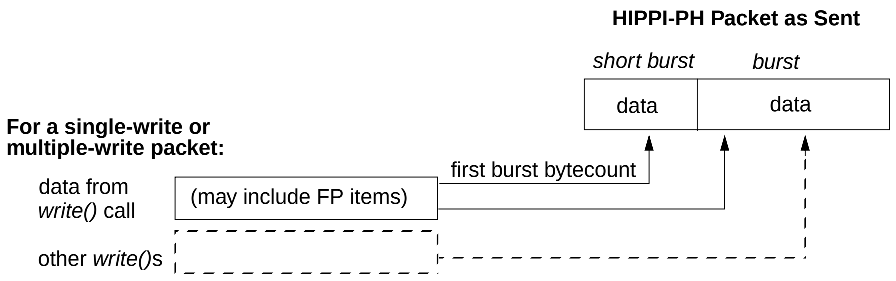


Figure 1-5 Creation of HIPPI-PH Packet With First Short Burst

How the Framing Protocol and D1 Data Are Handled on Reception

On incoming packets, HIPPI-PH does not check for the presence of an FP header nor does it interpret the FP header if one exists. If an FP header/D1 area are present in a packet, the HIPPI subsystem treats that packet just as it does any other packet (enqueuing the contents of the packet on the receive queue without any interpretation, separation, or special processing).

The HIPPI-FP Access Method

Description of HIPPI-FP

The HIPPI-FP access method controls the HIPPI protocol stack at the HIPPI-FP layer and provides for sharing of the HIPPI receive and/or transmit channels among applications, as illustrated in Figure 1-6. Up to 32 different applications (open file descriptors) can use the IRIS HIPPI subsystem simultaneously in any combination of transmitting-only, receiving-only, and transmitting-and-receiving.

HIPPI-FP supports the following functions for reception:

- Up to 32 different applications can simultaneously have open file descriptors for receiving HIPPI packets.
- Up to 8 different customer-developed ULPOs can be active simultaneously. Each application must bind to one ULPO. HIPPI-IPI is an example of a ULPO. (The HIPPI-LE module that is part of the HIPPI product does not count as one of these ULPOs.)
- The HIPPI subsystem demultiplexes incoming packets using the ULP-ids from the active ULPOs. It discards packets that do not match any of the ULP-ids in active ULPOs.
- For HIPPI-FP packets, the HIPPI subsystem verifies the presence of a valid FP header and discards packets that do not have a valid FP header.
- For HIPPI-FP packets, the HIPPI subsystem separates the FP header and D1 data from the D2 data so that the application’s first *read()* retrieves the FP header and D1 data, while its subsequent *read()*s retrieve the D2 data.
- The HIPPI subsystem maintains a packet offset value (bytecount) that can be used by character device applications to identify packet boundaries.

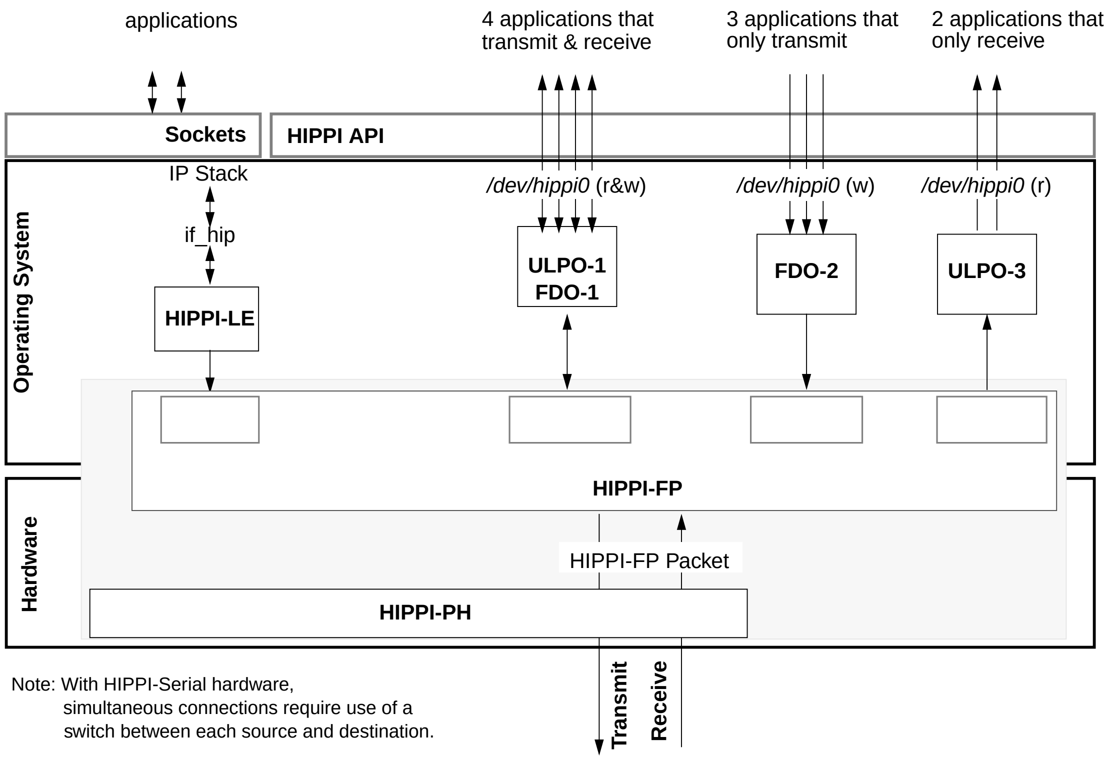

applications
4 applications that transmit & receive
3 applications that only transmit
2 applications that only receive
Sockets
HIPPI API
IP Stack
if_hip
HIPPI-LE
/dev/hippi0 (r&w)
/dev/hippi0 (w)
/dev/hippi0 (r)
ULPO-1
FDO-1
FDO-2
ULPO-3
Operating System
HIPPI-FP
HIPPI-FP Packet
Hardware
HIPPI-PH
Transmit
Receive
Note: With HIPPI-Serial hardware, simultaneous connections require use of a switch between each source and destination.

Figure 1-6 Block Diagram for Using HIPPI-FP

- User-layer applications can use the IRIX socket interface to access the services of network-layer stacks: the default stack is the Internet Protocol (IP) suite. The HIPPI product ships with a socket-based driver, *if_hip*, that supports IP-over-HIPPI (using HIPPI-LE). Customer-developed character device programs can coexist with socket-based programs. (This coexistence is not available in "mixed" configurations, where one of the HIPPI channels is being used by HIPPI-FP and the other by HIPPI-PH.)

  Note: The IP software requires equal-access to its lower layer services. If applications sharing the HIPPI subsystem with IP do not meet this requirement, the performance of IP is seriously compromised.

- Special "auto-bind" device files can be set up with more general permissions in order to allow user access to specific ULPOs.

HIPPI-FP supports the following functions for transmission:

- Up to 32 different applications can simultaneously have open file descriptors for transmitting HIPPI packets.
- Up to 8 different customer-developed FDOs can be active simultaneously. Each application must bind to one FDO. HIPPI-IPI is an example of an FDO. (The HIPPI-LE module that is part of the HIPPI product does not count as one of these FDOs.)
- The HIPPI subsystem creates an FP header for each packet.
- Allows an application to specify that the first burst of any packet contains only the FP header and, optionally, a D1 area. The word count of this first burst can be short (1 to 255 words) or standard (256 words). The B bit in the FP header is automatically set.
- Allows an application to specify the presence of D1 data and the size of the D1 area. The P bit in the FP header is automatically set.
- HIPPI-FP supports the four different functionality scenarios for transmission, summarized in Table 1-1.

With HIPPI-FP, there are certain constraints, as listed below:

- For transmitting, a HIPPI-FP header is attached to every outgoing packet. If the application does not specify an FP header, the HIPPI subsystem uses a default one (illustrated in Figure 1-7).
- For receiving, each incoming packet must have a valid HIPPI-FP header. The HIPPI subsystem demultiplexes incoming packets based upon the ULP identifier in the FP header.
- Incoming connection requests cannot be selectively rejected by the ULPO or application; each incoming connection request results in acceptance of the packet. However, the HIPPI subsystem discards packets with ULP-ids that do not match any of those that are currently bound. All applications that have opened a HIPPI file descriptor for receiving (reading) and have bound to a ULPO will receive all incoming packets destined to the bound ULP-id.
- All *read()*s and *write()*s must specify buffers that are 8-byte word-aligned. This is because direct memory access (DMA) occurs directly to or from user application space and the HIPPI device only handles word-aligned DMAs.
- The data lengths for all *read()*s and *write()*s must be multiples of 8 bytes up to a maximum of 16 megabytes.

HIPPI-FP Output

The destination endpoint is specified by an *ioctl()* call that sets the I-field for all subsequent packets (until the value is changed). The I-field can be changed at any time, and the *ioctl()* call is efficient enough that there is no problem with setting the I-field just before each *write()* call for a series of single-write packets.

HIPPI FP headers are automatically generated on output. The default FP header (illustrated in Figure 1-7) has the ULP identifier (specified at bind time), does not have any D1 area, and the P and B bits are off. If an application defines a size for the D1 area or specifies a first burst containing only FP header and D1 area, this information is included in the header. The D2_Size field in the FP header is filled with the proper value.

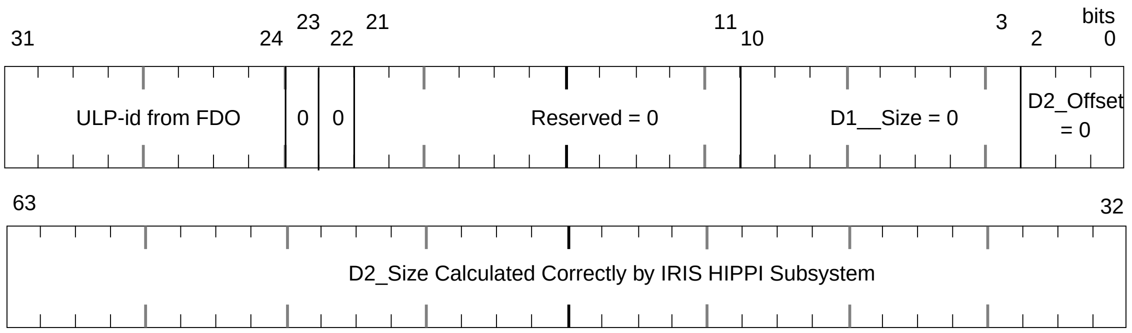


Figure 1-7 Default FP Header for HIPPI-FP Transmission

HIPPI-FP Input

When a device is opened for reading and bound to a ULPO with the HIPPI-FP access method, the associated application is able to retrieve all HIPPI packets that arrive with the bound ULP-id. The demultiplexing on ULP-ids is done on the HIPPI board so that DMA can occur directly to user-space. If multiple applications share a ULPO, demultiplexing the packets must be handled by an application-level program.

HIPPI-FP separates the FP header and D1_Data_Area from the D2 area. The application's first *read()* call returns the FPheader/D1data exactly. (The return value tells how large these areas are). Subsequent *read()* calls return the D2 area until the D2 data is completely read. By monitoring the HIPPI subsystem's packet offset value, the application can tell when the next *read()* is going to return the FPheader/D1area for a new packet.

It is possible to receive packets that are very large because reception can be broken up into multiple *read()*s. This also helps provide for some scattering of data, but small *read()*s are inefficient.

An application can retrieve a packet bytecount (offset) value that simplifies identification of packet boundaries.

How HIPPI Protocol Items Are Handled With the HIPPI-FP Access Method

This section describes how the HIPPI-FP access method handles the HIPPI I-field and the HIPPI Framing Protocol.

How I-Fields Are Handled on Transmission

Each FDO contains a value for the I-field that the HIPPI subsystem includes each time it sets up a connection. Applications use an *ioctl()* call to set the value to a new one whenever desired. The HIPPI subsystem does not interpret nor alter the I-field in any way during transmission.

How I-Fields Are Handled on Reception

The HIPPI subsystem does not interpret nor alter the I-field in any way during reception.

How the Framing Protocol Is Handled on Transmission

When accessed with the HIPPI-FP access method, the HIPPI subsystem uses the HIPPI Framing Protocol on all connections for receiving and transmitting, as explained below.

The HIPPI subsystem creates an FP header for each packet that it transmits. The default value for the generated FP header is as follows:

- D2_Size (that is, bits 63:32 of the FP header) is set to the size of the *write()* call for a single-write packet or, for a multiple-write packet, the bytecount indicated by the *ioctl()* call that starts the packet.
- ULP-id (that is, bits 31:24) is set to the ULP-id that was provided by the application when it bound to the ULPO.
- Control (P and B) and reserved bits are off (that is, bits 23:11 are set to zero).
- D1_Area_Size and D2_Offset are set to zero.

To include D1 data in a packet, an application specifies the size of the D1 area, using an *ioctl()* call. This action sets the D1_Area_Size in the FDO and causes the P bit in the FP header to be set ON. The application then does its *write* pointing to contiguous D1 and D2 data, or for a multiple-write packet, it does the first *write* pointing to D1 data or contiguous D1 and D2 data.

To place only the FP header and D1 data (optional) in the first burst and to set the B-bit, an application invokes an *ioctl()* call, specifying the size of the first burst. If the specified size is less than 256 words, the IRIS HIPPI subsystem handles the burst as a short burst. The HIPPI subsystem creates a first burst of the indicated size (short or standard length) that contains the following data:

- The required, and automatically generated, FP header (8 bytes).
- An optional number of D1 data bytes, up to a maximum of 1016.

The D2 size for the FP header is calculated by the IRIS HIPPI subsystem, as described immediately below.

- For a single-write packet, the D2 data size is the *size* of the *write()* call minus the D1_Area_Size, as shown in Figure 1-8.
- For any multiple-write packet, the D2 data size is as specified by an *ioctl()* command, as shown in Figure 1-9 (illustrating contiguous D1 and D2 data) and Figure 1-10 (illustrating FP header and D1 data separated into the first burst).

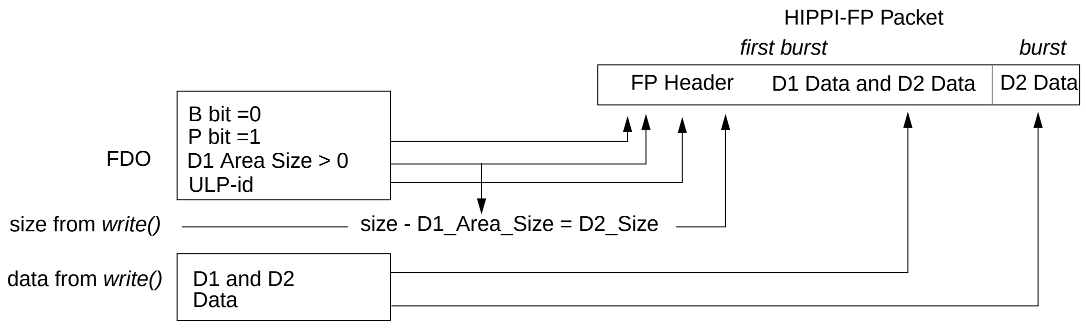


Figure 1-8 Single-Write HIPPI-FP Packet With D1 Data

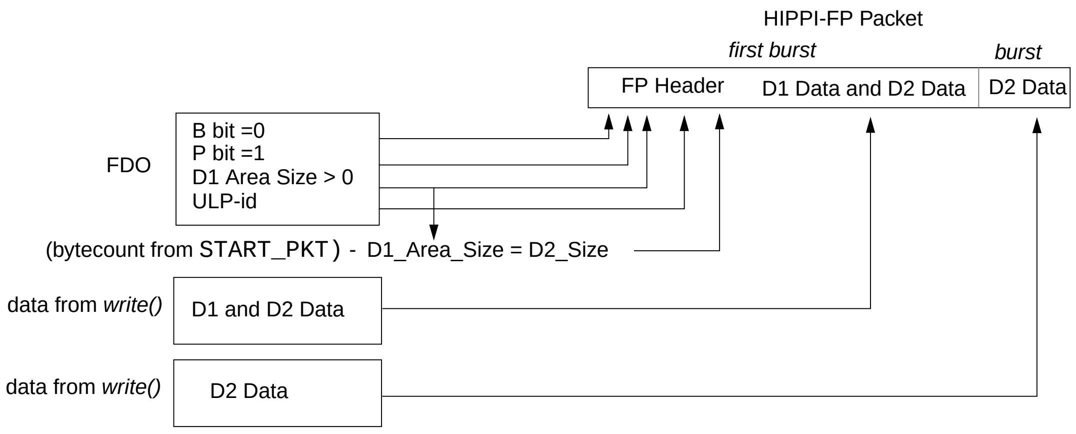


Figure 1-9 Multiple-Write HIPPI-FP Packet: Contiguous D1 and D2 Data

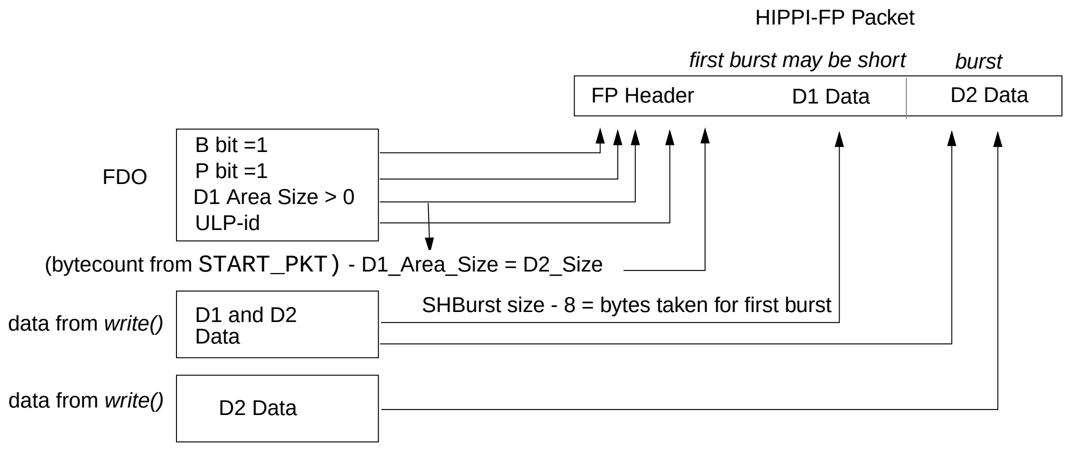


Figure 1-10 Multiple-Write HIPPI-FP Packet: Separate FP Header and D1 Data

How the Framing Protocol Is Handled on Reception

With each reception of a packet, the HIPPI subsystem interprets the FP header information, as required by the standard. The application's first *read()* retrieves the FPheader/D1data; subsequent *read()* calls retrieve D2 data. It is the responsibility of the reading application(s) to keep track of which *read()*s retrieve which kinds of data. The HIPPI subsystem demultiplexes incoming packets, using the ULP-id field. Incoming packets for unrecognized ULP-ids are discarded by the HIPPI subsystem.

Mixing HIPPI-PH and HIPPI-FP

HIPPI-PH and HIPPI-FP access methods can be used simultaneously so that they share one IRIS HIPPI board. There are some restrictions for this configuration:

- Each HIPPI-channel (receive or transmit) must be used by either HIPPI-PH or HIPPI-FP, but not both. For example, a number of applications and ULPOs can use HIPPI-FP for receiving demultiplexed data while a sending application uses HIPPI-PH, as illustrated in Figure 1-11.
- The TCP/IP over HIPPI-LE protocol stack cannot be supported because it requires HIPPI-FP access for both transmit and receive.
- A switch must be used between each the endpoints of each connection.

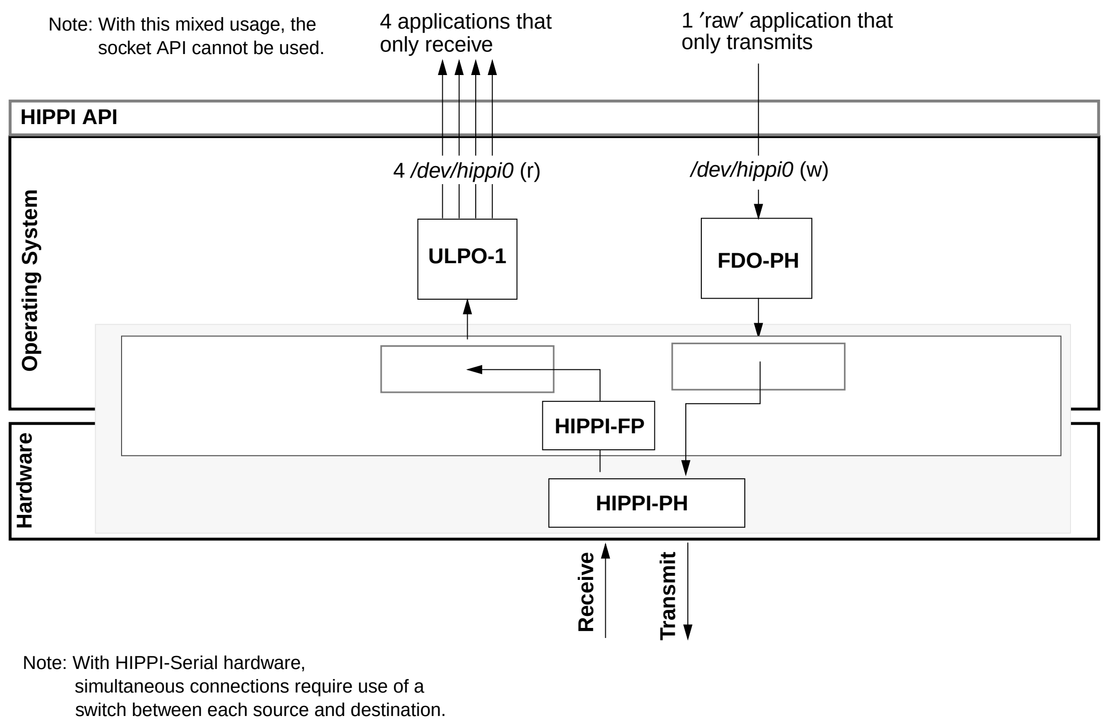

Note: With this mixed usage, the socket API cannot be used.
4 applications that only receive
1 'raw' application that only transmits
HIPPI API
Operating System
4 /dev/hippi0 (r)
/dev/hippi0 (w)
ULPO-1
FDO-PH
HIPPI-FP
Hardware
HIPPI-PH
Receive
Transmit
Note: With HIPPI-Serial hardware, simultaneous connections require use of a switch between each source and destination.

Figure 1-11 Block Diagram for Mixing HIPPI-PH and HIPPI-FP

Chapter 2

Programming Notes for Character Device API

This chapter describes how to interface an application to the IRIS HIPPI subsystem through the character device (*ioctl()*-based) application programming interface (API). A reference section containing an alphabetical listing of all the *ioctl()* calls is included under the heading "API Reference" on page 38. Code examples are provided in Appendix B, "Sample Programs."

Maximum Size for *write()*s and *read()*s

The variables *max_write_size* and *max_read_size* are used throughout these notes. The maximum size value you can use for these variables depends on the installed hardware, as summarized below.

- *max_xxx_size* = 2 megabytes when the hardware is a copper-based HIO board
- *max_xxx_size* = 16 megabytes when the hardware is a fiber-optics XIO board

To create a program that will behave correctly, regardless of which hardware is installed, include code that retrieves status from the hardware to determine the hardware type and sets the *max_xxx_size* appropriately. In the following example, *max_len* is used for both *max_write_size* and *max_read_size*:

```
/* verify which platform  */
if ( ioctl( fd_i, HIPIOC_GET_STATS, &hipstats ) < 0 ) {
    if ( errno == ENODEV ) fprintf(stderr,
        "%s: HIPPI board is down\n", argv[0]), exit(1);
    else
        perror("hipcntl: ioctl HIPIOC_GET_STATS failed"),exit(1);
    }

/* set the maximum length for reads and writes */
/* HST_FORMAT_ID_MASK and HST_XIO are defined in hippi.h */
max_len = ( (hipstats.hst_flags & HST_FORMAT_ID_MASK) == HST_XIO) ?
(0x1000000 + 8) : (0x200000 + 8);
```

Programming for the HIPPI-PH Access Method

This section describes how to program a module that accesses the HIPPI subsystem at the HIPPI-PH signalling layer. This access method does not use an intermediate protocol such as the HIPPI Framing Protocol. This API does not support sharing the subsystem with other APIs (for example, HIPPI-FP access mode or IRIX sockets). A sample program is provided under the heading “HIPPI-PH Example” on page 77.

Note: The interface described in this section is applicable for use with either the copper-based IRIS HIPPI-800 physical layer or the fiber optic-based HIPPI-Serial physical layer. The HIPPI-PH references within this section refer to only the signalling protocol that is common to these two HIPPI implementations.

Includes

The following file must be included in any program using the IRIS HIPPI API:

```
#include <sys/hippi.h>
```

Special Instructions

For maximum throughput, DMA between the HIPPI board and the host application occurs directly to/from user application space. Because of this, and the fact that the DMA component (ASIC) has a 64-bit interface, all application *read()*s and *write()*s must specify buffers that are 8-byte word-aligned, and the data bytecount must be a multiple of 8. (See the memalign(3C) reference page for a method of allocating 8-byte aligned memory).

Opening and Binding to the Device

An application can open a HIPPI device (for example, */dev/hippi0* or */dev/hippi1*) for read-only, write-only, or read-and-write access. The acronym *fd_hippi#*, in the examples below, refers to the file descriptor for the opened HIPPI device. Multiple applications can successfully *open()* a HIPPI device, although contention will occur if two or more try to *write()* at the same time.

Note: With a HIPPI-Serial physical layer, simultaneous connections to separate endpoints (for example, simultaneous use of a write-only file descriptor and a read-only one) requires the use of a HIPPI switch between the source and final destination.

It is important that the application open the HIPPI device with only the read/write flag settings that it needs. For example, if an application is not going to be doing *read()*s, it should set only the WRITE flag. When the READ flag is set, the HIPPI subsystem is told to expect HIPPI packets, so incoming packets are always accepted by the HIPPI device. The HIPPI subsystem holds each accepted packet until an application reads it. If no application consumes the incoming packets, the HIPPI device stalls for lack of buffer space.

Hint: This API does not provide a timeout or interrupt for *read()*s or *write()*s that do not complete. If you want your program to be interrupted when one of these calls fails to complete in a certain length of time, use the *alarm* mechanism. (See the alarm(2) reference page).

To set up an application as a HIPPI-PH user, use one of the following sets of calls at the "beginning of time":

- For a transmit-only connection:

  ```
  fd_hippi0=open ("/dev/hippi0", O_WRONLY);
  ioctl (fd_hippi#, HIPIOC_BIND_ULP, HIPPI_ULP_PH);
  ```

- For a receive-only connection:

  ```
  fd_hippi0=open ("/dev/hippi0", O_RDONLY);
  ioctl (fd_hippi#, HIPIOC_BIND_ULP, HIPPI_ULP_PH);
  ```

- For a transmit and receive connection:

  ```
  fd_hippi0=open ("/dev/hippi0", O_RDWR);
  ioctl (fd_hippi#, HIPIOC_BIND_ULP, HIPPI_ULP_PH);
  ```

Transmitting

For an application to transmit over its HIPPI-PH connection, a set (scenario) of calls must be made. The four possible scenarios are explained in the paragraphs that follow, and are illustrated in Figure 2-1. The order of the calls is unimportant except for the initial *write()* call, which actually allocates the resources and starts sending the data. Four functionality scenarios are supported. (See Table 1-1 for an overview of the four transmission functionality scenarios.)

Many of the *ioctl()* calls used in these scenarios write or set a value for a stored FDO parameter. These values are not cleared when a transmission completes, so prior settings can be reused with subsequent *write()* calls without resetting. All the calls should be made for the first transmission (since the device was opened) in order to initialize them to non-default values.

All application *write()*s must specify buffers that are 8-byte word-aligned, must use a maximum size as defined in “Maximum Size for write()s and read()s” on page 23, and must have a data bytecount that is a multiple of 8. (See the memalign(3C) reference page).

Notice the following:

1. When the `HIPIOCW_CONNECT` call is used, the HIPPI subsystem sets up a “permanent” connection. In contrast, when the `HIPIOCW_CONNECT` call is not used, the connection is disconnected as soon as the packet has been sent.
2. When the `HIPIOCW_START_PKT` call is used, many *write()*s may make up one packet. In contrast, when the `HIPIOCW_START_PKT` call is not used, one *write()* is a single packet.

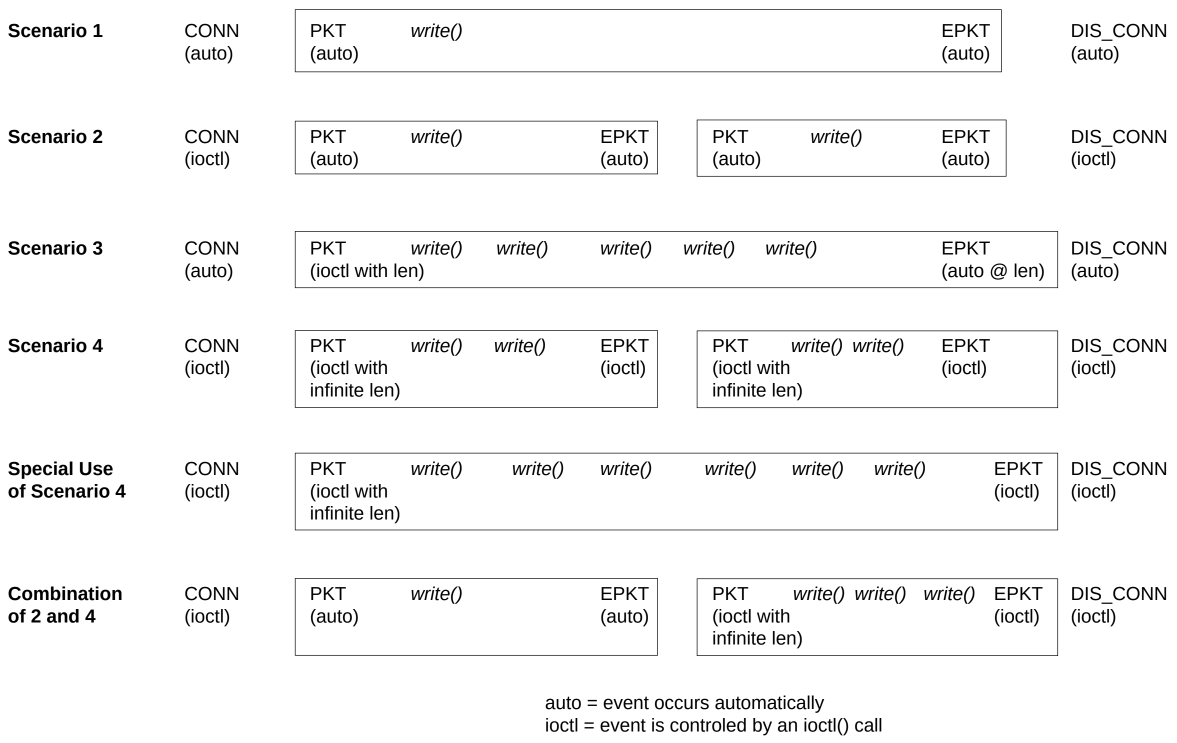


Figure 2-1 The Four Transmission Scenarios

Functionality Scenario 1

This scenario describes transmission of one small packet (less than *max_write_size*, as described in section "Maximum Size for write()s and read()s" on page 23) that uses one *write()* for the packet. The connection disconnects automatically when the packet has been completely sent.

The application makes an *ioctl()* call to specify the I-field, then makes the *write()* call. The maximum sized packet with this method is *max_write_size*. The packet and connection are both terminated when the data from the single *write()* call has completed.

```
ioctl (fd_hippi#, HIPIOCW_I, I-fieldValue);
/* I-field does not need to be reset for each pkt */
```

```
write (fd_hippi#, buffer, size);
```

```
/* PACKET line goes low (false) after one write */
/* connection is dropped after one write */
```

Functionality Scenario 2

This scenario describes transmission of many small packets (each no larger than *max_write_size*, as described in section "Maximum Size for write()s and read()s" on page 23) that use only one *write()* for each packet. The connection is kept open between packets.

The application makes an *ioctl()* call to specify the I-field for its "permanent" connection, then makes a *write()* call to send the first packet. The maximum-sized packet with this method is *max_write_size*. The **PACKET** signal is dropped automatically when the *write()* call completes. The connection, however, is not terminated, so the next packet can be another single-write in which the packet is terminated automatically, or a multiple-write (Functionality Scenario #4).

```
ioctl (fd_hippi#, HIPIOCW_CONNECT, I-fieldValue);
write (fd_hippi#, buffer, size);  /* first packet*/
/* PKT line goes low after one write. Connection is not dropped */
write (fd_hippi#, buffer, size);  /* second packet*/
/* PKT line goes low after one write. Connection is not dropped */

/* When application wants connection to be torn down, */
/* it tells the HIPPI subsystem to disconnect: */

ioctl (fd_hippi#, HIPIOCW_DISCONN);
```

Note: With HIPPI-Serial hardware, for the duration of this long-term connection, no packets are received from any source other than the endpoint (destination) of this long-term connection.

Functionality Scenario 3

This scenario describes transmission of a large packet that requires many *write()* calls and where the connection disconnects automatically when the packet has been completely sent.

The application makes an *ioctl()* call to specify the I-field, and one to define the size (bytecount) of the packet. It then makes the first *write()* call; subsequent *write()* calls are treated as part of the same packet until the bytecount is reached. The **PACKET** and **REQUEST** signals are automatically dropped after the specified number of bytes have been sent. This scheme allows an application to send very large packets. It also allows some data gathering on output. (Note, however, that if packets are formed using small-sized *write()* calls, performance degrades considerably.)

```
ioctl (fd_hippi#, HIPIOCW_I, I-fieldValue);
ioctl (fd_hippi#, HIPIOCW_SHBURST, firstburstsize);  /* only if size is changing */
ioctl (fd_hippi#, HIPIOCW_START_PKT, bytecount);
write (fd_hippi#, buffer, size);     /* buffer can point to FPheader + D1 data */
write (fd_hippi#, buffer, size);    /* size=only a part of the complete pkt*/
write (fd_hippi#, buffer, size);    /* max size for each write is max_write_size*/
write (fd_hippi#, buffer, size);
etc.

/* connection is dropped when packet is completely sent */
```

Functionality Scenario 4

This scenario describes transmission of many large packets that require many *write()* calls for each packet and where the connection is kept open.

The application makes an *ioctl()* call to specify the I-field for its "permanent" connection and one to specify the size (bytecount) of the packet. Each *write()* is treated as part of the same packet, until the bytecount is satisfied, at which time the packet is ended. When the application wants to terminate the connection, it makes an *ioctl()* call to disconnect.

```
ioctl (fd_hippi#, HIPIOCW_CONNECT, I-fieldValue);
ioctl (fd_hippi#, HIPIOCW_SHBURST, firstburstsize);  /* only if size is changing */
ioctl (fd_hippi#, HIPIOCW_START_PKT, bytecount);
write (fd_hippi#, buffer, size);  /* buffer can point to FPheader + D1 data */
write (fd_hippi#, buffer, size);  /* size=only a part of the complete pkt*/
write (fd_hippi#, buffer, size);  /* max size for each write is max_write_size */
etc.

/*when pkt's bytecount is complete, PKT line goes low*/
/*connection is not dropped*/

/* Optional: if the application wishes to start another packet,
/* it does this: */
ioctl (fd_hippi#, HIPIOCW_SHBURST, firstburstsize);  /* only if size is changing */
ioctl (fd_hippi#, HIPIOCW_START_PKT, bytecount);
...

/* When the application wants the connection to be torn down, */
/* it tells the HIPPI subsystem to disconnect */
ioctl (fd_hippi#, HIPIOCW_DISCONN);
```

Note: With HIPPI-Serial hardware, for the duration of this "permanent" connection, no packets are received from any source other than the endpoint (destination) of this "permanent" connection.

Special Use of Functionality Scenario 4

Scenario 4 can be used to send one infinite-sized packet on a long-term (“permanent”) connection.

The application makes an *ioctl()* call to specify the I-field for its “permanent” connection and one to specify the bytecount of the packet. The bytecount is specified as HIPPI_D2SIZE_INFINITY. All *write()* calls are then treated as one “infinite-sized” packet (that is, the **PACKET** signal is not deasserted), until the packet is specifically terminated by the application with a special *ioctl()* call. The connection is not dropped until the application disconnects it.

```
ioctl (fd_hippi#, HIPIOCW_CONNECT, I-fieldValue);
ioctl (fd_hippi#, HIPIOCW_SHBURST, firstburstsize);  /* only if size is changing */
ioctl (fd_hippi#, HIPIOCW_START_PKT, HIPPI_D2SIZE_INFINITY);
/*infinity=0xFFFFFFFF) */
write (fd_hippi#, buffer, size);   /*max size for each write is max_write_size*/
write (fd_hippi#, buffer, size);
etc.

/* Optional: if the application wishes to terminate this packet, it does this */
ioctl (fd_hippi#, HIPIOCW_END_PKT);

/* Optional: if the application wishes to start another packet,
/* it does one of these: */
ioctl (fd_hippi#, HIPIOCW_START_PKT, HIPPI_D2SIZE_INFINITY);
/* or */
ioctl (fd_hippi#, HIPIOCW_START_PKT, bytecount);

/* When the application wishes to tear down the connection, */
/* it does one of these: */
ioctl (fd_hippi#, HIPIOCW_END_PKT);
ioctl (fd_hippi#, HIPIOCW_DISCONN);
/* or */
ioctl (fd_hippi#, HIPIOCW_DISCONN);
```

Note: With HIPPI-Serial hardware, for the duration of this “permanent” connection, no packets are received from any source other than the endpoint (destination) of this “permanent” connection.

About Blocking on *write()*

IRIS HIPPI uses the first *write()* on each connection as the trigger for allocating/obtaining the hardware resources. For example, the **REQUEST** and **PACKET** signals are not asserted until the first *write()* call is made for a connection. This means that any number of *open()*s, `HIPIOCW_CONNECT`, and `HIPIOCW_START_PKT` calls can successfully be made in rapid succession for the same device; the HIPPI subsystem resources become unavailable (to other users) only when the first *write()* call is made for one of these open file descriptors. After the first successful *write()*, and as long as the connection for that *write()* remains active (that is, the **REQUEST** signal is asserted), *write()*s from other connections block. Once a connection is finished (that is, the **REQUEST** and **CONNECT** signals are deasserted), blocked *write()*s are serviced.

Receiving

In HIPPI-PH mode, all incoming data is accepted when the device file is opened for reading. The HIPPI subsystem does not reject any connection requests.

To retrieve its data, the application uses the calls below. All *read()*s retrieve sequentially received data. If an incoming packet contains an FP header and D1 data, the HIPPI subsystem does not interpret them and does not separate them from the D2 data, so the first *read()* may contain FP header, D1 data, and/or D2 data. To determine packet boundaries, the application can use an *ioctl()* call to retrieve the current offset (received bytecount) for the packet. When the returned value is 0, the next *read()* retrieves the first bytes from a new packet.

All application *read()*s must specify buffers that are 8-byte word-aligned, must use a maximum size as defined in "Maximum Size for write()s and read()s" on page 23, and must have a data bytecount that is a multiple of 8. (See the memalign(3C) reference page).

```
offset = ioctl (fd_hippi#, HIPIOCR_PKT_OFFSET);  /*when 0, nxt read is new pkt*/
read (fd_hippi#, buffer, size);  /*max size is max_write_size*/
offset = ioctl (fd_hippi#, HIPIOCR_PKT_OFFSET);  /*when 0, nxt read is new pkt*/
read (fd_hippi#, buffer, size);
etc.
```

When the application wishes to stop receiving data, it closes the file descriptor using the following call:

```
close (fd_hippi#);
```

Programming for the HIPPI-FP Access Method

This section describes how to program a module that uses the HIPPI Framing Protocol over the IRIS HIPPI subsystem. This API supports sharing the receive and/or transmit channels through the HIPPI subsystem with other upper layer protocols (ULPs). A sample program for this API is provided under the heading "HIPPI-FP Example" on page 81.

Note: Multiple applications can bind to a HIPPI file descriptor (for example, */dev/hippi1*) without affecting other applications that have opened the same device.

Using a HIPPI *ioctl()* call, the application "binds" itself to one ULPO and FDO. This action associates the application with one set of HIPPI information that is then used by the HIPPI subsystem whenever it services that application's read/write requests. The application specifies which ULPO by specifying the ULPO's 8-bit identifier.

It is important that the application open the HIPPI device with only the read/write flag settings that it needs. For example, if an application is not going to be doing *read()*s, it should set only the WRITE flag. When the READ flag is set, the HIPPI device is told to accept HIPPI packets on that ULP-id. All incoming packets for that ULP-id are accepted by the HIPPI device. The HIPPI subsystem holds each accepted packet until an application reads it. If no application consumes the incoming packets, the HIPPI device stalls for lack of buffer space.

Includes

The following files must be included in any program using the HIPPI API:

```
#include <sys/hippi.h>
```

Special Instructions

For maximum throughput, DMA between the HIPPI board and the host application occurs directly to/from user application space. Because of this, and the fact that the DMA component (ASIC) has a 64-bit interface, all application *read()*s and *write()*s must specify buffers that are 8-byte word-aligned, and the data bytecount must be a multiple of 8. (See the memalign(3C) reference page).

Opening and Binding to the Device

An application can open a HIPPI device (for example, */dev/hippi0* or */dev/hippi1*) for read-only, write-only, or read-and-write access. The acronym *fd_hippi#*, in the examples below, refers to the file descriptor for the opened HIPPI device.

Note: When the physical layer is IRIS HIPPI-Serial, simultaneous use of read-only and write-only files (for the same device) to different endpoints requires the use of a HIPPI switch between the source and the final destination.

To set up an application as a HIPPI-FP user, use one of the following sets of calls at the "beginning of time." Within the calls in these examples, the *ULP-id* is a positive number in the range 0-255 decimal (inclusive), where 4 is reserved for the IRIX 8802.2 Link Encapsulation (HIPPI-LE) ULP, and 6 and 7 are reserved for HIPPI-IPI3 implementations. Each ULP's identification must be unique among the ULPs that are bound to that HIPPI board.

- For a transmit-only connection:

  ```
  fd_hippi0=open ("/dev/hippi0", O_WRONLY);
  ioctl (fd_hippi#, HIPIOC_BIND_ULP, ULP-id);
  ```

- For a receive-only connection:

  ```
  fd_hippi0=open ("/dev/hippi0", O_RDONLY);
  ioctl (fd_hippi#, HIPIOC_BIND_ULP, ULP-id);
  ```

- For a transmit and receive connection:

  ```
  fd_hippi0=open ("/dev/hippi0", O_RDWR);
  ioctl (fd_hippi#, HIPIOC_BIND_ULP, ULP-id);
  ```

- For a monitoring connection to a HIPPI driver that has bypass functionality:

  ```
  fd_hippibp0=open ("/dev/hippibp0", O_RDONLY);
  ioctl (fd_hippibp#, HIPIOC_BIND_ULP, ULP-id);
  <only the HIPIOC_GET_BPSTATS is available>
  ```

Hint: The IRIS HIPPI API does not provide a timeout or interrupt for *read()*s or *write()*s that do not complete. If you want your program to be interrupted when one of these calls fails to complete in a certain length of time, use the *alarm* mechanism. (See the man page for alarm(2) reference page).

Transmitting

For a HIPPI-FP application to transmit over its HIPPI connection, one of the sets of calls documented below in the “Functionality Scenarios” must be made. The order of the calls is unimportant except for the initial *write()* call, which actually starts sending the data. Four functionality scenarios are supported. (See Table 1-1 for details on the four transmission functionality scenarios.)

Many of the *ioctl()* calls write or set a value for a stored ULPO or FDO parameter. These values are not cleared when a transmission completes, so prior settings can be reused with subsequent *write()* calls without resetting. All the calls should be made for the first transmission (since the device was opened) in order to initialize them to non-default values.

All application *write()*s must specify buffers that are 8-byte word-aligned, must use a maximum size as defined in “Maximum Size for write()s and read()s” on page 23, and must have a data bytecount that is a multiple of 8. (See the memalign(3C) reference page).

Notice the following:

1. When the `HIPIOCW_CONNECT` call is used, the HIPPI subsystem sets up a “permanent” connection. In contrast, when the `HIPIOCW_CONNECT` call is not used, the connection is disconnected as soon as the packet has been sent.
2. When the `HIPIOCW_START_PKT` call is used, many *write()*s may make up one packet. In contrast, when the `HIPIOCW_START_PKT` call is not used, one *write()* is a single packet.

Functionality Scenario 1

This scenario describes transmission of one small packet (less than *max_write_size*, as described on page 23), using one *write()* for the packet. The connection disconnects when the packet has been completely sent.

The application makes an *ioctl()* call to specify the I-field, then makes the *write()* call. The maximum sized packet with this method is *max_write_size*. The packet and connection are both terminated when the data from the single *write()* call has completed.

```
ioctl (fd_hippi#, HIPIOCW_I, I-fieldValue);
/* I-field does not need to be reset for each pkt */
ioctl (fd_hippi#, HIPIOCW_D1_SIZE, bytecount);     /* only if size is changing*/
write (fd_hippi#, buffer, size);    /*max size for each write is max_write_size*/
```

```
/* PKT line goes low (false) after one write */
/* connection is dropped after one write */
```

Functionality Scenario 2

This scenario describes transmission of many small packets (each no larger than *max_write_size*, as described on page 23) that use one *write()* for each packet. The connection is kept open between packets.

The application makes an *ioctl()* call to specify the I-field for its "permanent" connection, then makes a *write()* call to send the first packet. The maximum sized packet with this method is *max_write_size*. The **PACKET** signal is automatically dropped when the *write()* call completes. The connection, however, is not terminated, so the next packet can be another single-write or a multiple-write one (Functionality Scenario #4).

```
ioctl (fd_hippi#, HIPIOCW_CONNECT, I-fieldValue);
ioctl (fd_hippi#, HIPIOCW_D1_SIZE, bytecount);    /* only if size is changing*/
write (fd_hippi#, buffer, size);        /* first packet; max size = max_write_size*/
/* PKT line goes low after one write. Connection is not dropped */
write (fd_hippi#, buffer, size);        /* second packet; max size = max_write_size*/
/* PKT line goes low after one write. Connection is not dropped */

/* When application wants connection to be torn down, */
/* it tells the HIPPI subsystem to disconnect: */

ioctl (fd_hippi#, HIPIOCW_DISCONN);
```

Note: With HIPPI-Serial hardware or with IRIS HIPPI configured to support the IP suite of protocols, for the duration of this long-term connection, no packets are received from a source other than the endpoint (destination) of this long-term connection.

Functionality Scenario 3

This scenario describes the transmission of one large packet that requires many *write()* calls for the packet and where the connection disconnects when the packet has been completely sent.

The application makes an *ioctl()* call to specify the I-field, and another call to define the size (bytecount) of the packet. It then makes the first *write()* call; subsequent *write()* calls are treated as part of the same packet until the bytecount is reached. The **PACKET** and **REQUEST** signals are automatically dropped after the specified number of bytes have been sent. This scheme allows an application to send very large packets. It also allows some data gathering on output. (Note, however, that if packets are formed using small-sized *write()* calls, performance degrades considerably.)

```
ioctl (fd_hippi#, HIPIOCW_I, I-fieldValue);
ioctl (fd_hippi#, HIPIOCW_D1_SIZE, bytecount);    /* only if size is changing*/
ioctl (fd_hippi#, HIPIOCW_SHBURST, firstburstsize);  /*only if burst_1 is changing*/
ioctl (fd_hippi#, HIPIOCW_START_PKT, bytecount);
write (fd_hippi#, buffer, size);     /* buffer can include D1 data */
write (fd_hippi#, buffer, size);  /* size=only part of pkt and up to max_write_size*/
write (fd_hippi#, buffer, size);    /* max size for each write is max_write_size*/
write (fd_hippi#, buffer, size);
etc.

/* connection is dropped when pkt is completely sent */
```

Functionality Scenario 4

This scenario describes transmission of many large packets that require many *write()* calls for each packet and where the connection is kept open.

The application makes an *ioctl()* call to specify the I-field for its "permanent" connection, and another call to specify the size (bytecount) of the packet. Each *write()* is treated as part of the same packet, until the bytecount is satisfied, at which time the packet is ended. When the application wants to terminate the connection, it makes an *ioctl()* call to disconnect.

```
ioctl (fd_hippi#, HIPIOCW_CONNECT, I-fieldValue);
ioctl (fd_hippi#, HIPIOCW_SHBURST, firstburstsize);  /*only if burst_1 is changing*/
ioctl (fd_hippi#, HIPIOCW_D1_SIZE, bytecount);    /* only if size is changing*/
ioctl (fd_hippi#, HIPIOCW_START_PKT, bytecount);
write (fd_hippi#, buffer, size);  /* buffer can include D1 data */
write (fd_hippi#, buffer, size);  /* size=only a part of the complete pkt*/
write (fd_hippi#, buffer, size);  /* max size for each write is max_write_size */
etc.
/*when pkt completes, PKT line goes low*/
/*connection is not dropped*/

/* When the application wants the connection to be torn down, */
/* it tells the HIPPI subsystem to disconnect */

ioctl (fd_hippi#, HIPIOCW_DISCONN);
```

Special Use of Functionality Scenario 4

For an infinite-sized packet on a long-term (“permanent”) connection.

The application makes an *ioctl()* call to specify the I-field for its “permanent” connection, and another call to specify the bytecount of the packet. The bytecount is specified as HIPPI_D2SIZE_INFINITY. All *write()* calls are then treated as one “infinite-sized” packet (that is, the **PACKET** signal is not deasserted), until the connection is specifically disconnected by the application.

```
ioctl (fd_hippi#, HIPIOCW_CONNECT, I-fieldValue);
ioctl (fd_hippi#, HIPIOCW_SHBURST, firstburstsize);  /*only if burst_1 is changing*/
ioctl (fd_hippi#, HIPIOCW_D1_SIZE, bytecount);    /* only if size is changing*/
ioctl (fd_hippi#, HIPIOCW_START_PKT, HIPPI_D2SIZE_INFINITY);
/*infinity=0xFFFFFFFF) */
write (fd_hippi#, buffer, size);   /*max size for each write is max_write_size*/
write (fd_hippi#, buffer, size);
etc.
```

```
/* Optional: if the application wishes to terminate this packet
/* and start another without dropping the connection, it does this: */
```

```
ioctl (fd_hippi#, HIPIOCW_END_PKT);
ioctl (fd_hippi#, HIPIOCW_START_PKT, bytecount);
etc.
```

```
/* When the application wishes to tear down the connection */
ioctl (fd_hippi#, HIPIOCW_END_PKT);
ioctl (fd_hippi#, HIPIOCW_DISCONN);
```

Note: With HIPPI-Serial hardware or with IRIS HIPPI configured to support the IP suite of protocols, for the duration of this “permanent” connection, no packets are received from any source other than the endpoint (destination) of this “permanent” connection.

About Blocking on *write()*

IRIS HIPPI uses the first *write()* on each connection as the trigger for allocating/obtaining the hardware resources. For example, the **REQUEST** and **PACKET** signals are not asserted until the first *write()* call is made for a connection. This means that any number of *open()*s, `HIPIOCW_CONNECT`, and `HIPIOCW_START_PKT` calls can successfully be made in rapid succession for the same device; the HIPPI subsystem resources become unavailable (to other users) only when the first *write()* call is made for one of these open file descriptors. After the first successful *write()*, and as long as the connection for that *write()* remains active (that is, the **REQUEST** signal is asserted), *write()*s from other connections block.

Once a connection is finished (that is, the **REQUEST** and **CONNECT** signals are deasserted), blocked *write()*s are serviced.

Receiving

To receive data, the application uses the calls below. The first *read()* of a ULP's queue retrieves a packet's FP header and D1 area. Subsequent *read()* calls retrieve D2 data. When the `HIPIOCR_PKT_OFFSET` returns zero, the next *read()* will retrieve a new packet's header and D1 area.

All application *read()*s must specify buffers that are 8-byte word-aligned, must use a maximum size as defined in "Maximum Size for write()s and read()s" on page 23, and must have a data bytecount that is a multiple of 8. (See the memalign(3C) reference page).

```
read (fd_hippi#, buffer, size);  /* FPheader and D1 data */
read (fd_hippi#, buffer, size);  /* D2 data */
read (fd_hippi#, buffer, size);  /* D2 data */
offset = ioctl (fd_hippi#, HIPIOCR_PKT_OFFSET);  /*when 0, nxt read is new pkt*/
```

When the application wishes to stop receiving data, it closes the file descriptor using the following call:

```
close (fd_hippi#);
```

API Reference

This section describes the HIPPI *ioctl* calls that comprise the API to the IRIS HIPPI subsystem. These calls are defined in the *sys/hippi.h* file. Each application program that wants to use the services of the HIPPI connection uses these calls to define its ULPO and FDO values, to set up its connection(s), and to transmit or receive data. The API calls are listed in Table 2-1.

Table 2-1 IRIS HIPPI API Summary

Purpose	API Call	Page
Device Management:		
Bind an upper-layer application	`HIPIOC_BIND_ULP`	41
Enable/disable IRIS HIPPI board	`HIPPI_SETONOFF`	66

Table 2-1 (continued) IRIS HIPPI API Summary

Purpose	API Call	Page
Connection Management:		
Start/stop accepting connections	`HIPIOC_ACCEPT_FLAG`	40
Set timeout for source's connection	`HIPIOC_STIMEO`	53
Prepare to open a single-packet connection	`HIPIOCW_I`	62
Prepare to open a long-term connection	`HIPIOCW_CONNECT`	56
Terminate a long-term connection	`HIPIOCW_DISCONN`	59
Packet Control:		
Received packet's bytecount	`HIPIOCR_PKT_OFFSET`	55
Send a single-write packet	`After setting the I-field (with HIPIOCW_I), no additional call is necessary, other than the write().`	
Send a multiple-write packet	`HIPIOCW_START_PKT`	65
Define first burst of a multiple-write packet	`HIPIOCW_SHBURST`	63
Terminate a packet	`HIPIOCW_END_PKT`	60
Retrieve errors from failed *read()* and *write()* calls	`HIPIOCR_ERRS` `HIPIOCW_ERR`	54 61
Define HIPPI-FP Fields:		
Define D1 Area Size and set P-bit	`HIPIOCW_D1_SIZE`	58
Collect Statistics or Monitor Connection:		
Obtain standard statistics	`HIPIOC_GET_STATS`	44
Obtain bypass statistics	`HIPIOC_GET_BPSTATS`	42

HIPIOC_ACCEPT_FLAG

`HIPIOC_ACCEPT_FLAG` configures the HIPPI board to accept or refuse connection requests.

Note: After each execution of */usr/etc/hipcntl startup*, */etc/init.d/network*, or each restart of the system, the IRIS HIPPI software sets this flag ON (accept).

Usage

HIPPI device control for HIPPI-FP and HIPPI-PH.

`ioctl (`*fd_hippi#*`, HIPIOCW_ACCEPT_FLAG, `*value*`);`

The arg

The *value* is any non-zero value (including negative values) to accept connection requests, and 0 to reject connection requests.

Failures and Errors

This call fails for the following reasons:

- The IRIS HIPPI board is shutdown (for example, *hipcntl shutdown* or `HIPPI_SETONOFF` has been called).

HIPIOC_BIND_ULP

`HIPIOC_BIND_ULP` is used to bind an application's open file descriptor (*/dev/hippi0*, */dev/hippi1*, etc.) to a ULPO or FDO. If an application wishes to both transmit and receive, it can bind once to a read-and-write file descriptor, or it can make this call twice (once to a write-only file descriptor and once to a read-only one). This call prepares for a connection; it does not open the connection.

- When the HIPPI-FP access method is used, up to 32 different applications can be bound simultaneously.
- When the HIPPI-FP access method is used, up to eight different ULPOs can be bound to each HIPPI subsystem.
- When the HIPPI-PH method is used, only one ULPO (that being, HIPPI_ULP_PH) can be bound for the receive HIPPI channel and one FDO for the transmit channel.

Usage

Initialization of HIPPI-FP.

`ioctl (`*fd_hippi#*`, HIPIOC_BIND_ULP, `*ULP-id*`);`

Initialization of HIPPI-PH.

`ioctl (`*fd_hippi#*`, HIPIOC_BIND_ULP, HIPPI_ULP_PH);`

The arg

For HIPPI-FP, the `arg` is the identification for the ULPO or FDO that the application uses (range 0-255 decimal inclusive), where 4 is reserved for the IRIX 8802.2 Link Encapsulation (HIPPI-LE), 6 and 7 are reserved for HIPPI-IPI, 12 is reserved for the IRIS ST Protocol stack, and 144 is used (by default) by the optional IRIS Bypass module. Each ULPO/FDO implementation must have an identification, and each identification must be unique among the ULPOs and FDOs that are open (bound) for the particular HIPPI board (device).

For HIPPI-PH, the `arg` is HIPPI_ULP_PH.

Failures and Errors

This call fails for the following reasons:

- The maximum number of FDOs and ULPOs are already bound to the HIPPI device.
- A HIPPI-PH object (FDO or ULPO) is already bound for this type of access (read or write) to this file descriptor.

HIPIOC_GET_BPSTATS

`HIPIOC_GET_BPSTATS` is used to obtain statistics about the HIPPI bypass functionality.

Usage

Monitoring the HIPPI bypass functionality.

`ioctl (`*fd_hippibp#*`, HIPIOC_GET_BPSTATS, &hippibp_stats);`

The arg

The `arg` is a pointer to a `hippibp_stats` structure.

The `hippibp_stats` structure, from the *hippi.h* file, is provided below for reference:

```
typedef struct hippibp_stats {
/* BYPASS STATE */
  u_int   hst_bp_job_vec;         /* bypass job enable vector */
                                /* job 0 is in bit 31, job 7 bit 24*/
  u_int   hst_bp_ulp;             /* ulp used by bypass */

/* SOURCE STATISTICS */
  u_int   hst_s_bp_descs;         /* total bypass desc processed */
  u_int   hst_s_bp_packets;       /* total bypass packets sent */
  _uint64_t hst_s_bp_byte_count; /* total bypass bytes sent */

/* SOURCE ERRORS */
  /* descriptor errors */
  u_int   hst_s_bp_desc_hostx_err;      /* hostx is out of bounds*/
  u_int   hst_s_bp_desc_bufx_err;       /* buffer index out of bounds*/
  u_int   hst_s_bp_desc_opcode_err;     /* invalid opcode */
  u_int   hst_s_bp_desc_addr_err;       /* packet length + offset */
                                      /* would cross a page boundary*/
  u_int   hst_s_bp_resvd[6];           /* future expansion */

/* DESTINATION STATISTICS */
  u_int   hst_d_bp_descs;           /* total dest desc processed */
  u_int   hst_d_bp_packets;         /* total bypass packets received */
  __uint64_t hst_d_bp_byte_count;    /* total bypass bytes received */
/* DESTINATION ERRORS */
  /* descriptor errors */
  u_int   hst_d_bp_port_err;       /* port not enabled or port */
                                 /* or port too large */
  u_int   hst_d_bp_job_err;     /* job not enabled */
  u_int   hst_d_bp_no_pgs_err;   /* no longer used */
  u_int   hst_d_bp_bufx_err;    /* destination bufx not in bounds */
  u_int   hst_d_bp_auth_err;    /* received authentication did */
                               /* not match job authentication
                               /* for destination port */
  u_int   hst_d_bp_off_err;     /* offset plus packet length */
                               /* would cross a page boundary
                               /* or not aligned properly*/
  u_int   hst_d_bp_opcode_err;  /* received opcode was invalid */
  u_int   hst_d_bp_vers_err;    /* received version number invalid */
  u_int   hst_d_bp_seq_err;     /* sequence number for multi-pkt */
                               /* opcode was out of sequence */
  u_int   hst_d_bp_resvd[4];

} hippibp_stats_t;
```

Failures and Errors

This call fails for the following reasons:

- The driver is unable to copy the statistics from the board.
- The IRIS HIPPI board is shutdown (for example*, hipcntl shutdown* or `HIPPI_SETONOFF` has be called).

HIPIOC_GET_STATS

`HIPIOC_GET_STATS` is used to obtain statistics about the HIPPI board. The HIPPI product ships with a utility (*hipcntl*) for doing this kind of monitoring, so this *ioctl* call is not usually needed by customer-developed applications.

Usage

Monitoring of HIPPI-PH and HIPPI-FP devices and connections.

```
ioctl (fd_hippi#, HIPIOC_GET_STATS, &hippi_stats);
```

The arg

The `arg` is a pointer to a `hippi_stats` structure, described in Table 2-2. The structure is the same across all IRIS HIPPI implementations; however, the specific information varies from hardware to hardware. Refer to the table that is appropriate for your hardware.

Table 2-2 Information Retrieved by HIPIOC_GET_STATS

Field	Type	Description
hst_flags;	u_int	The value of bits 31-28 indicates the type of hardware: 0x0 = IRIS HIPPI HIO Mezzanine, 0x1 = IRIS HIPPI-Serial XIO Bits 27-0 contain flags that indicate the currently active states. See Table 2-3 or Table 2-4 for the specific flags associated with each hardware implementation.
		Source statistics.
hst_s_conns;	u_int	Total connections attempted by local SRC.
hst_s_packets;	u_int	Total packets sent by local SRC.
sf. data[14];	union u_int	Array containing either a *hip_p* (described in Table 2-3) or *hip_s* (described in Table 2-4) structure, depending on the specific hardware.
		Destination statistics.
hst_d_conns;	u_int	Total connections accepted.

Table 2-2 Information Retrieved by HIPIOC_GET_STATS

Field	Type	Description
hst_d_packets;	u_int	Total packets received.
df. data[14];	union u_int	Array containing either a *hip_p* (described in Table 2-3) or *hip_s* (described in Table 2-4) structure, depending on the specific hardware.

Table 2-3 Status Information Retrieved From Copper-based HIPPI HIO Mezzanine Board

Field	Value or Type	Description
Status flags:		
HST_FLAG_DSIC	0x0001	Local SRC sees inbound **INTERCONNECT** signal
HST_FLAG_SDIC	0x0002	Local DST sees inbound **INTERCONNECT** signal
HST_FLAG_DST_ACCEPT	0x0010	Local DST is accepting connections
HST_FLAG_DST_PKTIN	0x0020	Local DST: **PACKET** input signal is asserted
HST_FLAG_DST_REQIN	0x0040	DST: inbound **REQUEST** signal is asserted
HST_FLAG_SRC_REQOUT	0x0100	Local SRC: outbound **REQUEST** signal is asserted
HST_FLAG_SRC_CONIN	0x0200	Local SRC: **CONNECT** input signal is asserted
hip_p	struct	Source error counts
rejects;	u_int	connection attempts rejected
dm_seqerrs;	u_int	count of sequence errors from SRC's data state machine
cd_seqerrs;	u_int	count of invalid sequencing of inbound control signals at the SRC's connection state machine
cs_seqerrs;	u_int	count of sequence errors detected within SRC's connection state machine
dsic_lost;	u_int	inbound **INTERCONNECT** signal was deasserted before local SRC terminated connection

Table 2-3 (continued) Status Information Retrieved From Copper-based HIPPI HIO

Field	Value or Type	Description
timeo;	u_int	timed out connection attempts
connls;	u_int	connections dropped by other side
par_err;	u_int	source parity error
resvd[6];	u_int	reserved for future compatibility
hip_p	struct	Destination error counts
badulps;	u_int	packets dropped due to unknown ULP-id
ledrop;	u_int	HIPPI-LE packets dropped
llrc;	u_int	connections dropped due to LLRC error
par_err;	u_int	connections dropped due to parity errors
seq_err;	u_int	connections dropped due to sequence errors
sync;	u_int	synchronization errors
illbrst;	u_int	packets with illegal burst sizes
sdic_lost;	u_int	connections dropped due to loss of inbound **INTERCONNECT** signal
nullconn;	u_int	connections with zero packets
resvd[5];	u_int	reserved for future compatibility

Table 2-4 Status Information Retrieved From Fiber Optics-based HIPPI-Serial XIO Board

Field	Value or Type	Description
Status flags:		
HST_FLAG_LOOPBACK	0x0004	Board's internal loopback mode is enabled
HST_FLAG_DST_ACCEPT	0x0010	Local DST is accepting connections

Table 2-4 (continued) Status Information Retrieved From Fiber Optics-based HIPPI-Serial

Field	Value or Type	Description
HST_FLAG_DST_PKTIN	0x0020	Local DST: **PACKET** signal input is asserted
HST_FLAG_DST_REQIN	0x0040	Local DST: **REQUEST** signal input is asserted
HST_FLAG_SRC_REQOUT	0x0100	Local SRC: **REQUEST** signal is asserted
HST_FLAG_SRC_CONIN	0x0200	Local SRC: **CONNECT** input signal is asserted
HST_FLAG_DST_LNK_RDY	0x00010000	Local SERIAL DST: the on-board HIPPI processor is in its operational state (that is, it is in state 2 of the "link reset state machine"). This flag is absent (not set) only if the DST state machine transitions to state 0 or 1, caused by losing contact with the HIPPI-Serial (G-link) chip on the board or the fiber connection.
HST_FLAG_DST_FLAG_SYNC	0x00020000	Local SERIAL DST: the alternating flag within the incoming data frame is correct (that is, it is synchronized)
HST_FLAG_DST_OH8_SYNC	0x00040000	Local SERIAL DST: the OH8 (framing) overhead bit from the incoming data stream is correct
HST_FLAG_DST_SIG_DET	0x00080000	Local SERIAL DST: incoming signal on fiber is detected
hip_s	struct	Source error counts
rejects;	u_int	Connection **REQUEST**s that were rejected
resvd0[2];	u_int	reserved
glink_resets;	u_int	Times that the firmware reset both the HIPPI-Serial (G-link) chips. A reset occurs when the firmware believes one of the HIPPI-Serial (G-link) chips is not responding. However, a reset can be caused by faulty cabling, ODLs, or connectors since the firmware cannot identify the true cause for an unresponsive HIPPI-Serial portion of the board.

Table 2-4 (continued) Status Information Retrieved From Fiber Optics-based HIPPI-Serial

Field	Value or Type	Description
glink_err;	u_int	Count of times that the firmware fails to see any one of the following flags for more than half a second: DST_OH8_SYNC, DST_FLAG_SYNC, DST_LNK_RDY, or DST_SIG_DET. This event can be counted, at maximum, 50 times per second (at 25MHz).
timeo;	u_int	Count of connection attempts by SRC that timed out.
connls;	u_int	Count of connections made by SRC that were dropped by the other side.
par_err;	u_int	Count of SRC parity errors. This error indicates that a local parity error (for example, on the IRIS HIPPI board) resulted in the transmission of invalid data.
resvd1[4];	u_int	reserved
numbytes_hi;	u_int	Most significant digits for total count of bytes sent.
numbytes_lo;	u_int	Least significant digits for total count of bytes sent.
hip_s	struct	Destination error counts.
badulps;	u_int	Packets dropped due to unknown ULP-id.
ledrop;	u_int	HIPPI-LE packets dropped.
llrc;	u_int	Connections dropped due to LLRC errors.
par_err;	u_int	Connections dropped due to parity errors.

Table 2-4 (continued) Status Information Retrieved From Fiber Optics-based HIPPI-Serial

Field	Value or Type	Description
frame_state_err;	u_int	Count of framing (OH8 overhead bit) errors that occurred while **PACKET** signal was asserted or HIPPI state transition errors. Examples of state transition errors include: **no_REQUEST —> PACKET** **no_REQUEST —> BURST** **REQUEST —> BURST** **BURST —> REQUEST** **BURST —> no_REQUEST**
flag_err;	u_int	Count of data-frame alternating-flag-bit synchronizations that were lost while **PACKET** signal was asserted.
illbrst;	u_int	Packets with illegal burst sizes.
pkt_lnklost_err;	u_int	Count of packets that were aborted due to the DST_FLAG_SYNC, DST_OH8_SYNC, or DST_LNK_RDY flag becoming unset (not true) when the **PACKET** signal was asserted.
nullconn;	u_int	Connections with zero packets.
rdy_err;	u_int	Count of bursts received for which no **READY**s had been sent.
bad_pkt_st_err;	u_int	Number of packets that started improperly. For example, a sequence of **PACKET**-no_**BURST**- deasserted_**PACKET** (that is, a null packet), or a faulty FP packet (**BURST** followed by less than 12 bytes and a deasserted **PACKET** signal).
resvd;	u_int	reserved
numbytes_hi;	u_int	Most significant digits for total count of bytes received.
numbytes_lo;	u_int	Least significant digits for total count of bytes received.

The `hippi_stats` structure, from the *hippi.h* file, is provided below for reference.

```
typedef struct hippi_stats {
     u_int hst_flags;/* status flags */
/* Used by HIO™ board on Challenge™ or Onyx® systems only */
#define HST_FLAG_DSIC 0x0001           /* SRC sees IC */
#define HST_FLAG_SDIC 0x0002           /* DST sees IC */

/* Used only by HIPPI-Serial XIO™ board */
#define HST_FLAG_LOOPBACK 0x0004        /* internal loopback enabled */

/* HIPPI flags used by all types of hardwre */
#define HST_FLAG_DST_ACCEPT 0x0010          /* DST is accepting connections */
#define HST_FLAG_DST_PKTIN 0x0020           /* DST: PACKET input is high */
#define HST_FLAG_DST_REQIN 0x0040           /* DST: REQUEST input is high */
#define HST_FLAG_SRC_REQOUT 0x0100          /* SRC: REQUEST is asserted */
#define HST_FLAG_SRC_CONIN 0x0200           /* SRC: CONNECT input is high */

/* HIPPI-Serial flags used only with HIPPI-Serial */
#define HST_FLAG_DST_LNK_RDY 0x00010000
#define HST_FLAG_DST_FLAG_SYNC 0x00020000 /* SERIAL DST: alternating flg sync'd*/
#define HST_FLAG_DST_OH8_SYNC 0x00040000
#define HST_FLAG_DST_SIG_DET 0x00080000  /* SERIAL DST: signal detect at DST */

/* Error statistics are different for Hippi Parallel (HP) vs. Hippi Serial (HS).
*  In an effort to keep some binary compatibility, some fields
*  defined originally for Hippi Parallel are re-used for Hippi Serial.
*/
/* Source statistics */
     u_int hst_s_conns;             /* total connections attempted */
     u_int hst_s_packets;           /* total packets sent */

union {
  u_int     data[14];
  struct {
     u_int rejects;                /* connection attempts rejected */
     u_int dm_seqerrs;             /* data sm sequence error */
     u_int cd_seqerrs;             /* conn sm sequence error, dst */
     u_int cs_seqerrs;             /* conn sm sequence error, src */
     u_int dsic_lost;
     u_int timeo;                  /* timed out connection attempts*/
     u_int connls;                 /* connections dropped by other side*/
     u_int par_err;                /* source parity error */
     u_int   resvd[6];            /* reserved for future compatibility */
  } hip_p;                        /* parrallel hippi */
```

```
    struct {
       u_int   rejects;             /* connection attempts rejected */
       u_int   resvd0[2];           /* reserved for future compatibility */
       u_int   glink_resets;        /* times firmware reset glink */
       u_int   glink_err;           /* glink error count */
       u_int   timeo;              /* connection attempts timed out */
       u_int   connls;              /* connections dropped by other side */
       u_int   par_err;             /* source parity error */
       u_int   resvd1[4];           /* reserved for future compatibility */
       u_int   numbytes_hi;         /* number of bytes sent */
       u_int   numbytes_lo;        /* number of bytes sent */
    } hip_s;                       /* serial hippi */

  } sf;                            /* source format, system specific */


  /* Destination statistics */
      u_int hst_d_conns;            /* total connections accepted */
      u_int hst_d_packets;          /* total packets received */

    union {
      u_int       data[14];
      struct {

      u_int badulps;          /* pkts dropped due to unknown ULP*/
      u_int ledrop;           /* HIPPI-LE packets dropped */
      u_int llrc;             /* conns dropped due to llrc error */
      u_int par_err;          /* conns dropped due to parity err */
      u_int seq_err;          /* conns dropped due to sequence err */
      u_int sync;             /* sync errors */
      u_int illbrst;          /* packets with illegal burst sizes */
      u_int sdic_lost;        /* conns dropped due to sdic lost */
      u_int nullconn;         /* connections with zero packets */
      u_int resvd[5];         /* reserved for future compatibility*/
    } hip_p;
    struct {
       u_int   badulps;           /* packets dropped due to unknown ULP */
       u_int   ledrop;            /* HIPPI-LE packets dropped */
       u_int   llrc;              /* conns dropped due to llrc error */
       u_int   par_err;           /* conns dropped due to parity error */
       u_int   frame_state_err;    /* framing err or state transition err; eg: */
                                       * no request -> packet
                                       * no request -> burst
                                       * request -> burst
                                       * burst -> request
                                       * burst -> no request
       u_int   flag_err;          /* flag sync lost during packet */
       u_int   illbrst;           /* packets with illegal burst sizes */
       u_int   pkt_lnklost_err;   /* lost linkready when PACKET asserted */
       u_int   nullconn;          /* connections with zero packets */
       u_int   rdy_err;           /* data received when no readys sent */
       u_int   bad_pkt_st_err;     /* packet got off to a bad start */
       u_int   resvd;
       u_int   numbytes_hi;        /* number bytes received */
       u_int   numbytes_lo;        /* number bytes received */
       } hip_s;

    } df;                         /* destination format, system specific */

  } hippi_stats_t;
```

Failures and Errors

This call fails for the following reasons:

- The driver is unable to copy the statistics from the board.
- The IRIS HIPPI board is shutdown (for example, *hipcntl shutdown* or `HIPPI_SETONOFF` has be called).

HIPIOC_STIMEO

`HIPIOC_STIMEO` sets the period of time for which the IRIS HIPPI source channel waits for a **CONNECT** or **READY** signal before aborting the connection. With the Challenge/Onyx HIO board, the granularity for this timeout is 250 milliseconds; IRIS HIPPI rounds the user-specified value to the nearest 250 millisecond interval. With the Origin/Onyx2 XIO board, the granularity is 1 millisecond.

Usage

Transmission for HIPPI-FP and HIPPI-PH.

`ioctl (`*fd_hippi#*`, HIPIOC_STIMEO, `*milliseconds*`);`

The arg

The range of valid values for milliseconds is 1 to FFFFFFFF inclusive (hexadecimal notation).

Failures and Errors

This call fails for the following reasons:

- IRIS HIPPI board is shutdown (for example, *hipcntl shutdown* or `HIPPI_SETONOFF` has be called).
- The value used for the *milliseconds* argument is out of range.

HIPIOCR_ERRS

`HIPIOCR_ERRS` returns the error status from the last *read()* call for the indicated file descriptor.

Usage

Error monitoring for reception with HIPPI-FP and HIPPI-PH.

`error = ioctl (`*fd_hippi#*`, HIPIOCR_ERRS);`

The arg

There is no arg for this call.

Returned Value

The returned error is a 6-bit vector indicating the errors that occurred on the last *read()*, as summarized in Table 2-5.

Table 2-5 Errors for Failed read() Calls

Bit Position	Hex Mask	Error
0	0x01	HIP_DSTERR_PARITY: Destination parity error.
1	0x02	HIP_DSTERR_LLRC: Destination LLRC error.
2	0x04	HIP_DSTERR_SEQ: Destination sequence error. Causes the SDIC lost error also.
3	0x08	HIP_DSTERR_SYNC: Destination synchronization error.
4	0x10	HIP_DSTERR_ILBURST: Destination illegal burst error.
5	0x20	HIP_DSTERR_SDIC: Destination **SDIC** lost error.

Failures and Errors

This call fails for the following reasons:

- The application is not bound.
- If the returned `error` is a negative value, then an error occurred while making the *ioctl()* call, and none of the bits should be interpreted.

HIPIOCR_PKT_OFFSET

`HIPIOCR_PKT_OFFSET` retrieves the offset for the packet being received.

Usage

Reception for HIPPI-FP and HIPPI-PH.

```
offset = ioctl (fd_hippi#, HIPIOCR_PKT_OFFSET);
```

The arg

There is no arg for this call.

Returned Value

The returned *offset* is an integer indicating the current offset (number of bytes received so far) for the packet in the next *read()*. When the *offset* is 0, the next *read()* starts a new packet.

When the returned *offset* reaches 0x7FFFFFFF, the counter sticks (that is, does not count any higher and does not roll over to zero). However, the counter will again return true count values when the next packet arrives.

Failures and Errors

This call fails for the following reasons:

- The application is not bound.
- The file descriptor has not been opened for reading by this application.

HIPIOCW_CONNECT

`HIPIOCW_CONNECT` causes a long-term (many-packet) connection to be established with the next *write()*. The argument sets the value for the I-field that will be used on the connection request. Once this call has been made, the HIPPI-subsystem sets up a connection with the next *write()* call, and does not tear the connection down until the `HIPIOCW_DISCONN` call is invoked. Multiple applications can call `HIPIOCW_CONNECT` for the same device successfully; however, once one application does a *write()*, the HIPPI subsystem's resources are unavailable for other applications' *write()s*.

Note: For single-packet connections, use `HIPIOCW_I`.

Usage

Transmission for HIPPI-FP and HIPPI-PH.

`ioctl (`*fd_hippi#*`, HIPIOCW_CONNECT, `*I-fieldValue*`);`

The arg

The *I-fieldValue* is a 32-bit number used as the I-field. IRIS HIPPI does not verify, alter, or interpret the I-field value.

For a HIPPI-SC compliant I-field, bit 31 of the I-field must be set to 0 and the remaining bits (30:0) must be partitioned into fields, as summarized in Table 2-6. (For more details about the I-field, see Figure A-2.)

"Locally-administered" schemes are legal and supported. For a locally-administered scheme, bit 31 must be set to 1 and bits 30:0 can be set to comply with any locally-defined protocol or can be set to zero (for example, I-field value = 80000000 hex).

Note: Using a "locally-administered" I-field severely limits interoperability, especially in the areas of routing and HIPPI switch control.

Table 2-6 IRIS HIPPI Support for Fields in I-Field

	Valid Values for IRIS HIPPI (binary)		
Bits in I-field	**Locally-Defined**	**HIPPI-SC Compliant**	**Comments**
31	1	0	Both I-field formats (local and HIPPI-SC) are supported.
30:29	anything	anything	
28	anything	0	IRIS HIPPI hardware currently supports only 800 Mbits/second. It is the application's responsibility to set this correctly.
27	anything	0 / 1	It is the application's responsibility to set the direction bit correctly.
26:25	anything	00 / 01/11	All addressing schemes are supported.
24	anything	0 / 1	It is the application's responsibility to set the camp-on bit correctly.
23:0	anything	anything	All addressing schemes are supported.

Failures and Errors

This call fails for the following reasons:

- The application is not bound.
- The application has not disconnected from a long-term connection that was established with `HIPIOCW_CONNECT` prior to this request. (If no data has ever been sent, the connection is not necessarily open at the physical layer.)

HIPIOCW_D1_SIZE

`HIPIOCW_D1_SIZE` is used to set the size of the D1_Area and set the P-bit in a HIPPI-FP FDO. This call specifies a D1 area size that is placed in the FP header of all subsequently transmitted packets.

This call has the following characteristics:

- The size must be zero or a multiple of 8.
- When the D1 area size is greater than 0, the P-bit in the FP header is set to 1.

To send its D1 data, the application can concatenate the D2 data to the D1 data so that the first burst contains both kinds of data, or it can use `HIPIOCW_SHBURST` to place only the FP header and the D1 data in the first burst. The HIPPI subsystem uses all of this information to correctly calculate the values for the FP header, as explained in "How HIPPI Protocol Items Are Handled With the HIPPI-FP Access Method" on page 18.

Usage

Transmission for HIPPI-FP.

`ioctl (`*fd_hippi#*`, HIPIOCW_D1_SIZE, `*bytecount*`);`

The arg

The *bytecount* is the size in bytes to be placed in the D1_Area_Size field of the HIPPI-FP header of subsequent packets. Valid sizes fall within the range of 0-1016 (decimal), inclusive, and must be evenly divisible by 8.

Failures and Errors

This call fails for the following reasons:

- The application is not bound.
- The bound FDO is HIPPI-PH.
- The *bytecount* is not valid.

HIPIOCW_DISCONN

`HIPIOCW_DISCONN` is used for terminating a permanent connection that was opened with the `HIPIOCW_CONNECT` call. This call causes the HIPPI subsystem to tear down the connection immediately.

Note: To terminate a packet without tearing down the connection, use HIPIOCW_END_PKT.

Usage

Transmission for HIPPI-FP and HIPPI-PH.

```
ioctl (fd_hippi#, HIPIOCW_DISCONN);
```

The arg

There is no arg for this call.

Failures and Errors

This call fails for the following reasons:

- The application is not bound.
- No long-term (permanent) connection has been set up; there is nothing to disconnect.

HIPIOCW_END_PKT

`HIPIOCW_END_PKT` terminates the current packet (that is, causes the HIPPI subsystem to drive the **PACKET** signal false). This call is required only when the packet length was specified as infinite.

Usage

Transmission for HIPPI-FP and HIPPI-PH.

`ioctl (`*fd_hippi#*`, HIPIOCW_END_PKT);`

The arg

There is no arg for this call.

Failures and Errors

This call fails for the following reasons:

- The application is not bound.
- There is no inprogress packet; that is, this call has not been preceded by a `HIPIOCW_START_PKT` call.

HIPIOCW_ERR

`HIPIOCW_ERR` returns the error status from the last *write()* call for the indicated file descriptor.

Usage

Error monitoring for transmission with HIPPI-FP and HIPPI-PH.

```
error = ioctl (fd_hippi#, HIPIOCW_ERR);
```

The arg

There is no arg for this call.

Returned Value

The returned error is an integer, as summarized in Table 2-7.

Table 2-7 Errors for Failed write() Calls

Hex Value	Error
0	HIP_SRCERR_NONE: No error occurred on last *write()*.
1	HIP_SRCERR_SEQ: Source sequence error.
2	HIP_SRCERR_DSIC: Source lost **DSIC** error.
3	HIP_SRCERR_TIMEO: Source timed out connection.
4	HIP_SRCERR_CONNLS: Source lost inbound **CONNECT** signal during transmission.
5	HIP_SRCERR_REJ: Source's connection **REQUEST** was rejected.
6	HIP_SRCERR_SHUT: Interface is shut down.

Failures and Errors

This call fails for the following reasons:

- The application is not bound.

HIPIOCW_I

`HIPIOCW_I` prepares the HIPPI subsystem to set up a single-packet connection. The command specifies a new value for the HIPPI I-field in the application's FDO. (The I-field is also known as CCI.) The HIPPI subsystem uses this value as the I-field for all subsequent connection requests (that is, at each *write* call), until `HIPIOCW_I` is called again. The HIPPI subsystem does not alter or interpret the I-field contents. The HIPPI subsystem drops each connection as soon as the data from the *write()* call is sent.

Note: For a long-term (many-packet) connection, use the `HIPIOCW_CONNECT` call.

Usage

Transmission for HIPPI-PH and HIPPI-FP.

`ioctl (`*fd_hippi#*`, HIPIOCW_I, `*I-fieldValue*`);`

The arg

The *I-fieldValue* is a 32-bit number. IRIS HIPPI does not verify, alter, or interpret the I-field value.

For a HIPPI-SC compliant I-field, bit 31 of the I-field must be set to 0 and the remaining bits (30:0) must be partitioned into fields, as summarized in Table 2-6. (For more details about the I-field, see Figure A-2.)

"Locally-administered" schemes are legal and supported. For a locally-administered scheme, bit 31 must be set to 1 and bits 30:0 can be set to comply with any locally-defined protocol or can be set to zero (for example, I-field value = 80000000 hex).

Note: Using a "locally-administered" I-field severely limits interoperability, especially in the areas of routing and HIPPI switch control.

Failures and Errors

This call fails for the following reasons:

- The application is not bound.

HIPIOCW_SHBURST

`HIPIOCW_SHBURST` defines the size of the first burst for all subsequent packets. The size may be shorter than a standard burst, or full-sized. The IRIS HIPPI subsystem's functionality is slightly different for HIPPI-PH and HIPPI-FP applications, as explained below.

Note: If the first burst is short, it is the responsibility of the application to pad out the D2 data to a multiple of 256 words, so that all the non-first bursts are full-sized. The IRIS HIPPI software does not verify the data size nor pad the final burst.

For a HIPPI-PH application, the call causes the HIPPI subsystem to "break off" the indicated number of bytes from the data provided by the first *write()* call, and send these bytes as the first burst. When the desired first burst consists of 256 words, it is not necessary to make this call. When `HIPIOCW_SHBURST` is called with a *bytecount* of 0, the IRIS HIPPI subsystem creates standard-sized first bursts.

For a HIPPI-FP application, the call causes the IRIS HIPPI subsystem to create a first burst that contains only the FP header and D1 data and to set the B-bit in the FP header. When `HIPIOCW_SHBURST` is called with the following bytecounts, the first burst is created as described:

- With a *bytecount* of 0, the first burst is standard-sized and contains the FP header and 1016 bytes of data from the *write()* call. The B-bit is set to 0.
- With a *bytecount* of 8, the first burst is short and contains only the FP header. The B-bit is set to 1.
- When the *bytecount* is larger than 8 but smaller than 1024, the first burst is short; it contains 8 bytes of FP header and [*bytecount* minus 8] of D1 data. The HIPPI subsystem "breaks off" the D1 data bytes from the data provided with the first *write()* call. The B-bit is set to 1.

  Note: When D1 data is included, it is the application's responsibility to also call `HIPIOCW_D1_SIZE` to ensure a properly filled-in FP header.
- When the *bytecount* is 1024, the first burst is standard-sized; it contains 8 bytes of FP header and 1016 bytes of D1 data. The HIPPI subsystem "breaks off" the D1 data bytes from the data provided with the first *write()* call. The B-bit is set to 1.

Usage

Transmission of multiple-write packets for HIPPI-FP and HIPPI-PH. Once called, the setting applies to all packets, until called again.

`ioctl (`*fd_hippi#*`, HIPIOCW_SHBURST, `*bytecount*`);`

The arg

For HIPPI-PH, the *bytecount* can be any value from 0-1024 decimal (inclusive) that is evenly divisible by 8.

For HIPPI-FP, the *bytecount* can be any value from 0-1024 decimal (inclusive) that is evenly divisible by 8. The minimum *bytecount* for a short first burst is 8 (that is, large enough to include the FP header) and, if `HIPIOCW_D1_SIZE` has been called, it must be [8 + D1_Area_Size].

Failures and Errors

This call fails for the following reasons:

- The application is not bound.
- The packet has not been setup as a multiple-write packet, using the `HIPIOCW_START_PKT` command.
- The size of *bytecount* is not valid.

HIPIOCW_START_PKT

`HIPIOCW_START_PKT` controls the HIPPI subsystem's **PACKET** signal. The signal is held high (**PACKET** = true) for the bytecount provided in the call's argument (or for HIPPI-FP, the bytecount plus 8, thus including the FP header). The bytecount should be so large that a number of *write()* calls are required to send it. This call must be made for each multiple-write packet.

Usage

Transmission for HIPPI-FP and HIPPI-PH.

`ioctl (`*fd_hippi#*`, HIPIOCW_START_PKT, `*bytecount*`);`

The arg

The *bytecount* is either the actual bytecount of the D1 and D2 areas of the packet or a value indicating "infinite." (Infinite packets are supported only when the connection is permanent or long-term.)

- The range of valid values for an actual *bytecount* is multiples of 8 between 0 and 0xFFFFFFF8 hexadecimal, inclusive. The maximum actual length for any packet is 4 gigabytes less 8 bytes.
- An "infinite" or indeterminate packet is defined by a bytecount of `HIPPI_D2SIZE_INFINITY` (which is 0xFFFFFFFF).

Failures and Errors

This call fails for the following reasons:

- The application is not bound.
- A packet is currently in progress. For example, for an infinite packet, the `HIPIOCW_END_PKT` call has not terminated the current packet.

HIPPI_SETONOFF

`HIPPI_SETONOFF` does shutdown and bringup of the IRIS HIPPI board. ON (bringup) initializes everything on the board, leaving the board in the UP state. OFF (shutdown) completes inprogress work with errors, turns everything off, resets the onboard CPU, and transitions to the DOWN state. This command is intended for administration and maintenance purposes only; hence, it is only available to superuser (root).

Usage

Board shutdown and bringup. Only available to superuser (root).

`ioctl (`*fd_hippi#*`, HIPPI_SETONOFF, arg);`

The arg

The `arg` is 1 for ON (bringup) and 0 for OFF (shutdown). Multiple, sequential calls for OFF while the board is down results in multiple resets of the board's CPU, as described in Table 2-8.

Table 2-8 Actions Caused by HIPPI_SETONOFF

	Board = DOWN	**Board = UP**
Command = OFF (0)	reset onboard CPU	shutdown, which includes CPU reset
Command = ON (1)	bringup	error

Failures and Errors

This call fails for the following reasons:

- The application is not superuser (root).
- The file descriptor is bound. Closing the file descriptor will unbind it.
- For ON, there are other open (cloned) file descriptors that must be closed before the board can be brought up.

Chapter 3

Programming Notes for Sockets-based API

This chapter describes how to interface an application to the IRIS HIPPI subsystem through IRIX sockets in order to use the services of the IP network stack.

Note: For complete instructions for the IRIX socket API, see Chapter 2 of the InSight (online) document *IRIX Network Programmer's Guide,* and the man pages for socket(), bind(), getsocketname(), connect(), listen(), accept(), send(), recv(), etc.

Maximum Size for *write()*s and *read()*s

The maximum size for a buffer used within a call that transmits and retrieves data (for example, *write()* and *read()* calls) depends on the installed hardware, as summarized below.

- Copper-based HIO board (HIPPI-800): *max_bufsize* = 2 megabytes
- Fiber-optics XIO board (HIPPI-Serial): *max_bufsize* = 16 megabytes

To create a program that will behave correctly, regardless of which hardware is installed, include code that retrieves status from the hardware to determine the hardware type and sets the *max_bufsize*, as illustrated in the following example:

```
#include <sys/hippi.h>
/* verify which platform  */
if ( ioctl( fd_from_open, HIPIOC_GET_STATS, &hipstats ) < 0 ) {
    if ( errno == ENODEV ) fprintf(stderr,
        "%s: HIPPI board is down\n", argv[0]), exit(1);
    else
        perror("hipcntl: ioctl HIPIOC_GET_STATS failed"),exit(1);
    }

/* set the maximum length for reads and writes */
/* HST_FORMAT_ID_MASK and HST_XIO are defined in hippi.h */
max_bufsize = ( (hipstats.hst_flags & HST_FORMAT_ID_MASK) == HST_XIO)
? (0x1000000 + 8) : (0x200000 + 8);
```

Note: See Chapter 2 for complete details on the usage of IRIS HIPPI *ioctl()* calls (for example, "HIPIOC_GET_STATS" on page 44).

Programming for IRIX Sockets

This section describes how to program a module that uses IRIX' BSD-based sockets. Programs use this API to utilize the services of the network stack for the Internet Protocol Suite (IP). This API supports sharing the receive and/or transmit channels through the IRIS HIPPI subsystem with other upper layer protocols (such as HIPPI-FP).

Includes

The following files must be included in any program using the IRIX sockets API:

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

Special Instructions

For maximum throughput, DMA between the IRIS HIPPI board and the host application occurs directly to/from user application space. Due to the design of this DMA engine, the following rules must be followed by all application *read()*s and *write()*s within this API:

- The specified buffers must be 8-byte-word-aligned.
- The data bytecount must be a multiple of 8.

See memalign(3C) for a method of allocating 8-byte aligned memory.

Opening a Connection to IP Stack

To create a connection to the IP stack, use a series of calls similar to this:

```
s = socket (AF_INET, SOCK_DGRAM, 0);
bind(s, &client_addr, sizeof(client_addr) );
connect(s, &serv_addr, sizeof(serv_addr) );
```

where *client_addr* corresponds to one of the available and configured IRIS HIPPI network interfaces (for example, *hip0* configured with IP address 192.56.142.5).

Transmitting

For an application to transmit over its IRIS HIPPI socket, it simply does a *write()*, or any of its variations (such as *send()*, *sendmsg()*, or *writev()*.) See “Maximum Size for write()s and read()s” on page 67 and “Special Instructions” on page 68 for additional details.

Receiving

To retrieve data, the application simply does a *read()*, or any of its variations (such as *recv()*, *recvmsg()*, or *readv()*.) All retrievals return data that has been put into message-correct order by the network stack. See “Maximum Size for write()s and read()s” on page 67 and “Special Instructions” on page 68 for additional details.

Hint: The IRIS HIPPI subsystem holds each accepted packet until an application reads it. If no application consumes the incoming packets, the HIPPI device will eventually stall for lack of buffer space.

Closing the Connection

When the application wishes to stop sending/receiving data, it closes the connection using a *close()* or *shutdown()* call.

Important HIPPI Concepts

I-Field

The format for the standard HIPPI I-field (also referred to as CCI) that accompanies each connection request is shown in Figure A-1. The seven fields are described in Table A-1.

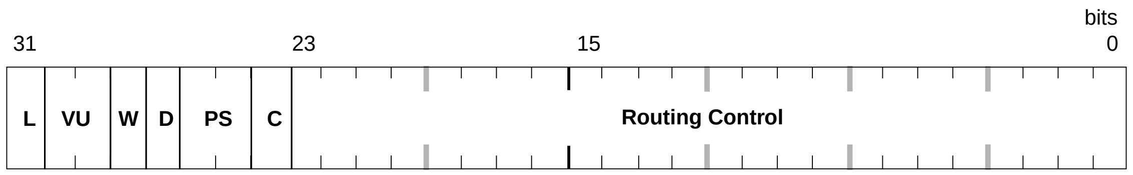


Figure A-1 I-Field Format

Table A-1 Fields of the HIPPI I-Field

Field	**Bits**	**Description**
L	31	Local or Standard Format: 0=bits 30:0 of I-field conform to the usage described in this table 1=bits 30:0 are implemented in conformance to a private (locally-defined) protocol
VU	30:29	Vendor Unique Bits: Vendors of end-system HIPPI equipment may use these bits for any purpose. Switches do not alter or interpret these bits.
W	28	Width: 0=the data bus of the transmitting (source) HIPPI is 32 bits wide for 800 megabits/second 1=source’s data bus is 64 bits wide for 1600 megabits/second

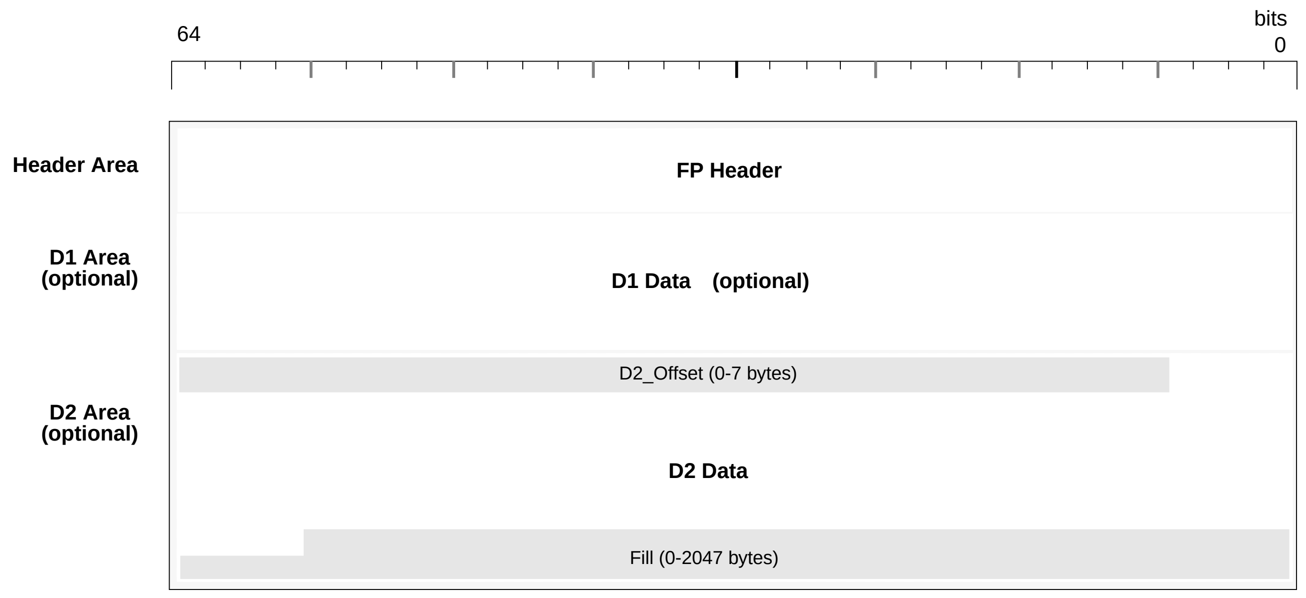
Table A-1 (continued) Fields of the HIPPI I-Field

Field	Bits	Description
D	27	Direction: 0=least significant bits of Routing Control field contain the next address for switch to use 1=most significant bits of Routing Control field contain the next address for switch to use
PS	26:25	Path Selection: 00=source routing 01=Routing Control field contains logical addresses. Switch must select first route from a list of routes. 10=reserved 11=Routing Control field contains logical addresses. Switch selects route.
C	24	Camp-on: 0=switch does not retry if connection is rejected 1=switch continues trying to establish a connection until the source aborts the connection request
Routing Control	23:0	Routing Address: This field may contain source routing addresses or logical addresses, as indicated by the PS field. For source routing, the field contains a concatenated list of switch addresses that, when followed, lead to the destination. For logical addressing, the field contains two 12-bit addresses (destination and source) that are used by the intermediate switches to select a route from a table.

HIPPI-FP Packet

Each HIPPI packet using the HIPPI Framing Protocol (HIPPI-FP) has a required 64-bit segment called the FP header, and two optional segments called D1 Area and D2 Area, as illustrated in Figure A-2. The D1 area is intended for communicating control (D1) information. It can also be used for padding out the first burst in order to position the user (D2) data in the second burst. The D2 area contains user/application data. The size of the D1 area is defined within the FP header. The size of the D2 area is not specifically defined, but is implicit due to the protocol definition. The D2 area consists of D2 data and possibly an offset and filler. The D2 offset and D2 data are defined in the FP header. The size of the filler can be calculated by rounding up to the next 64-bit word boundary, because the D2 area is required to be an integral number of 64-bit words.

The format for the HIPPI-FP packet is as shown in Figure A-2. The FP header consists of seven fields, shown in Figure A-3 and described in Table A-2.



NOTE: The size of each included area must be an integral number of 64-bit words.
The first word of each area must be 8-byte aligned.

Figure A-2 Packet Format for HIPPI Framing Protocol

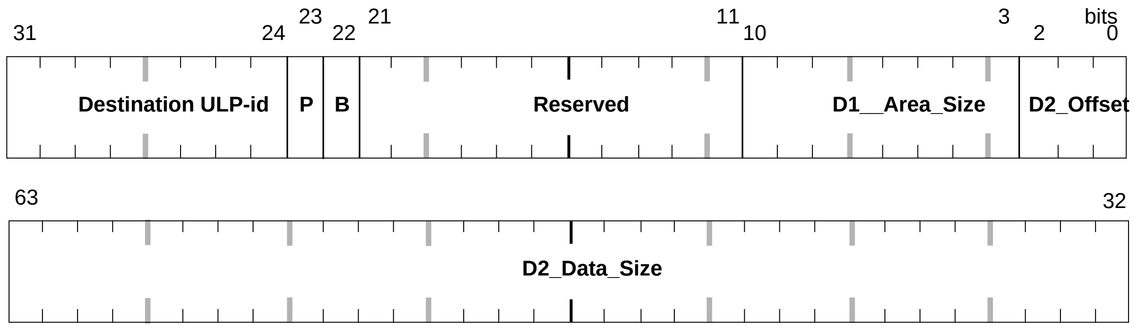


Figure A-3 FP Header Format

Table A-2 Fields of FP Header

Field	Bits	Range of Values (in hex)	Description
D2 Data Size	63:32	0 - FFFFFFFF	Number of bytes of D2 data in this packet not counting the D2 offset nor the D2 fill. A size of FFFFFFFF (hexadecimal) indicates a packet of unknown, indeterminate, or “infinite” length.
Dest ULP-id	31:24	0 - FF	The upper layer identification number for the destination.
P	23	0 / 1	Present: 0=there is no D1_Area in this packet 1=there is D1 data in the D1_Area of this packet
B	22	0 / 1	Burst Boundary: 0=D2 data starts before beginning of second burst of this packet 1=D2 data starts at beginning of second burst of this packet
Reserved	21:11	000	Must be zero.

Table A-2 (continued) Fields of FP Header

Field	Bits	Range of Values (in hex)	Description
D1 Area Size	10:3	0 - 7F	The number of 64-bit words in the D1 Area. The area does not necessarily contain valid D1 data; the area may be defined for padding purposes only.
D2 Offset	2:0	0 - 7	The number of bytes between the last byte of the D1 Area and the first byte of D2 data.

Appendix B

Sample Programs

HIPPI-PH Example

The following code is an example of a ULP using the HIPPI-PH API.

```
#include <stdio.h>
#include <sys/errno.h>
#include <sys/types.h>
#include <fcntl.h>
#include <stdarg.h>
#include <sys/hippi.h>

#define DEFAULT_ULP         0x80
#define MAX_DEV_NAME_LEN    1024
#define MAX_WRITE           0x1000000 /* 16 meg */

extern int errno;

static char *usage =
“Usage: fptest [-D<device_name] [-I<ifield>] [-U<ulp>] [-n<num
packets] [-l<pkt length>] [-s] [-r]\n”
“-D: Device name. Default is /dev/hippi0.\n”
“-I: Ifield. Default is 0, should be set every time.\n”
“-U: ULP. Default is 0x80.\n”
“-n: Number of packets to send. 0 is send forever. Default is 64.\n”
“-l: Length of packet in bytes. Default is 1024.\n”
“-s: Send mode. This is the default.\n”
“-r: Receive mode.\n”;


static char *src_errs[] = {
    “no error”,
    “SRC sequence error”,
    “SRC lost DSIC”,
    “SRC timeout”,
    “SRC lost connect”,
    “SRC connect rejection”,
    “HIPPI interface shutdown”,
    “SRC parity error”
};

static char *dst_errs[] = {
    “DST Parity Error”,
    “DST LLRC Error”,
    “DST Sequence Error”,
    “DST Sync Error”,
    “DST Illegal Burst”,
    “DST SDIC loss”,
    “DST Framing Error”,
    “DST lost linkrdy in packet”,
    “DST No Packet Signal in Connection”,
    “DST Data Received with no READY’s sent”,
    “DST No Burst in Packet”,
    “DST Illegal State Transition”
};

int send = 1, fd = -1;

void
die(char * str, ...) {
    va_list ap;

    va_start (ap, str);
    vfprintf(stderr, str, ap);
    va_end (ap);

    exit(1);
}

void
phiperr(int retval) {

    if (retval < 0 && errno == EIO) {
        retval = ioctl(fd, HIPIOCW_ERR);
        if (retval < 0)
            perror(“HIPIOCW_ERR ioctl failed\n”);
        else
            fprintf(stderr,
                    “HIPIOCW_ERR: %d: %s\n”,retval,
                    (send) ? src_errs[retval] : dst_errs[retval]);
    }
    else
```

```
        perror(“READ/WRITE ERROR”);

    exit(1);
}

main(int argc, char **argv)
{
    extern char *optarg;
    extern int optind;
    char *buf;
    char device_name[MAX_DEV_NAME_LEN] = “/dev/hippi0”;
    uint32_t num_pkts = 64,
             pkt_len = 0x400,
             ifield = 0,
             ulp = DEFAULT_ULP;
    int i, c;


    if(argc > 1) {
        while ((c = getopt(argc, argv, “D:I:U:n:l:srh”)) != EOF) {
            switch (c) {
            case ‘D’:
                strncpy(device_name, optarg, MAX_DEV_NAME_LEN);
                break;
            case ‘I’:
                ifield = strtoul(optarg, NULL, 0);
                break;
            case ‘U’:
                ulp = strtoul(optarg, NULL, 0);
                break;
            case ‘n’:
                num_pkts = strtoul(optarg, NULL, 0);
                break;
            case ‘l’:
                pkt_len = strtoul(optarg, NULL, 0);
                break;
            case ‘s’:
                send = 1;
                break;
            case ‘r’:
                send = 0;
                break;
            case ‘h’:
                printf(usage);
                exit(0);
            default:
                die(usage);
            }
        }
    }

    if (pkt_len > MAX_WRITE)
        die(“ERROR: pkt_len (%d) must be <= %d\n”, pkt_len, MAX_WRITE);


    buf = (char*)memalign(8, pkt_len);

    if (!buf) {
        perror(“memalign() “);
        exit(1);
    }

    printf(“Opening HIPPI-FP dev = %s\n”,
           device_name);

    if (send)
        fd = open(device_name, O_WRONLY);
    else
        fd = open(device_name, O_RDONLY);

    if (fd == -1) {
        perror(“open() “);
        exit(1);
    }

    printf(“Binding to ULP = 0x%x\n”, ulp);
    if (ioctl(fd, HIPIOC_BIND_ULP, ulp) < 0)
        die(“ERROR: HIPIOC_BIND_ULP failed to bind to ulp\n”);

    if (ioctl(fd, HIPIOCW_I, ifield) < 0)
        die(“ERROR: HIPIOCW_I failed to set ifield\n”);

    printf(“%s mode:\n”, (send) ? “Send” : “Receive”);

    for (i = 1; (num_pkts ? (i < num_pkts + 1) : 1); i++) {
        int retval;

        if (send) {
            if ((retval = write(fd, buf, pkt_len)) != pkt_len) {
                fprintf(stderr, “pkt %d: trouble writing\n”, i);
                phiperr(retval);
            }
        }
        else {
            if ((retval = read(fd, buf, 8)) != 8) {
                fprintf(stderr, “pkt %d: trouble reading header\n”, i);
                phiperr(retval);
            }
            if ((retval = read(fd, buf, pkt_len)) != pkt_len) {
                fprintf(stderr, “pkt %d: trouble reading data\n”, i);
                phiperr(retval);
            }
        }

        if (i%64 == 0)
            printf(“.\n%s %d packets\n”, (send) ? “Sent” : “Received”,
i);
        else
            printf(“.”);
    }
}
```

HIPPI-FP Example

The following code is an example of a ULP using the HIPPI-FP API.

```
#include <stdio.h>
#include <sys/errno.h>
#include <sys/types.h>
#include <fcntl.h>
#include <stdarg.h>
#include <sys/hippi.h>

#define DEFAULT_ULP         0x80
#define MAX_DEV_NAME_LEN    1024
#define MAX_WRITE           0x1000000 /* 16 meg */

extern int errno;

static char *usage =
“Usage: fptest [-D<device_name] [-I<ifield>] [-U<ulp>] [-n<num
packets] [-l<pkt length>] [-s] [-r]\n”
“-D: Device name. Default is /dev/hippi0.\n”
“-I: Ifield. Default is 0, should be set every time.\n”
“-U: ULP. Default is 0x80.\n”
“-n: Number of packets to send. 0 is send forever. Default is 64.\n”
“-l: Length of packet in bytes. Default is 1024.\n”
“-s: Send mode. This is the default.\n”
“-r: Receive mode.\n”;


static char *src_errs[] = {
    “no error”,
    “SRC sequence error”,
    “SRC lost DSIC”,
    “SRC timeout”,
    “SRC lost connect”,
    “SRC connect rejection”,
    “HIPPI interface shutdown”,
    “SRC parity error”
};

static char *dst_errs[] = {
    “DST Parity Error”,
    “DST LLRC Error”,
    “DST Sequence Error”,
    “DST Sync Error”,
    “DST Illegal Burst”,
    “DST SDIC loss”,
    “DST Framing Error”,
    “DST lost linkrdy in packet”,
    “DST No Packet Signal in Connection”,
    “DST Data Received with no READY’s sent”,
    “DST No Burst in Packet”,
    “DST Illegal State Transition”
};

int send = 1, fd = -1;

void
die(char * str, ...) {
    va_list ap;

    va_start (ap, str);
    vfprintf(stderr, str, ap);
    va_end (ap);

    exit(1);
}

void
phiperr(int retval) {

    if (retval < 0 && errno == EIO) {
        retval = ioctl(fd, HIPIOCW_ERR);
        if (retval < 0)
            perror("HIPIOCW_ERR ioctl failed\n");
        else
            fprintf(stderr,
                    "HIPIOCW_ERR: %d: %s\n",retval,
                    (send) ? src_errs[retval] : dst_errs[retval]);
    }
    else
        perror("READ/WRITE ERROR");

    exit(1);
}

main(int argc, char **argv)
{
    extern char *optarg;
    extern int optind;
    char *buf;
    char device_name[MAX_DEV_NAME_LEN] = "/dev/hippi0";
    uint32_t num_pkts = 64,
             pkt_len = 0x400,
             ifield = 0,
             ulp = DEFAULT_ULP;
    int i, c;


    if(argc > 1) {
        while ((c = getopt(argc, argv, "D:I:U:n:l:srh")) != EOF) {
            switch (c) {
            case 'D':
                strncpy(device_name, optarg, MAX_DEV_NAME_LEN);
                break;
            case 'I':
                ifield = strtoul(optarg, NULL, 0);
                break;
            case ‘U’:
                ulp = strtoul(optarg, NULL, 0);
                break;
            case ‘n’:
                num_pkts = strtoul(optarg, NULL, 0);
                break;
            case ‘l’:
                pkt_len = strtoul(optarg, NULL, 0);
                break;
            case ‘s’:
                send = 1;
                break;
            case ‘r’:
                send = 0;
                break;
            case ‘h’:
                printf(usage);
                exit(0);
            default:
                die(usage);
            }
        }
    }

    if (pkt_len > MAX_WRITE)
        die(“ERROR: pkt_len (%d) must be <= %d\n”, pkt_len, MAX_WRITE);


    buf = (char*)memalign(8, pkt_len);

    if (!buf) {
        perror(“memalign() “);
        exit(1);
    }

    printf(“Opening HIPPI-FP dev = %s\n”,
           device_name);

    if (send)
        fd = open(device_name, O_WRONLY);
    else
        fd = open(device_name, O_RDONLY);

    if (fd == -1) {
        perror(“open() “);
```

```
            exit(1);
        }

        printf(“Binding to ULP = 0x%x\n”, ulp);
        if (ioctl(fd, HIPIOC_BIND_ULP, ulp) < 0)
            die(“ERROR: HIPIOC_BIND_ULP failed to bind to ulp\n”);

        if (ioctl(fd, HIPIOCW_I, ifield) < 0)
            die(“ERROR: HIPIOCW_I failed to set ifield\n”);

        printf(“%s mode:\n”, (send) ? “Send” : “Receive”);

        for (i = 1; (num_pkts ? (i < num_pkts + 1) : 1); i++) {
            int retval;

            if (send) {
                if ((retval = write(fd, buf, pkt_len)) != pkt_len) {
                    fprintf(stderr, “pkt %d: trouble writing\n”, i);
                    phiperr(retval);
                }
            }
            else {
                if ((retval = read(fd, buf, 8)) != 8) {
                    fprintf(stderr, “pkt %d: trouble reading header\n”, i);
                    phiperr(retval);
                }
                if ((retval = read(fd, buf, pkt_len)) != pkt_len) {
                    fprintf(stderr, “pkt %d: trouble reading data\n”, i);
                    phiperr(retval);
                }
            }

            if (i%64 == 0)
                printf(“.\n%s %d packets\n”, (send) ? “Sent” : “Received”,
    i);
            else
                printf(“.”);
        }
    }
```

Index of API Calls

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-2227-005.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-932-0801
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389