Sun HIPPI/P 1.0 Character Device Interface User's Guide and Reference Manual

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900
U.S.A

Part No.: 805-7707-10
March 1999, Revision A

Send comments about this document to:
docfeedback@sun.com

1999 Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, California 94303-4900 U.S.A.

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any. Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Parts of the product may be derived from Berkeley BSD systems, licensed from the University of California. UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

Sun, Sun Microsystems, the Sun logo, SunSoft, SunDocs, SunExpress, and Solaris are trademarks, registered trademarks, or service marks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK and Sun™ Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

RESTRICTED RIGHTS: Use, duplication, or disclosure by the U.S. Government is subject to restrictions of FAR 52.227-14(g)(2)(6/87) and FAR 52.227-19(6/87), or DFAR 252.227-7015(b)(6/95) and DFAR 227.7202-3(a).

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright 1999 Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, Californie 94303 Etats-Unis. Tous droits réservés.

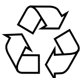
Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a. Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Des parties de ce produit pourront être dérivées des systèmes Berkeley BSD licenciés par l'Université de Californie. UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company, Ltd.

Sun, Sun Microsystems, le logo Sun, SunSoft, SunDocs, SunExpress, et Solaris sont des marques de fabrique ou des marques déposées, ou marques de service, de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays. Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

L'interface d'utilisation graphique OPEN LOOK et Sun™ a été développée par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une licence non exclusive de Xerox sur l'interface d'utilisation graphique Xerox, cette licence couvrant également les licenciés de Sun qui mettent en place l'interface d'utilisation graphique OPEN LOOK et qui en outre se conforment aux licences écrites de Sun.

CETTE PUBLICATION EST FOURNIE "EN L'ETAT" ET AUCUNE GARANTIE, EXPRESSE OU IMPLICITE, N'EST ACCORDEE, Y COMPRIS DES GARANTIES CONCERNANT LA VALEUR MARCHANDE, L'APTITUDE DE LA PUBLICATION A REPONDRE A UNE UTILISATION PARTICULIERE, OU LE FAIT QU'ELLE NE SOIT PAS CONTREFAISANTE DE PRODUIT DE TIERS. CE DENI DE GARANTIE NE S'APPLIQUERAIT PAS, DANS LA MESURE OU IL SERAIT TENU JURIDIQUEMENT NUL ET NON AVENU.



Please
Recycle

Contents

1. Overview 17

Understanding HIPPI 17

HIPPI Network Hardware Overview 18

HIPPI Connection Processing 18

HIPPI Packets 19

HIPPI I-Field 19

HIPPI-FP Operation 19

HIPPI-PH Operation 20

General Operation 20

Unexpected Packets 22

Undelivered Packets 22

NIC State 22

Special Files 23

Error Management 23

Byte Order 24

Data Buffers 24

NIC Limits 24

Data Movement Timeouts 25

Upper Layer Protocols 25

2. Management 27

Device Management 27

Opening Devices 27

Binding a Read ULP 28

HIPPI-FP and HIPPI-PH Modes 28

Data and Header Processing 29

Closing Devices 29

Obtaining Device Statistics 29

Multiple Packet Connections 30

Connection Management 30

Establishing a Connection 31

Specifying Destination Devices 31

Specifying Destination ULP 32

Many-Packet Connections 32

3. Processing 33

Received Packets 33

HIPPI-FP Separate Headers and Data 33

HIPPI-FP Combined Headers and Data 34

Unknown Packet Sizes 34

HIPPI-PH Packet Read 35

Packet Truncate 35

Packet Read Errors 35

Process Interrupt 36

Receive Queues 36

Sent Packets 36

I-Field Processing 37

FP Header Management 37

HIPPI-PH Mode 38
Unknown Packet Sizes 39
Short Bursts 39
Transmit Queue 39
`write()` and `writev()` Calls 40
Process Interrupt 41
Transmit Errors 41
I/O Multiplexing 42
Read Devices 42
Write Devices 42
Exception Devices 42

4. Portability 43
Application Portability 43
Maximum `read()` and `write()` Length 43
Buffer Alignment 43
Endian 44
Maximum Packet Length 44

5. CDI Reference 45
Header File 45
Interface Functions 45
`HIP_APP_OPEN` Call 46
Usage 46
Arguments 46
Failures and Errors 46
`close()` Call 46
Usage 47
Arguments 47

Failures and Errors 47

ioctl() Call 47

open() Call 47

Usage 48

Arguments 48

Failures and Errors 48

read() and readv() Calls 48

Usage 49

Arguments 49

Failures and Errors 50

select() Call 50

Usage 50

Arguments 51

Failures and Errors 51

write() and writev() Calls 51

Usage 52

Arguments 52

Failures and Errors 53

Ioctls 53

HIPIOC_BIND_ULP Call 53

Usage 54

Arguments 54

Failures and Errors 54

HIPIOC_GET_DEV Call 55

Usage 55

Arguments 55

Failures and Errors 55

HIPIOC_GET_DEVICE_STATE Call 55

Usage 55
Arguments 56
Failures and Errors 56
HIPIOC_GET_NICS Call 56
Usage 56
Arguments 56
Failures and Errors 57
HIPIOC_UNBIND_ULP Call 57
Usage 57
Argument 57
Failures and Errors 57
HIPIOCR_EIO Call 58
Usage 58
Arguments 58
Failures and Errors 58
HIPIOCR_ERRS Call 59
Usage 59
Arguments 59
Failures and Errors 59
HIPIOCR_GET_D1 Call 60
Usage 60
Arguments 60
Failures and Errors 60
HIPIOCR_GET_FP Call 61
Usage 61
Arguments 61
Failures and Errors 61
HIPIOCR_PKT_OFFSET Call 62

Usage 62

Arguments 62

Failures and Errors 62

HIPIOCR_SEP_HDR Call 63

Usage 63

Arguments 63

Failures and Errors 63

HIPIOCR_TRUNCATE_PKT Call 64

Usage 64

Arguments 64

Failures and Errors 64

HIPIOCW_CONNECT Call 64

Usage 65

Arguments 65

Failures and Errors 65

HIPIOCW_D1_AREA Call 66

Usage 66

Arguments 66

Failures and Errors 66

HIPIOCW_D1_AREA_PTR Call 67

Usage 67

Arguments 67

Failures and Errors 67

HIPIOCW_D1_SIZE Call 68

Usage 68

Arguments 68

Failures and Errors 68

HIPIOCW_DISCONN Call 69

Usage 69
Arguments 69
Failures and Errors 69
HIPIOCW_END_PKT Call 69
Usage 70
Arguments 70
Failures and Errors 70
HIPIOCW_ERR Call 70
Usage 70
Arguments 70
Failures and Errors 71
HIPIOCW_I Call 71
Usage 71
Arguments 72
Failures and Errors 72
HIPIOCW_SEP_HDR Call 72
Usage 73
Arguments 73
Failures and Errors 73
HIPIOCW_SET_ULP Call 73
Usage 74
Arguments 74
Failures and Errors 74
HIPIOCW_SHBURST Call 74
Usage 75
Arguments 75
Failures and Errors 75
HIPIOCW_START_PKT Call 75

Usage 76

Arguments 76

Failures and Errors 76

6. Troubleshooting 77

NIC Installation and Operation 77

▼ To Test the Installation and Operation of the NIC 77

Optical Modules and Cables 79

Optical Connections 79

▼ To Turn On RunCode and Check the Status 79

Optical Loopback Test 80

▼ To Set Up the Loop-Back Test 80

Optical Loop-Back Through a Switch 81

▼ To Set Up Optical Loop-Back Through a Switch 81

Optical Testing Between NICs 83

▼ To Test the Optical Connection Between NICs 83

Long Packets Between NICs 86

▼ To Set Up the NIC for Long Packets 86

▼ To Change the EEPROM for Long Packets 87

Preface

The HIPPI Character Device Interface User's Guide and Reference Manual describes the operation and use of the High Performance Parallel Interface (HIPPI) Network Interface Card (NIC).

Before You Read This Book

This manual is intended for Sun Enterprise system administrators and application developers, who should have a working knowledge of UNIX systems, particularly those based on the Solaris operating environment. If you do not have such knowledge, you should first read the Solaris User and System Administration AnswerBook documentation provided with your Sun Enterprise server and consider UNIX system administration training.

How This Book Is Organized

This manual contains the following chapters:

Chapter 1 "Overview" describes the general operation of the HIPPI network interface card.

Chapter 2 "Management" describes the process that an application uses to access the HIPPI network interface card.

Chapter 3 "Processing" explains how packets are received and sent over the HIPPI network.

Chapter 4 “Portability” explains the compatibility and portability issues of the HIPPI network.

Chapter 5 “CDI Reference” contains the Character Device Interface (CDI) reference pages that include the header file, the interface functions, the ioctls, and special files.

Chapter 6 “Troubleshooting” outlines techniques that are helpful in isolating problems.

Appendix A “Special Files” describes how the CDI supports multiple NICs in the domain and multiple devices on each NIC.

Appendix B “HIPPI-SC Excerpts” contains information that is excerpted from the HIPPI-SC specification.

Appendix C “HIPPI-FP Excerpts” contains information that is excerpted from the HIPPI-FP specification.

Appendix D “HIPPI-PH Excerpts” contains information that is excerpted from the HIPPI-PH specification.

Chapter “Glossary” contains a list of words and phrases and their definitions.

Chapter 1 describes entrance requirements for 14 trade schools. The chapter includes an application form.

Appendix A should be used only by experienced technical writers in panic situations.

Glossary is a list of words and phrases found in this book and their definitions.

Using UNIX Commands

This document may not contain information on basic UNIX commands and procedures such as shutting down the system, booting the system, and configuring devices.

See one or more of the following for this information:

- AnswerBook online documentation for the Solaris operating environment
- Other software documentation that you received with your system
- Other software documentation that you received with your system

Typographic Conventions

TABLE P-1 Typographic Conventions

Typeface or Symbol	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`
`AaBbCc123`	What you type, when contrasted with on-screen computer output	`%` **`su`** `Password:`
AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this.
	Command-line variable; replace with a real name or value	To delete a file, type `rm` *filename*.

Shell Prompts

TABLE P-2 Shell Prompts

Shell	Prompt
C shell	*machine_name*%
C shell superuser	*machine_name*#
Bourne shell and Korn shell	$
Bourne shell and Korn shell superuser	#

Related Documentation

TABLE P-3 Related Documentation

Application	Title	Part Number
Reference	*HIPPI/P 1.0 CDI Reference Manual*	805-7708-10
	Sun HIPPI/P 1.0 Installation and User's Guide	805-7133-10
Background information	*ANSI X3.183-1991, High-Performance Parallel Interface, Mechanical, Electrical, and Signaling Protocol Specification (HIPPI-PH)*	N/A
	Serial-HIPPI Specification, Revision 1.0, Serial HIPPI Implementors Group, May 17, 1991	N/A
	ANSI X3.218-199x, High-Performance Parallel Interface – Framing Protocol (HIPPI-FP)	N/A
	ANSI X3.218-199x, High-Performance Parllel Interface – Encapsulation of ISO 8802-2 (IEEE Std. 802.2) Logical Link Control Protocol Data Units (HIPPI-LE)	N/A
	ANSI X3.222-199x, High-Performance Parallel Interface – Physical Switch Control (HIPPI-SC)	N/A
	IEEE Standard 802.1A	N/A

Sun Documentation on the Web

The `docs.sun.com` web site enables you to access Sun technical documentation on the Web. You can browse the `docs.sun.com` archive or search for a specific book title or subject at:

`http://docs.sun.com`

Sun Welcomes Your Comments

We are interested in improving our documentation and welcome your comments and suggestions. You can email your comments to us at:

`smcc-docs@sun.com`

Please include the part number of your document in the subject line of your email.

1

Overview

The Character Device Interface (CDI) enables application programs to access the HIPPI (High-Performance Parallel Interface) network using a UNIX® character device interface. The CDI passes HIPPI packets to and from the HIPPI network and controls the operation of the HIPPI NIC (Network Interface Card) that is configured into the host system. The CDI supports multiple NICs in a host as well as hosts with multiple processors (SMP).

The usual `open()`, `close()`, `read()`, `write()`, `ioctl()`, and `select()` interfaces can control the operation of the NIC and pass packets over the HIPPI network. When an operation fails, -1 is returned, and errno is set to a value in `/usr/include/sys/errno.h`. Throughout this guide, error numbers that are listed are in addition to generic errors that may occur when calling the interface. The host system man page details the generic errors.

The application opens one or more HIPPI Transfer Device special files and uses `read()`, `write()`, and `ioctl()` to manage the transfers. Each transfer device can be exclusively opened by one process at a time for read only, write only, or both read and write.

Understanding HIPPI

This section contains background information that is helpful in understanding the underlying HIPPI technology.

HIPPI Network Hardware Overview

HIPPI is a switched point-to-point connection-based network technology. A source NIC (transmitter) connects to a destination NIC (receiver). When the connection is established, one or more packets are passed over the connection. When the data transfer has been completed, the connection is terminated. The sender can wait for a connection (CampON) or give up if the destination is busy.

The source and destination are independent and may operate concurrently. The NIC is capable of transferring 1,600,000,000 bits per second (burst) over the HIPPI media because HIPPI transfers 800,000,000 bits per second independently on the source and on the destination. HIPPI provides a hardware flow control mechanism when the maximum rates cannot be sustained due to host constraints such as I/O bus capability, system memory bandwidth, and general system load.

HIPPI packets are routed through HIPPI switches by placing a routing control field, called the CCI field, or I-Field, ahead of the packet. The routing control field contains the destination address and optionally the source address (for use in returning packets to the sender). There are two forms of addresses *logical* and *source*. Logical addressing identifies a destination with a 12-bit address. The switch is responsible for establishing the physical route through the switch. Source routing specifies the physical path from the source to the destination (see Appendix B “HIPPI-SC Excerpts” for more information).

A HIPPI network consists of a HIPPI fabric and a set of HIPPI end-points. The fabric consists of a collection of interconnected HIPPI switches. It can be viewed as a *cloud* that handles passing packets between end-points. HIPPI end-points are HIPPI NICs in the host and on other peripherals. Usually, the administrator assigns a unique logical address to each end-point in the HIPPI fabric.

HIPPI media may be either copper cables or optical cables. Sun provides NICs that are based on the RoadRunner ASIC and have optical media connections. The Sun NICs include both source and destination interfaces.

HIPPI hardware does not support broadcast or multicast. However, logical addresses are reserved for pseudo broadcast mechanisms. The pseudo broadcast mechanisms are used by the network driver.

HIPPI Connection Processing

HIPPI passes data packets over a point-to-point connection. The connection must be established before packets can be transferred. After a connection is established, one or more packets can be transferred from the source to the destination.

The connection is usually opened for a single packet and closed after the packet is transferred. However, this interface permits the connection to be opened for multiple packets or opened indefinitely. Connections that are opened indefinitely, or for multiple packets, must be explicitly closed by the application.

HIPPI Packets

HIPPI packets are sent as a set of 1024-byte bursts. The packet ends when the data has been sent. A packet may have one short burst (a burst that is less than 1024 bytes). The short burst is usually the last burst in the packet. However, it may be the first burst in the packet. When it is the first burst, the application must ensure that the last burst is full size. Each burst is followed by a hardware-managed LLRC field.

Even though there is one I-Field for a connection, the CDI and RoadRunner ASCI require each packet to be preceded by an I-Field.

When HIPPI-FP headers are used, the presence of a short, first burst may be noted by a flag in the header (see Appendix C “HIPPI-FP Excerpts” for more information).

HIPPI I-Field

An I-Field introduces each connection (see Appendix B “HIPPI-SC Excerpts” for more information). The I-Field is used by the switch to route the packet to its destination. The CDI requires an I-Field before every packet, even when more than one packet is passed during a connection (even though the extra I-Fields do not appear on the HIPPI media). The I-Field is stripped during reception and is not available to the application.

HIPPI-FP Operation

The HIPPI-FP header contains information about the packet (see Appendix C “HIPPI-FP Excerpts” for more information). The header is always in Big Endian byte order. The header is always at the start of the first burst.

Following the FP header is the optional D1 header. The D1 header is managed by the application. It can be up to 1016-bytes long and is in the first burst. When a short, first burst is indicated, the FP header and D1 area are in the first burst.

Following the D1 area is the optional D2 area (user data). This area is managed by the application. When a short, first burst is indicated, the D2 area starts at the beginning of the second burst and is a whole number of bursts long.

HIPPI-PH Operation

HIPPI-PH does not use headers. The packet starts with a start-packet indicator and ends with an end-packet indicator. The packet consists of one or more bursts. A full-size burst is 1024 bytes. Only one short burst can be in a packet. The short burst may be either the first or last burst of the packet. The packet length must be a multiple of 4 bytes in length.

General Operation

When a connection is established, all of the HIPPI resources (that is, NICs, paths, and through switches) that are needed by the connection are exclusively dedicated to the connection until the disconnect. The I-Field can request that a connection wait until a busy resource is released (CampON) or immediately fail.

A NIC that is operational will always accept a connection request and will accept packets even if no application, or an unexpected application, is ready to receive the data. The sender has no way of determining whether the data was received by the destination application or just discarded.

The application treats HIPPI as a datagram service, similar in behavior to UDP/IP. HIPPI provides a *best effort* transport. Packets may or may not be delivered to the destination. If the destination NIC is running, the packets will be accepted. If an application is not ready to receive the packets, they are discarded. If an unexpected application is ready to receive the packets, the packets are delivered to the waiting application. Because any source can send to any destination, the receiving application must verify the source of the data before the data is processed by the application. The application must be able to discard unwanted data.

HIPPI is defined as a low-level transfer protocol, HIPPI-PH, which is frequently used as a dedicated link between systems. HIPPI-FP is layered on HIPPI-PH, providing a packet transfer protocol that passes packets among users of the HIPPI network. Applications should consider using HIPPI-PH in a direct-connection configuration where the resources can be dedicated to the connection. HIPPI-FP should be used in a general network configuration. The administrator uses the `hippi(1M)` utility to set the NIC to operate in HIPPI-FP or HIPPI-PH mode.

The CDI manages the connection process and the details of passing HIPPI-FP and HIPPI-PH packets. It also provides error management and statistics gathering services. It uses blocking I/O (attempts to use non-blocking I/O are ignored). When a `read()` or `write()` is executed, the process sleeps until the request is completely processed. The `select()` system call may be used to avoid long pauses in processing.

The CDI independently transmits packets and receives packets. During the transmission, the CDI uses an `ioctl()` to manage headers and control the transfer and a `write()` or `writev()` to send the data. During the reception, the CDI uses an `ioctl()` to control the transfer and `read()` or `readv()` to receive the data. The NIC uses DMA transfers to move data directly between the application data buffers and its local memory. The data does not pass through operating system buffers.

All transmit requests are funneled to a single transmit queue. Transmit requests are added to the queue in the order in which they arrive. Each sender has exclusive access to the transmit queue until all of the packets for the connection have been queued.

Receive processing is different for HIPPI-FP and HIPPI-PH. In HIPPI-FP mode the application binds to an Upper Layer Protocol (ULP). The NIC analyzes the HIPPI-FP header and routes packets based on the ULP in the HIPPI-FP header directly to the application program. When there is no application initialized to receive packets to the ULP, the packets are discarded (the sender does not receive an error indication). Because HIPPI-PH does not have defined headers, all received packets are passed to the single HIPPI-PH ULP.

The CDI supports server applications by allowing multiple applications to be concurrently bound to a single ULP. As packets arrive for the ULP, they are distributed among the applications that are bound to the ULP, based on the order of each `read()` that is queued and waiting for data. This has the effect of randomly distributing packets among the waiting applications.

When an error occurs in transferring a packet, `read()`, `readv()`, `write()`, and `writev()` return –1 and set `errno` to `EIO`. An `ioctl()` provides detailed information about the failure.

The CDI supports multiple NICs in the host system. The application directs the CDI to a specific NIC by opening the proper special file for the NIC (for example, `NIC0`, `NIC1`, `NIC2`, `NIC`*n*).

The `hippi(1M)` utility controls the operating mode of the NIC (for example, FP verses PH). It can be used to start and stop the NIC. It can also be used to provide operating statistics. The CDI provides an `ioctl()` to get the current mode settings.

Because the HIPPI resources are assigned for the duration of the transfer, a process that unexpectedly stops sending or receiving data will indefinitely tie up the HIPPI channel. To prevent this, timeouts are used on both the receive and transmit channels. After a packet starts to transfer it must continue to transfer. If data is not transferred during the timeout period, the packet is truncated, and the connection is closed. You can configure the timeout by using the `hippitune(1M)` command.

Unexpected Packets

Usually, when an application starts, it opens a HIPPI transfer device and binds to a ULP that it uses for receiving packets. The application then sends and receives packets. The method of determining the destination NIC address and ULP that the application should use is not defined in the CDI.

The result is any HIPPI source can pass packets to any HIPPI destination. The packets are passed to a NIC and ULP. The desired application is expected to be bound to the destination ULP before the first packet arrives. The source does not know for sure that the destination is the expected application or that the application is actually set up as expected.

With HIPPI, you cannot establish a session as you can if you are using TCP/IP). Usually, connections are established and broken for each packet instead of being established for the duration of time the application need to execute the transmission. Thus, an application may receive packets that it is not expecting.

The application must validate the source of every packet it receives before the packet is processed. The application must ensure that unexpected packets are handled properly (usually discarded). When the source application sends packets to the wrong destination application, the source is not notified of the error.

Undelivered Packets

Usually, the destination NIC accepts a packet whether or not an application is ready to receive it. If an application is not running, and the ULP is not bound, the packet is discarded by the NIC. The source application is not notified of this error.

NIC State

The NIC is a programmable device that executes a firmware program called RunCode. The `hippi(1M)` utility is used to manage the NIC including loading RunCode into the NIC's local memory (SRAM), starting the RunCode and setting the operating state (see the `hippi(1M)` man page for more information).

The operating states include:

- `on` – The NIC is executing RunCode.
- `off` – The NIC is stopped.
- `long` – The CDI can send packets that are longer than 64 kilobytes.
- `short` – The CDI is limited to sending 64-kilobyte packets.
- `fp` – The CDI is in the HIPPI-FP mode.
- `ph` – The CDI is in the HIPPI-PH mode.

In `fp` mode the network driver can operate concurrently with the CDI. When you configure the network driver, you should limit the maximum packet size to the network MTU-size (64 bytes) by setting it to `short`. Send requests are queued to be sent one at a time. Permitting CDI applications to send long packets ties up the NIC for extended periods of time and results in lengthy delays in sending network packets. The NIC receives a single packet at a time; therefore, receiving long packets delays the arrival of network packets. When the CDI is set to `short`, the CDI prevents long packets from being sent, although it still receives long packets. The delays can be long enough for network protocols to time out. Forcing the CDI to limit packets to network MTU ensures that the network applications get a reasonable share of HIPPI bandwidth.

Special Files

The CDI accesses each NIC through a set of special files that are created when the product is installed. Each NIC has one special file that is used for device control, called the control device, and a series of special files, called transfer devices. Applications use the transfer devices to access the NIC for data transfer (`read()` and `write()`).

Error Management

Errors are reported as soon as practical after they are detected. Errors in calling the access functions are reported immediately. The function returns `-1` and the error code is placed in `errno`. Values for `errno` are found in the host system `/usr/include/sys/errno.h` header file.

When an error occurs in transferring a packet the packet is truncated, and the connection is terminated. When using multiple-packets-per-connection, the next packet sent by the application establishes a new connection.

When calling an interface function, such as `open()`, `close()`, `ioctl()`, `read()`, and `write()`, on a transfer device while the NIC is off, `ENODEV` is returned.

Because `read()` and `write()` are not supported on the control device, `EINVAL` is returned. The following list includes other conditions that return specific `errno` values:

- Parameter errors return `EINVAL`.
- Invalid buffer pointers return `EFAULT`.
- Insufficient permissions return `EPERM`.

The response to `read()` I/O transfer errors may be configured using the `HIPIOCR_EIO` `ioctl()`. The application may discard the data and return `EIO` (default action), or it may ignore the error and return the length of the available data. In the later case the application must call the `HIPIOCR_ERRS` `ioctl()` after every `read()` to determine the true return status.

Byte Order

The HIPPI network is Big Endian. The I-Field and HIPPI-FP headers must be in Big Endian byte order. Little Endian systems will likely have to byte swap these headers to host order for local processing.

The D1 area (optional) and D2 area (optional) are in whatever endian the application desires. When an application operates with an application on a different endian system, the applications must properly manage the endian of the data.

Data Buffers

The CDI restricts buffers to be aligned on 8-byte boundaries. When multiple `write()` or `read()` functions are used to process a packet, the buffer for each `write()` or `read()`, except the last, must be a multiple of 8 bytes. When `writev()` or `readv()` is used each segment must be 8-byte aligned. All segments except the last segment for the packet must be a multiple of 8 bytes in length. The CDI does not restrict packet length. Host systems may further restrict length or alignment.

NIC Limits

Up to 31 unique, receive ULPs may be bound at a time. The product is installed with one control device and 31 transfer devices per NIC.

The NIC is shared among all concurrent users. The latency and available bandwidth vary as the number of active users change as well as usage of the HIPPI network change. It is possible for an application to have to wait a considerable time to send or receive a packet.

Data Movement Timeouts

The NIC uses separate receive and transmit data-movement timeouts to monitor data movement. After a connection is opened and packet data starts to flow, the timeout is active. If data is not passed during the timeout period, the packet is truncated; the connection is dropped, and an error is returned. The timeout values are set by the administrator using the `hippitune(1M)` utility.

Upper Layer Protocols

The HIPPI-FP header defines an 8-bit Upper Layer Protocol (ULP) field that is used by the NIC to directly route a received packet to an application. HIPPI-FP explicitly specifies the use of some of the ULPs and reserves a range of ULPs for later assignment. ULPs in the range of 128 to 255 (decimal) are available for general application use. ULPs in the range of 0 to 127 (decimal) are reserved by HIPPI-FP and should not be used by the application. The application must have superuser privilege to bind to ULPs in the reserved range.

Within the reserved range, certain ULPs have been reserved for specific uses. The specific ULPs that are reserved by HIPPI-FP must not be used by the application. The CDI does not permit access to the following ULPs:

- 2 – Memory interface
- 3 – Memory interface initialization
- 4 – ISO 8802.2 link encapsulation
- 6 – IPI-3 slave
- 7 – IPI-3 master
- 8 – IPI-3 peer
- 10 – HIPPI-FC mapping to fibre channel

2

Management

This chapter contains information about the process that an application uses to access the HIPPI NIC, including how it opens and closes devices and connections. This chapter also contains information about multiple-packet connections and many-packet connections.

Device Management

This section contains information about how devices are opened, what operating modes are available, how the data and headers are processed, how the devices are closed, and what statistics can be obtained.

Opening Devices

The application uses the `HIP_APP_OPEN` macro to open a CDI, transfer-device, special file. Each CDI transfer device is opened for exclusive access by the application. Because the transfer devices are opened for exclusive access, they must be closed before a process calls `fork()`. (Failure to close the device before the `fork()` command may result in unpredictable behavior).

`HIP_APP_OPEN` opens a transfer device on the given NIC for, read (`O_RDONLY`), write (`O_WRONLY`), or both read and write (`O_RDWR`). An application can have more than one transfer device open at a time and call `HIPIOC_GET_NICS` `ioctl()` to determine the number of configured NICs.

Binding a Read ULP

While in HIPPI-FP mode, the NIC routes packets to applications based on the ULP field in the HIPPI-FP header. When an application opens a transfer device for read, it must use the `HIPIOC_BIND_ULP` `ioctl()` to bind the transfer device to a ULP. Until a ULP is bound no packets can be received. The bind may be for exclusive access or shared access. When the NIC is in HIPPI-PH mode, no ULP exists. However, the application must still bind the transfer device to establish shared or exclusive access.

Multiple transfer devices may be bound to the same ULP. When this occurs, a `read()` from the applications are queued and completed, in turn, as packets arrive. This results in the received packets appearing to be randomly distributed among the applications. This feature permits server applications to have multiple processes monitoring the same ULP.

The open transfer device may be bound to a single ULP (or no ULP) at any point in time. The `HIPIOC_UNBIND_ULP` `ioctl()` removes the ULP binding. The application may then use `HIPIOC_BIND_ULP` to bind the transfer device to another ULP. A transfer device that is open for write-only does not bind to a ULP.

The sending and receiving applications must negotiate a suitable ULP. The CDI does not provide a means of determining suitable ULPs. The CDI also does not provide a means for a destination to validate the source of a packet.

HIPPI-FP and HIPPI-PH Modes

The NIC operating mode is set by the `hippi(1M)` utility (there is no `ioctl()` in the CDI to set the operating mode). However, the application can determine the current mode by using the `HIPIOC_GET_DEVICE_STATE` `ioctl()`.

In HIPPI-FP mode, the NIC multiplexes received packets based on ULP. The application binds a ULP to a transfer device then it reads the transfer device file by using the `read()` function. When a packet arrives for an unbound ULP, the packet is received and discarded.

Every packet that is sent must have an I–Field and a valid HIPPI-FP header. Even though the I-Field is only needed at the first packet of a connection, the CDI and hardware consistently require an I-Field before every packet. The HIPPI-FP mode must be set when the network driver is configured.

HIPPI-PH mode does not use headers. All packets arrive at the same place and are passed to the next request in the receive queue. Every connection that is established must have an I-Field.

Data and Header Processing

While in HIPPI-FP mode, the CDI supports two methods of handling headers and data. One mode, called separate headers, uses an `ioctl()` to process headers and a `read()` or `write()` function to process data. The other mode, called combined headers, treats the entire packet as data.

Separate-header mode is the default for sending packets. The FP header and, optionally, the D1 area are cached in the CDI then they are affixed to each packet before it is sent.

Combined-headers mode is the default for receive. The entire packet (starting with the FP header) is read by the `read()` function.

Receive processing and send processing differ in the following ways:

- Send processing includes an I-Field before each packet; receive processing does not include the I-Field.
- Send processing permits passing a short, first burst; receive processing does not notice burst length.

HIPPI-PH does not support headers, so all header related `ioctl()` functions are invalid in this mode.

Closing Devices

The device may be explicitly closed by the application or implicitly closed during process termination. When the device is closed, an outstanding `read()` or `write()` is completed with `EIO`. When the last transfer device that is bound to a ULP is closed, the ULP is unbound. Subsequent packets to the ULP are discarded.

On close, a packet that is partially transmitted is truncated. The connection, including a multiple-packet connection, is closed as well.

All CDI transfer devices must be closed before calling `fork()`; otherwise, the behavior of the CDI is unpredictable.

Obtaining Device Statistics

Operating statistics for the NIC are maintained by the CDI. The application may get the current statistics by using the `HIPIOC_GET_STATS ioctl()`. Also, you can invoke `hippi status` or `hippistat -x` to display NIC statistics.

Multiple Packet Connections

An application may need to send a group of packets to a destination without packets from other applications being interspersed. The CDI provides a multiple-packet connection that is managed by using the `ioctl()` function. Because a multiple-packet connection ties up the two NICs and the paths through the switches, you should dedicate resources to multiple-packet connections or limit their use.

The application can send multiple packets over the same connection in the following way. The connection is announced by a `HIPIOCW_CONNECT` `ioctl()`. The next `write()` starts a new packet and opens the connection. As many packets as desired may be sent over the connection. After the connection is opened, it remains open until one of the following conditions occur:

- It is closed by a `HIPIOCW_DISCONN` `ioctl()`.
- An error occurs (including `Tx Idle`).
- The transfer file is closed.
- The NIC stops.

After the connection is opened, the `Tx Idle` watchdog is in effect. It remains in effect until the connection is closed. For this reason, the `Tx Idle` timeout must be configured to be longer than the time between packets on the connection.

`HIPIOCW_CONNECT` can be used between packets. It fails if the application is between a `write()` of a multiple `write()` packet.

Connection Management

A connection is established between a source and a destination before packets can be sent. Additionally, when FP headers are used, the source must also know the ULP at the destination.

HIPPI provides a datagram service. It behaves in a similar fashion to UDP/IP. In this context, the HIPPI connection lasts long enough to pass a packet between the source NIC and the destination NIC. It is not a connection that is similar to TCP/IP.

The connection attempt is performed on the open NIC. In a multiple NIC configuration, the application must open the correct NIC. The CDI does not provide a mechanism for managing the various NICs in a configuration.

Establishing a Connection

Before a packet is sent, the source must establish a connection. This is done by transmitting an I-Field. The I-Field is routed through the HIPPI network to a destination device using information in the I-Field to control processing.

The I-Field contains either a logical address or a source route. Bits in the I-Fields specify the address format. HIPPI switches use the address to route the connection through the switches.

The destination device may be in one of the following states:

- Available to accept the connection
- Already connected to another source
- Not available (disconnected, turned off, or missing)

When the destination is available, a connection is immediately established, and the source can start to pass packet data. Packets may be passed until a disconnect occurs (usually at the end of a packet).

When the destination is already connected to another source, the `CAMP_ON` bit in the I-Field is used to direct processing. When `CAMP_ON` is set, the source waits for the destination to become available. The use of `CAMP_ON` is very common. When the destination disconnects, it immediately connects to a source that is camped on. Because `CAMP_ON` ties up the source and all of the HIPPI switch ports along the path from source to destination (even though no data is being passed), a `CAMP_ON` timeout is provided so that the source can break the connection attempt after a configured period of time. The `hippitune(1M)` utility is used to set this timeout.

When the destination is busy and `CAMP_ON` is not set, the connection is immediately rejected. The `hippitune(1M)` command can be used to configure the number of times that the connection is retried before giving up and how long the source should wait between retries. When the destination is not available, the connection attempt is immediately rejected.

When two HIPPI devices are directly connected (instead of going through a HIPPI switch), the I-Field is not really needed. However, it must still appear in the expected places even though its content is not examined.

Specifying Destination Devices

When HIPPI NICs are connected using HIPPI-SC switches, the address of the destination device must be known to the application before HIPPI packets can be sent. There is no standard method of finding the address of a destination device using HIPPI packets. It is up to the application to determine the destination address.

When two HIPPI NICs are directly connected, the address of the destination device is not used. The address of the destination device is placed in the I-Field.

Specifying Destination ULP

When the destination NIC is in HIPPI-FP mode, the transmitted packets must have a valid HIPPI-FP header. The destination ULP must be known to the application before HIPPI packets can be sent. A standard method does not exist for finding a destination ULP by using HIPPI packets. The application must determine the destination. All packets that are sent to a destination are accepted. Error messages are not returned if the ULP is not valid or if it is in use by another application.

Many-Packet Connections

The `HIPIOCW_CONNECT` `ioctl()` is used to open a many-packet connection when the next `write()` is executed. A many-packet connection is closed by using `HIPIOCW_DISCONN` or by any error. The data movement timeouts are in effect.

3

Processing

This chapter explains how packets are received and sent over the HIPPI network.

Received Packets

The received packets are read by `read()` or `readv()`. See the host-system man pages for the general behavior of these interfaces. Alternatively, `read()` or `readv()` can read the data, the D2 area, and an `ioctl()` can read headers and track the progress of packet processing.

A single `read()` never returns more than one packet. A single packet may be read by using one or more `read()` calls. When multiple `read()` calls are used, all buffers must be 8-byte aligned, and all but the last buffer must be a multiple of 8 bytes in length. Each `read()` returns the actual number of bytes that are read by that call.

The FP header is always in Big Endian. The other headers and data are in whatever byte order the application chooses. The application must completely process each packet that arrives (both expected and unexpected packets). An `ioctl()` call can be used to track the progress of receiving packets and to discard unwanted packets.

HIPPI-FP Separate Headers and Data

When separate-header mode is set by the `HIPIOCR_SEP_HDR` `ioctl()`, the CDI caches the FP header and D1 area in the driver. The application reads the FP header by using the `HIPIOCR_GET_FP` `ioctl()`. `HIPIOCR_GET_FP` is used before the `read()` call so that it blocks the transmission until the packet arrives instead of the `read()` call blocking the transmission.

The FP header is always in Big Endian byte order. The D1-size field indicates the size of the D1 area (if any), and the D2-size field indicates the size of the D2 area (if any). Using the D1-size field and the D2-size field, the application can determine the overall length of the packet. When the D2-size field contains 0xFFFFFFFF, the length is not known. See "Unknown Packet Sizes" on page 34 for processing details.

When the D1-size field is not zero (0), HIPIOCR_GET_D1 ioctl() gets the D1 header for the latest packet. HIPIOCR_GET_FP must be called before HIPIOCR_GET_D1. A series of read() calls may be used to read the D2-data area. The headers must be processed before the D2 area.

If the application is not concerned with headers, the packets (D2 data) may be processed by just using read() calls because the headers are discarded. When headers are ignored, the read() call blocks the data until a packet arrives.

HIPPI-FP Combined Headers and Data

When combined-header mode (the default) is set by the HIPIOCR_SEP_HDR ioctl(), a series of read() calls are used to read the whole packet, starting at the FP header. The FP header is always in Big Endian byte order. The D1-size field indicates the size of the D1 area (if any), and the D2-size field indicates the size of the D2 area (if any). Using the D1-size field and the D2-size field, the application can determine the overall length of the packet. When the D2-size field contains 0xFFFFFFFF, the length is not known. See "Unknown Packet Sizes" on page 34 for processing details. The read() call blocks the calling thread until data is available to be read.

Unknown Packet Sizes

In HIPPI-FP mode, when the D2 size of a packet is not known and when the packet is sent, D2 is set to D2SIZE_UNKNOWN (-1 or 0xFFFFFFFF). The application is expected to read as much data in the packet as it can, or it should read a portion of the data and discard the rest.

When the NIC is in FP mode, the packet must be a multiple of 8 bytes in length. HIPPI-PH mode does not use headers, so all packets are of unknown length. When the NIC is in PH mode, the packet must be a multiple of 4 bytes in length. When a short, first burst is used, the remainder of the packet must be a multiple of 1024 bytes in length.

The application can use the HIPIOCR_PKT_OFFSET ioctl() to report the byte offset into the current packet. When the maximum byte count (MAX_PKT_OFFSET) is reached, the remainder of the packet can still be read; however, the byte count will

not advance. MAX_PKT_OFFSET is 2048 times 4096 bytes or 0x7FFFFFFF bytes. When the whole packet has been read, the counter is reset to zero (0), and the ioctl() returns a 0-byte offset.

HIPPI-PH Packet Read

HIPPI-PH mode does not use headers. The header related ioctl() is not valid and returns EINVAL. The application can use the HIPIOCR_PKT_OFFSET ioctl() to keep track of how much data has been read (up to MAX_PKT_OFFSET bytes) and to determine packet boundaries (see “Unknown Packet Sizes” on page 34).

Packet Truncate

When a packet is not expected by the application or is longer than the application is prepared to handle, the HIPIOCR_TRUNCATE_PKT ioctl() can be used to discard the remainder of the packet. This is useful when processing packets with multiple read() calls.

Packet Read Errors

Errors may be detected as the host system processes the read() or as the CDI attempts to receive packets. Using the default, UNIX error management, read() returns -1 and sets a return code in errno. The values of errno are found in /usr/include/sys/errno.h. When an error occurs, data in the NIC is discarded. This data would have been returned to the application.

Data errors return EIO. When processing has been interrupted by a signal, EIO is returned. When an EIO or EIO error is returned, the HIPIOCR_ERRS ioctl() can be used to get a more detailed analysis of the error.

The HIPIOCR_EIO ioctl() can be used to place the CDI in error- monitor mode. In this mode, the application will never receive an EIO from a read(), and it will never lose data that is available to the NIC. Thus, the application must use the HIPIOCR_ERRS ioctl() after each read() to check the error status. When using this mode, the application receives all of the available data, whether it is corrupted or not.

Process Interrupt

When a process interrupt is signaled (usually `^C` or `kill()`), a pending read or write returns `EIO`, and the packet is truncated (unprocessed data is discarded). The next `read()` starts a new packet.

Receive Queues

When the first transfer device is bound to a ULP, a receive queue is set up for the ULP. In shared mode many transfer devices may be bound to the same ULP. Each transfer device can perform packet reads independent of the other transfer devices, even if more than one read is needed to receive the packet.

Each read is queued in the ULP queue and processed in turn. When a packet requires more than one read, other readers must wait for the whole packet to be received before their requests can be queued.

Sent Packets

HIPPI provides a means of sending packets through switches over an established connection to a destination. Every connection is established by a switch control block called an I-Field. HIPPI packets are passed over the established connection. The content and format of the HIPPI packets depends on the HIPPI protocol in use.

HIPPI-FP packets consists of a HIPPI-FP header followed by a D1 header (optional), which is followed by the D2-data area (optional). The FP header and D1 header can be sent as a short, first burst.

HIPPI-PH packets do not have headers. The packet is preceded by a packet-start byte and is terminated by a packet-end byte.

Normally, when an application communicates with another application, the I-Field, FP header, and D1 header to not change much from packet to packet. Thus, the application can cache the header information in the CDI so that the CDI can format the packets by using the cached headers.

Packets can be sent by using one or more `write()` calls. One or more packets can be sent over a single connection. Multiple packets cannot be sent in a single `write()` call.

I-Field Processing

Every connection is established by using a HIPPI-SC I-Field. The I-Field is set by using the HIPIOCW_I or HIPIOCW_CONNECT ioctl() as described in "Connection Management" on page 30. The I-Field is cached in the CDI. After it is set, it is used on successive packets until a new value is set. The I-Field needs to be set before the first connection is established or whenever a packet is sent to a different destination. The application must ensure that the I-field is always in Big Endian order. The I-Field is always set by ioctl(); it is never part of a user buffer.

FP Header Management

In HIPPI-FP mode, the FP header is required. The header may be cached in the driver and reused from packet to packet, or it may be placed at the beginning of the data buffer. The write mode is set by using HIPIOCW_SEP_HDR ioctl(). When the FP header is in the application buffer, the length of the fields must be correct, and it must be in Big Endian order (see Appendix B "HIPPI-SC Excerpts" for the FP header format). When the FP header is cached by the CDI, the application uses a set of ioctl() calls to manage the various fields. The cached values are used until they are changed or the mode is changed to combined-header-and-data-mode. Entering separate-headers-and-data mode clears the cached FP header.

HIPIOCW_SET_ULP sets the destination ULP. Observe the rules for reserved ULPs described in "Data Movement Timeouts" on page 25. HIPIOCW_SHBURST forces the first burst to be short (see "HIPPI-PH Mode" on page 38). All remaining bursts must be a full 1024 bytes in length.

HIPIOCW_D1_SIZE sets the size of the D1 area. The D1 area is a multiple of 8 bytes in length, up to 1016 bytes. If the D1-data area is not present, the D1 length is set to zero (0). The application manages the content of the D1 space. When the D1-size field is zero (0), the packet does not have a D1 header.

HIPIOCW_D1_AREA_PTR tells the CDI the location of the D1 buffer in the application. HIPIOCW_D1_AREA_PTR is used when the application manages the D1 area in a private buffer. When this option is used, the first write() call of the packet causes the current D1 area to be copied to the kernel cache and sent along with the FP header. The application can change the content of the D1 header up to the write() call. The remainder of the packet is transferred directly from the application buffer to the NIC. When the buffer pointer is set to NULL, the D1 user-buffer is no longer used, and the D1 area is part of the application buffer.

HIPIOCW_D1_AREA copies the D1 area to a cache in the CDI driver. The application can replace the D1 data in the cache if needed. HIPIOCW_D1_SIZE must be called first to set the size. When the buffer pointer is NULL, the D1 cache is cleared, and the D1 area is part of the application buffer.

The application can do the following:

- Not use a D1 header.
- Place the D1 area before the data in the data buffer.
- Place the D1 area in a separate buffer.
- Cache the data in the kernel.

Cached data is used in multiple packets until it is changed. If `HIPIOCW_D1_AREA` or `HIPIOCW_D1_AREA_PTR` is not used, the write buffer contains the D1 area.

You cannot set the D2-offset field. The D2-size field is set indirectly by one of the following:

- By calling the `write()` call without calling the `HIPIOCW_START_PKT ioctl()` creates a packet with a single `write()` call. When the D1 area is in the user buffer, the length of the `write()` call includes the D1 area, and the `write()` length must be at least as large as the D1 size. When `HIPIOCW_SEP_HDR` is set, the FP and D1 headers are included in the `write()` call. In this case, the D2 size is the length of the `write()` call less the headers.
- By using `HIPIOCW_START_PKT` to start a new packet of the specified D2 length. Multiple `write()` calls may be used to complete the packet. The packet will have the specified D2 length. When the D1 area is in the user buffer, the first `write()` call must be at least as long as the D1 area. When in separate-header mode, the first `write()` call must be at least the same size as the FP and D1 headers. Extra bytes from the `write()` calls are returned to the application. When the length is `D2SIZE_UNKNOWN`, `HIPIOCW_END_PKT` is used to terminate the packet.

When the CDI is set to combined-header-and-data mode, the entire packet, beginning with the FP header, is processed by using a `write()` call. None of the header manipulation `ioctl()` calls are available. The application must ensure that the FP header is in Big Endian and that the ULP is valid. The I-Field is always set by an `ioctl()`.

HIPPI-PH Mode

Headers are not used in the HIPPI-PH mode. Therefore, you can take one of two approaches to writing packets:

- By using the `HIPIOCW_START_PKT ioctl()` to specify the length of the packet, followed by one or more `write()` calls to send the packet
- By writing the packet using one or more `write()` calls and terminating the packet by calling the `HIPIOCW_END_PKT ioctl()`

The I-Field is set by using the `HIPIOCW_I` or `HIPIOCW_CONNECT ioctl()`.

Unknown Packet Sizes

In HIPPI-FP mode, the sender can start to send a packet before it determines the final length of the packet. The FP header's D2 field is set to `D2SIZE_UNKNOWN` (`-1` or `0xFFFFFFFF`) by the `HIPIOCW_START_PKT` `ioctl()`. The other `ioctl()` calls that are used to manipulate the FP header still operate. The first `write()` call of the packet must be long enough to pass all of the needed headers. When the packet is complete, `HIPIOCW_END_PKT` `ioctl()` is called to terminate the packet.

When the NIC is in FP mode, the packet must be a multiple of 8 bytes in length. When the NIC is in PH mode, the packet must be a multiple of 4 bytes in length. When a short, first burst is used, the first `write()` call writes the short, first burst. The remainder of the packet must be a multiple of 1024 bytes.

Short Bursts

HIPPI permits one short burst in a packet. The short burst can be either the first burst or the last burst. Usually, the short burst is at the end of the packet. However, the application can call `HIPIOCW_SHBURST` to force the first burst of a packet to be short.

A short, first burst is useful in communicating with hardware that can read the first burst, analyze the content, and pass the remainder of the data. Unless there is a hardware assist or some specific reason to use a short burst, you should not use it. Sun's HIPPI NIC performs the needed analysis of the header without requiring a short, first burst. Also, you should not use a short, first burst when the first burst is a full-size burst (for example, if D1 is 1016 bytes).

The short burst can be used in both HIPPI-PH and HIPPI-FP modes. In HIPPI-FP mode, the burst size is not used because it is calculated from the FP and D1 sizes. In HIPPI-PH mode, the burst size is set from the `ioctl()`. The short-burst size must be a multiple of 8 bytes in length and less than or equal to 1016 bytes. A standard HIPPI burst is 1024 bytes. When a short, first burst is used, the D2 area must be a multiple of 1024 bytes in length.

Transmit Queue

The CDI has a single transmit queue. Applications place the packets at the tail of the queue. The NIC takes packets from the head of the queue and sends them to their destination.

Because the queue is shared by all applications, access is restricted to one application at a time. The first (or only) `write()` call for a packet waits for the queue to become available. When the application is granted access, it holds the queue until the last packet for the connection is queued (not transferred). The application can generate packets by using multiple `write()` calls and multiple packets over the connection.

The application blocks the queue from the time that the `write()` call is called until the time when the NIC sends the data and the `write()` call completes. When sending packets using multiple `write()` calls, the application should perform the calls as quickly as possible. The transmit-idle-timeout limits the delay between calls. If data is not transmitted during the timeout period, the connection is broken, and the packet is discarded. You can set the transmit-idle-timeout by using the `hippitune(1M)` command. If the application does not want to wait for the transmit queue to become available, the `select()` call can be used to terminate the transmission (see “I/O Multiplexing” on page 42 for more information).

`write()` and `writev()` Calls

The `write()` and `writev()` calls can be used to send packets. The packet processing details are set before calling the `write()` call. See the host-system man pages for the general behavior of these interfaces.

All application buffers must be 8-byte aligned. All buffers for a packet, except for the last buffer, must be a multiple of 8 bytes in length. The last `write()` call can be any number of bytes in length if you send a packet with a known length.

Depending on the various conditions described above, the first `write()` call for a packet ensures that the I-Field and necessary headers precede the data. Subsequent `write()` calls for the packet pass packet data.

In HIPPI-FP mode, the packet must be the length that is specified in the D2-size field in the FP header. The packet is held open until the required number of bytes have been transferred. If the application attempts to write too many bytes, the extra bytes are returned by the `write()` call. (The return value from the `write()` call is not the same as the length.)

You can use one or more `write()` calls to send a single packet. When multiple `write()` calls are used, the packet length is set by an `HIPIOCW_START_PKT` `ioctl()`. If the application attempts to send more bytes than specified in the `HIPIOCW_START_PKT` `ioctl()`, the extra bytes are not sent, and the `write()` call returns the actual number of bytes that were sent. Multiple packets may not be sent in a single `write()` call.

You can send one or more packets over a single connection. By default, the connection is established when a packet is sent. The `HIPIOCW_CONNECT` `ioctl()` is used to open a connection for multiple packets (the connection is actually opened on

the write() call at the start of the next packet). When HIPIOCW_CONNECT is used, HIPIOCW_DISCONN must be used to close the connection when the last packet ends. If the packet has already ended, the connection is closed immediately.

When multiple write() calls are used or when unknown-length packets are sent, the length of each write() call must conform to the following rules:

- The first write() call must be long enough to pass all of the headers.
- The HIPPI-FP mode length, except for the last write() call, must be a multiple of 8 bytes in length.
- The HIPPI-PH mode length must be a multiple of 8 bytes, and the total length must be a multiple of 4 bytes (the length of the last write() call in the packet may be a multiple of 4 bytes).
- The length of the short, first-burst must be a multiple of 8 bytes, and the total length must be a multiple of 1024 bytes.

Process Interrupt

When a process that is sending is interrupted, the write() is truncated; a disconnect is performed; and, write() returns EIO. The next write() starts a new packet. The application must be ready to receive and process (usually discard) truncated packets.

When close() is called while sending a packet, the packet is truncated, and the connection is closed. No further packets can be sent because the transfer device is closed.

Transmit Errors

Errors in setting up the write() call (for example, a bad buffer pointer or invalid parameter) are returned immediately. The write() call returns -1 and errno contains the error code.

Errors in transmitting the data result in an EIO error. The packet is truncated, and a disconnect is performed. When this error occurs, the application can get additional information about the error by calling the HIPIOCW_ERR ioctl().

When the process is interrupted by using ^C or kill(1M), the packet is truncated, and a disconnect is performed. The errno value is set to EIO. The write() call cannot be restarted.

I/O Multiplexing

All `read()` calls and `write()` calls block the calling thread until data has been received or sent, respectively. (The CDI implements blocking of I/O). The application can use the `select()` call to avoid using the `read()` call until data is ready to be received. The application can also use the `select()` call to avoid blocking the transmission while another process is sending. The call to `select()` is described in the host-system man page. The CDI does not change the processing for the `select()` call.

Read Devices

The `select()` call for the `read()` call completes when data is available to be read. When multiple `read()` calls are used to process a packet, the `select()` call should be used only to wait for the arrival of the packet. It should not be called between successive `read()` calls of the same packet. If it is called while the application already has a connection open, it will return immediately with `EINVAL`.

When multiple processes are using the `select()` call for the same bound ULP, the process that has been waiting the longest will complete first. The other processes will continue to wait. This eliminates races for the receive queue.

Write Devices

The `select()` call for the `write()` call completes when the transmit queue is not busy. This does not guarantee that the queue will be available by the time that a `write()` call can be performed. When multiple transfer devices are using the `select()` call for the `write()` call, only the most recently queued `select()` call will complete.

When the application uses multiple `write()` calls in each packet or multiple packets for each connection, the `select()` call should be called only when the application is ready to send the first packet of a new connection. If it is called while the application already has a connection open, it will return immediately with `EINVAL`.

Exception Devices

The CDI does not support exception processing using the `select()` call.

4

Portability

This chapter explains the portability issues of the HIPPI network.

Application Portability

Applications may be made more portable by using system features that exist on all systems and observing hardware restrictions that exist on some systems.

Maximum `read()` and `write()` Length

Host systems place restrictions on the maximum length of a single `read()` or `write()` call. Systems can pass as much as 2 megabytes (2 times 1024 times 1024) in `read()` and `write()` calls. Because multiple `read()` and `write()` calls can be used in a single packet, this doesn't restrict packet size.

Buffer Alignment

To improve efficiency on some host systems, the buffers should be aligned to cache line boundaries (usually 32 bytes). For large transfers, the buffers should be aligned to page boundaries. Pages are usually 4096, 8192, or 16384 bytes.

Endian

The D1 and D2 areas in HIPPI-FP packets, and the entire HIPPI-PH packet, can be in either Big Endian or Little Endian. When both the source machine and the destination machine are the same endian, they should not have a problem transferring packets. When they have an opposite order, byte swapping may be needed. The applications should coordinate the packet contents and agree to the needed conversions.

Maximum Packet Length

Some systems limit the maximum length of a packet to 4096 times 4096 minus 8 bytes (`0xFFFFFFF8`).

5

CDI Reference

This chapter contains the Character Device Interface (CDI) reference pages, including the header file, the interface functions, the ioctls, and any special files.

In general, the interfaces behave as described in the host-system man pages. Specific details are presented in this document. Error codes that are returned in `errno` are sent to `/usr/include/sys/errno.h` on each host system. The general error codes are described in the host-system man pages. Specific error codes are described in this document.

Header File

The `ioctl.h` header file contains information that is needed by applications to call `ioctl()` and open special files. The header file is located in different directories on different host-system architectures.

Two utilities, `blast(1M)` and `sink(1M)` are distributed as source. These utilities provide examples of how the CDI can be used.

Interface Functions

The interface functions are standard, UNIX character device-driver functions. Host-system man pages describe the general behavior. Any unexpected processing is noted in the function description.

HIP_APP_OPEN Call

The HIP_APP_OPEN call is a macro that opens a transfer device on the given NIC. A transfer device can be opened by a single process at a time. Additional attempts to open the transfer device return EBUSY. This macro tries to open transfer devices until it succeeds or until all transfer devices have been tried. The macro calls open() and HIPIOC_GET_DEV ioctl().

Usage

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

#include <ioctl.h>

int fd_hippi, flags;

HIP_APP_OPEN(NIC0, flags, fd_hippi);
```

Arguments

The following arguments can be used:

- NIC0 – Represents the name of the control-device special file for the NIC. NIC0 represents the first NIC. ioctl.h defines the first 16 NICs (NIC0 to NIC15).
- flags – Contains the open() flags as defined in sys/fcntl.h. The application should only call HIP_APP_OPEN() with the flags that are really needed. The flags are O_RDONLY, O_WRONLY, O_RDWR. Unsupported flags are ignored.
- fd_hippi – Represents the file-descriptor variable of the newly opened CDI transfer device. It is not a pointer to the variable.

Failures and Errors

When an error occurs, fd_hippi is set to -1 and errno is set to the error code. The following list explains the possible errors:

- EBUSY – All special files have been tried and are busy.
- ENODEV – The NIC is not running.

close() Call

The close() call is used to close an open CDI device. See the close() man page for more details on the general operation of close().

When close() is called, any outstanding read() or write() calls are completed with EIO. Any incoming packet is truncated, and the connection is closed. A transmitting packet is also truncated, and the connection, including multiple-packet connections, is closed. Any packets for the special file that are in the transmit queue are returned with EIO.

If the close() call is the last UNBIND for a receive ULP, the ULP becomes invalid, and future packets to the ULP are discarded.

Usage

```
close(fd_hippi)
```

Arguments

fd_hippi – Represents the file descriptor of the CDI special file that is open.

Failures and Errors

The close() call has no special failures.

ioctl() Call

The ioctl() calls that control the operation of the CDI are described in “Ioctls” on page 53.

open() Call

The open() call is used to open a control device or open a transfer device for reading or writing. This call returns a file descriptor that is used for future calls to read(), write(), ioctl(), and close() (see the open() man page for more details).

The open() call of the control device (sub-device 0) for a NIC always succeeds as long as the name of the control device is valid and accessible. The open() call of transfer devices for the NIC fail if the NIC is stopped or the file is already open.

The open() call is not called directly by applications. Rather, HIP_APP_OPEN() is used to open a transfer device for processing.

Usage

```
#include <fcntl.h>

#include <sys/fcntl.h>

#include <sys/types.h>

int fd_hippi, flags;

static char path[] = "/dev/hippi/h0";

mode_t mode;

fd_hippi = open(path, flags, mode);
```

Arguments

The following arguments are supported:

- `fd_hippi` – Represents the file descriptor of a CDI special file that is open.
- `flags` – Can contain `O_RDONLY`, `O_WRONLY`, or `O_RDWR`. All other flags (in particular, `O_NDELAY`) are ignored.

Failures and Errors

The follow list explains the failures and errors that are associated with the `open()` call:

- `ENODEV` – The NIC is stopped.
- `EBUSY` – The device is already open.
- `ENOMEM` – The system ran out of memory while initializing the device.
- `EINVAL` – An invalid minor number was specified.

`read()` and `readv()` Calls

The `read()`, or `readv()`, call reads data from the HIPPI device (see the `read()` man page for more details). The supplied buffers must be 8-byte aligned. The last `read()` of a packet can be any length; all other `read()` calls must be a multiple of 8 bytes in length.

The application must validate the received packet because any source can send to any destination. The application supplied buffer can be any length and start at any byte offset. You can obtain a minor performance advantage in aligning the buffer to

a 32-byte (cache line) boundary for small packets and to a page boundary (usually 4096, 8192, or 16384 bytes) for large packets. Some host systems run faster by using buffers that are a length that is a multiple of the cache-line.

The `read()` call blocks transmission until data is available in HIPPI-PH mode or until data is available for the bound ULP in HIPPI-FP mode. The `read()` call returns the number of bytes that have been read. The content of the data is determined by the `ioctl()` calls that have been used to condition the `read()` call.

In combined-header-and-data mode (the default), the `read()` call passes the entire packet, starting with the FP header. Because multiple `read()` calls can be used for a packet, the FP header can be read with a `read()` call that is 8 bytes in length. The FP header includes the D1 size and D2 size and is in Big Endian.

In separate-header-and-data mode, the `read()` call returns data from the D2 area. The `HIPIOCR_GET_FP` `ioctl()` is used to retrieve the FP header. The entire packet must be processed. When the data is not needed, the `HIPIOCR_TRUNCATE_PKT` `ioctl()` can be used to discard the remainder of the packet. If data arrives, and if there is no `read()` pending, the NIC controls the flow of the HIPPI network until the data is read or the Receive-Idle-Timeout truncates the packet.

In HIPPI-FP mode, packets to unbound ULPs are discarded. When a packet of unknown length is being received, use the `HIPIOCR_PKT_OFFSET` `ioctl()` to detect the end of the packet. If an `EIO` error occurs, the `HIPIOCR_ERRS` `ioctl()` can be used to get more detailed information.

Usage

```
#include <unistd.h>

int fd_hippi, len;

char buf[BUF_SIZE];

len = read(fd_hippi, buf, BUF_SIZE);
```

Arguments

The following arguments are supported by the `read()` and `readv()` calls:

- `fd_hippi` – The file descriptor of a CDI special file that is open
- `buf` – The user-space buffer that will receive the data
- `BUF_SIZE` – The size of the buffer
- `len` – The actual number of bytes that were read (`-1` if an error occurred)

Failures and Errors

The following lists explains the failures and errors that can occur:

- ENODEV – The NIC is stopped.
- EINVAL – The read of a sub-device returned zero (0).
- EINVAL – An invalid minor number was specified.
- EINVAL – The device was not open for O_RDONLY or O_RDWR.
- EINVAL – The HIPPI-PH mode and length was not a multiple of 8 bytes in length and greater than zero (0).
- EIO – A data reception error occurred (use HIPIOCR_ERRS for more details).
- EIO – A process was killed (^C).

select() Call

The select() call enables the application to determine if data is waiting to be read or if the transmit queue is empty for write() (see the select() man page for more details).

The select() for read() completes when a packet is available to be read. When multiple processes are waiting for the same ULP, the process that has been waiting the longest will complete. The others will continue to wait.

The select() for write() completes when the send queue is free. If multiple processes are waiting for the same ULP, the process that has been waiting the longest completes. The other processes continue to wait.

The FD_SET() macro in sys/select.h can be used to set the file descriptor bits in readfds and writefds. HIPPI file descriptors can be intermixed with other file descriptors (for example, files and sockets).

The details of calling select() are specific to the host system. The following is OSF1.

Usage

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

#include <sys/time.h>

#include <sys/select.h>

int fds, fd_hippi, width;

struct timeval timeout;

fd_set readfds, writefds;
```

```
FD_ZERO(&writefds);

FD_SET(fd_hippi, &writefds);

FD_ZERO(&readfds);

FD_SET(fd_hippi, &readfds);

width = fd_hippi + 1;

timeout.tv_sec = 1;

timeout.tv_usec = 0;

fds = select(width, &readfds, &writefds, NULL, &timeout);
```

Arguments

The following arguments are supported:

- `width` – The maximum number of descriptors to check (the highest file descriptor is `+1`)
- `readfds` – The `FD_SET` that refers to all of the file descriptors that check for available data
- `writefds` – The `FD_SET` that refers to all of the file descriptors that check to see if the transmit queue is empty
- `timeout` – The timeout value for `select()` (if conditions do not change during this time, `select()` returns zero)
- `fds` – The number of file descriptors that are ready to be processed

Failures and Errors

The `select()` call has no special failures or errors.

`write()` and `writev()` Calls

The `write()`, and `writev()`, call is used to write data to the HIPPI device (see the `write()` man page for more details on general operation). The `write()` call accepts a single buffer, and the `writev()` call accepts a vector of buffers. The supplied buffers must be 8-byte aligned. The last `write()` of a packet can be any length; all other `write()` calls must be a multiple of 8 bytes in length. A minor performance advantage can be achieved by aligning the buffer to a 32-byte (cache line) boundary for small packets and to a page boundary (usually 4096, 8192, or 16384 bytes) for large packets. Another minor performance advantage can be achieved by making buffers a multiple of the cache-line size.

Generally, the `write()`, and `writev()`, call writes the data immediately to the device and then returns its status. However, there are cases where internal states, such as network flow control and long transfers from other applications, can cause the `write()` call to block the transmission. The `write()` call does not return an exit status until the data is written or an error has occurred.

The `write()` call's operation is based on the `ioctl()` calls that have been used to condition packet transmission. The packet length is the length of the `write()` call unless the `HIPIOCW_START_PKT` `ioctl()` is used to specify a length. If the application attempts to send too much data, the `write()` call returns the actual number of bytes that were sent (the extra data is not sent). The packet remains open until all of the expected data is sent.

In HIPPI-FP mode (when `D2SIZE_UNKNOWN` packet length is specified), or in HIPPI-PH mode, the `HIPIOCW_END_PKT` `ioctl()` is used to terminate the packet. Also, every `write()` call for the packet must be a multiple of 8 bytes in length. In HIPPI-PH mode, the length of the `write()` call must be greater than zero (0) and a multiple of 4 bytes.

When a short, first burst is used, the application must write bursts that are a multiple of 1024-bytes in length for the remainder of the packet. This requirement exists because a packet can have only one short burst. Fill bytes at the end of the D2 area are undefined; they are not zero (0).

The Transmit-Idle-Timeout is active from the start of the first `write()` call of a connection until the connection is closed. If the timeout is exceeded, the packet is truncated; the connection is dropped, and `EIO` is returned. When `EIO` is returned, you can use `HIPIOCW_ERR` to get more details.

Usage

```
#include <unistd.h>

int fd_hippi, len;

char buf[BUF_SIZE];


len = write(fd_hippi, buf, BUF_SIZE);
```

Arguments

The following arguments are supported:

- `fd_hippi` – The file descriptor of a CDI special file that is open
- `buf` – The user-space buffer that will receive the data

- BUF_SIZE – The size of the buffer in bytes
- len – The actual number of bytes that were written (-1 if an error occurred)

Failures and Errors

The following list explains the failures and errors that can occur:

- ENODEV – The NIC is stopped.
- EINVAL – The write() call to a sub-device returned zero (0).
- EINVAL – An invalid minor number was specified.
- EINVAL – The device was not open for O_WRONLY or O_RDWR.
- EINVAL – The HIPPI-PH mode and length is not a multiple of 4 bytes and is not greater than zero (0).
- EIO – A data reception error occurred (use HIPIOCW_ERR for more details).
- EIO – The process was killed (^C).

Ioctls

The ioctl() calls in this section are used to control the CDI.

HIPIOC_BIND_ULP Call

This ioctl() is used to bind a receive ULP to the transfer device. A ULP must be bound to the transfer device before packets can be read. Multiple devices can be bound to the same ULP. The device file must be open for O_RDONLY or O_RDWR. In addition, the NIC must be running in HIPPI-FP mode. Packets that arrive for a ULP before the ULP is bound are discarded.

A ULP may be bound for exclusive use of the process. Alternatively, the ULP may be bound to more than one transfer device. A positive ULP_id requests an exclusive bind, and a negative ULP_id permits sharing the bound ULP with other applications. There can be only one ULP bound to a transfer device at a time, and a transfer device can accept packets to a single ULP.

ULPs in the range of 128 to 255 may be bound by any application program. ULPs 2, 3, 4, 6, 7, 8, and 10 are reserved by the HIPPI-FP standard for specific purposes and can not be bound by a user or root application. The remaining ULPs in the range of 0 to 127 are reserved by the HIPPI-FP standard for future use. They may be bound to applications that have root privilege.

When used with HIPPI-PH, the ULP_id value HIPPI_ULP_PH or HIPPI_ULP_PH_EXCL(1) requests an exclusive bind; ULP_id HIPPI_ULP_PH_SHR requests a non-exclusive bind.

If HIPIOCW_SET_ULP has not been executed, the ULP can also be used as the transmit ULP. A maximum of 31 different ULPs may be concurrently bound in FP mode.

Usage

```
#include <ioctl.h>

int fd_hippi, ret, ULP_id;


ret = ioctl(fd_hippi, HIPIOC_BIND_ULP, ULP_id);
```

Arguments

The following arguments are supported:

- fd_hippi – The file descriptor of a CDI transfer device that is open
- ULP_id – The ULP that is being bound (When the ULP_id is greater than zero (0), the bind is exclusive. No other devices can be concurrently bound. When the ULP_id is less than zero (0), the bind is not exclusive, so other devices may be bound.)

Failures and Errors

The following list explains the failures and errors that can occur:

- EINVAL – The ULP_id is greater than 255.
- EPERM – The ULP_id is less than 128 and is not superuser.
- EPERM – The ULP is reserved and defined (2, 3, 4, 6, 7, 8, 10).
- EINVAL – The ULP is already bound to this device.
- EBUSY – The ULP is already bound to another device and an exclusive bind is requested.
- ENODEV – The NIC is not running.
- EINVAL – The device was not open for O_RDONLY or O_RDWR.
- EINVAL – The fd_hippi argument is a control device (sub-device zero).
- EINVAL – The NIC is in HIPPI-PH mode, and the ULP is not HIPPI_ULP_PH, HIPPI_ULP_PH_EXCL, or HIPPI_ULP_PH_SHR.
- EINVAL – The maximum number of ULPs are already bound.

HIPIOC_GET_DEV Call

This ioctl() is not called directly by the application. It is called by the HIP_APP_OPEN() macro. The ioctl() returns the name of an unused transfer device that can be opened by the application. However, the application must be prepared to retry the call because the transfer device can be opened by another application before the first application transmits the open() call.

Usage

```
#include <ioctl.h>

int fd_hippi, ret, device;


ret = ioctl(fd_hippi, HIPIOC_GET_DEV, &device);
```

Arguments

The following arguments are supported:

- fd_hippi – The file descriptor of a CDI transfer device that is open
- device – The device number that was returned

Failures and Errors

The following list explains the failures and errors that can occur:

- EINVAL – The fd_hippi descriptor is not an open control device.
- EBUSY – No devices are free.
- ENODEV – The NIC is not running.

HIPIOC_GET_DEVICE_STATE Call

This ioctl() returns the operating mode bits for the NIC.

Usage

```
#include <ioctl.h>

int fd_hippi, ret, status;
```

```
ret = ioctl(fd_hippi, HIPIOCR_GET_DEVICE_STATE, &status);
```

Arguments

The following arguments are supported by the `HIPIOCR_GET_DEVICE_STATE` call:

- `fd_hippi` – The file descriptor of a CDI transfer device that is open
- `status` – The operating mode of the device

The following list explains the possible operating modes:

- `HIP_ON` (`0x01`) – State of the NIC (running or stopped)
- `HIP_LONG` (`0x02`) – Long packets (or short packets—64 kilobytes)
- `HIP_PH` (`0x04`) – HIPPI-PH mode (or HIPPI-FP mode)
- `HIP_LOOPBACK` (`0x08`) – Internal loop-back mode (or network mode)
- `HIP_DIRECT` (`0x10`) – Direct connect mode (or switched mode)

Failures and Errors

If `ret` is `-1`, the following error occurred in processing the `ioctl()`. In this case `err_status` does not contain valid data.

- `ENODEV` – The NIC is not running, and this is not the controlling device.

`HIPIOC_GET_NICS` Call

This `ioctl()` returns the number of NICs in the configuration.

Usage

```
#include <essioctl.h>

int fd_hippi, ret, nics;


ret = ioctl(fd_hippi, HIPIOCR_GET_NICS, &nics);
```

Arguments

The following arguments are supported:

- fd_hippi – The file descriptor of any open CDI control device or CDI transfer device
- nics – The number of NICs in the configuration

Failures and Errors

The HIPIOCR_GET_NICS() call has no special failures or errors.

HIPIOC_UNBIND_ULP Call

This ioctl() is used to unbind the currently bound receive ULP. There may be multiple devices bound to the same ULP. When the last device is unbound, the ULP is considered unbound, and future packets to the ULP are discarded by the NIC. The transfer device must be open for O_RDONLY or O_RDWR.

HIPIOC_UNBIND_ULP cannot be executed while the process is receiving a packet (for instance, if the process is between reads of a multiple-read packet.)

When hippi off is executed, all ULPs are unbound. When close() is executed, the ULP is unbound.

Usage

```
#include <ioctl.h>

int fd_hippi, ret;


ret = ioctl(fd_hippi, HIPIOC_UNBIND_ULP);
```

Argument

The following arguments are supported:

- fd_hippi – The file descriptor of a CDI transfer device that is open

Failures and Errors

The following list explains the failures and errors that can occur:

- EINVAL – The ULP is not bound to this device.
- ENODEV – The NIC not running.

- EINVAL – The device is not open for O_RDONLY or O_RDWR.
- EINVAL – fd_hippi is a controlling device (sub-device 0).

HIPIOCR_EIO Call

This ioctl() sets the receive error processing mode. When the argument is zero (0), the default, a read() error results in the read() call returning -1 and errno being set to EIO. This results in the loss of whatever data had been received up to the error. When the argument is not zero (0), EIO is never returned, and all of the data that is available in the NIC is passed to the application. In this mode, the application must call HIPIOCR_ERRS after each read() call to determine the read() completion code.

Usage

```
#include <ioctl.h>

int fd_hippi, ret, mode;


ret = ioctl(fd_hippi, HIPIOCR_EIO, mode);
```

Arguments

The following arguments are supported:

- fd_hippi – The file descriptor of a CDI transfer device that is open
- mode – Mode setting that determines processing when data errors occur (When mode is zero (0), EIO is generated if a data error occurs. When mode is not zero, EIO is not generated if a data error occurs.)

Failures and Errors

The following list explains the failures and errors that can occur:

- ENODEV – The NIC not running.
- EINVAL – The device is not open for O_RDONLY or O_RDWR.
- EINVAL – fd_hippi is a controlling device (sub-device 0).

HIPIOCR_ERRS Call

This ioctl() can be called after a read() call to provide detailed information about an EIO error from the most recent read() call.

Usage

```
#include <ioctl.h>

int fd_hippi, ret, err_status;


ret = ioctl(fd_hippi, HIPIOCR_ERRS, &err_status);
```

Arguments

The following arguments are supported:

- fd_hippi – The file descriptor of a CDI transfer device that is open
- err_status – The result of the last read() call (Multiple bits can be set. A value of zero (0) indicates that no error occurred.)

The following list contains the possible err_status values:

- HIP_DSTERR_PARITY (0x001) – Destination parity error
- HIP_DSTERR_LLRC (0x002) – Destination LLRC error
- HIP_DSTERR_FRAME (0x004) – Frame error (Glink)
- HIP_DSTERR_SYNC (0x008) – Flag sync error (Glink)
- HIP_DSTERR_ILBURST (0x010) – Destination illegal burst error
- HIP_DSTERR_SDIC (0x020) – State error
- HIP_DSTERR_SHBST (0x040) – Unexpected short burst
- HIP_DSTERR_RDY (0x080) – Lost Link ready (Glink)
- HIP_DSTERR_RXTO (0x100) – Receive timeout
- HIP_DSTERR_PKTLEN (0x200) – Packet length error

Failures and Errors

The following list contains the failures and errors that can occur:

- The ioctl() returned zero (0) when an EIO error has not occurred in the current packet.

If ret is -1, one of the following errors occurred in processing the ioctl(). In this case err_status does not contain valid data.

- EINVAL – fd_hippi is a controlling device.

- EINVAL – fd_hippi is not bound to a ULP.
- EINVAL – fd_hippi is not open for O_RDONLY or O_RDWR.
- ENODEV – The NIC is not running.

HIPIOCR_GET_D1 Call

This ioctl() returns the D1 area from the most recently received packet. The NIC must be in HIPPI-FP mode. The device must be in receive-separate-header mode, HIPIOCR_SEP_HDR. The size of the D1 area in the FP header is used to control the transfer. The application should provide a 1016-byte buffer for the D1 area (maximum size). The FP header must be read first. HIPIOCR_GET_D1 cannot be called after a read() call.

Usage

```
#include <ioctl.h>

int fd_hippi, ret;

char buf[1016];


ret = ioctl(fd_hippi, HIPIOCR_GET_D1, buf);
```

Arguments

The following arguments are supported:

- fd_hippi – The file descriptor of a CDI transfer device that is open
- buf – The buffer that will receive the D1 area

Failures and Errors

The following list contains the failures and errors that can occur:

- EINVAL – fd_hippi is a controlling device.
- EINVAL – fd_hippi is not open for O_RDONLY or O_RDWR.
- EINVAL – The ULP is not bound.
- EINVAL – The system is not in HIPPI-FP mode.
- ENODEV – The NIC is not running.

HIPIOCR_GET_FP Call

This ioctl() returns the FP header from the most recently received packet. The header is in Big Endian order. The NIC must be in HIPPI-FP mode.

The device must be in receive-separate-header mode (HIPIOCR_SEP_HDR). The application should provide an 8-byte buffer for the FP header. The returned FP header is in Big Endian.

When called, HIPIOCR_GET_FP must be called before a read() call. This ioctl() blocks transmission until a packet arrives for the bound ULP.

Usage

```
#include <ioctl.h>

int fd_hippi, ret;

unsigned int fp[2];


ret = ioctl(fd_hippi, HIPIOCR_GET_FP, fp);
```

Arguments

The following arguments are supported:

- fd_hippi – The file descriptor of a CDI transfer device that is open
- fp – The buffer that will receive the FP header

Failures and Errors

The following list contains the failures and errors that can occur:

- EINVAL – fd_hippi is a controlling device.
- EINVAL – fd_hippi is not open for O_RDONLY or O_RDWR.
- EINVAL – The ULP is not bound.
- EINVAL – The network is not in HIPPI-FP mode.
- ENODEV – The NIC is not running.

HIPIOCR_PKT_OFFSET Call

This ioctl() monitors the number of bytes that have been read for the packet. In separate-headers-and-data mode, it contains the number of D2 bytes that have been read. In combined-headers-and-data mode, it contains the total number of bytes that have been read. When the whole packet has been read, zero (0) is returned, indicating the start of a new packet.

The maximum count that is recorded is MAX_PKT_OFFSET bytes. After that, the counter does not increment, even if the packet is longer. The file must be open for O_RDONLY or O_RDWR, and a ULP must be bound.

Usage

```
#include <ioctl.h>

int fd_hippi, ret;

unsigned int offset;


ret = ioctl(fd_hippi, HIPIOCR_PKT_OFFSET, &offset);
```

Arguments

The following arguments are supported:

- fd_hippi – The file descriptor of a CDI transfer device that is open
- offset – The byte offset into the packet (The maximum reported value is MAX_PKT_OFFSET bytes. If the packet is longer than MAX_PKT_OFFSET, the counter stops advancing. When the whole packet has been read, the offset is zero).

Failures and Errors

The following list contains the failures and errors that can occur:

- EINVAL – fd_hippi is a controlling device.
- EINVAL – fd_hippi is not bound to a ULP.
- EINVAL – fd_hippi is not open for O_RDONLY or O_RDWR.
- ENODEV – The NIC is not running.

HIPIOCR_SEP_HDR Call

This ioctl() sets the operating mode to use separate-headers-and-data mode or to use combined-headers-and-data (the default) for the transfer device. This is effective in HIPPI-FP mode only.

In separate-header-and-data mode, the FP header is read by using the HIPIOCR_GET_FP ioctl(), and the D1 area is read by using the HIPIOCR_GET_D1 ioctl(). In combined-header-and-data mode, the application buffer contains the complete packet, the FP header, the D1 area (optional), and the D2 area (optional).

Usage

```
#include <ioctl.h>

int fd_hippi, ret, mode;


ret = ioctl(fd_hippi, HIPIOCR_SEP_HDR, mode);
```

Arguments

The following arguments are supported:

- fd_hippi – The file descriptor of a CDI transfer device that is open
- mode – The operating mode (When it is zero (0), the separate-header-and-data mode is being used. When it is not zero (0), the combined-header-and-data mode is being used—the default.

Failures and Errors

The following list contains the failures and errors that can occur:

- EINVAL – fd_hippi is a controlling device.
- EINVAL – fd_hippi is not open for O_RDONLY or O_RDWR.
- ENODEV – The NIC is not running.
- EINVAL – The NIC is in HIPPI-PH mode.

HIPIOCR_TRUNCATE_PKT Call

This ioctl() discards the remainder of the current receive packet. The file must be open for O_RDONLY or O_RDWR, and a ULP must be bound. If a packet is not present, zero (0) is returned.

Usage

```
#include <ioctl.h>

int fd_hippi, ret;


ret = ioctl(fd_hippi, HIPIOCR_TRUNCATE_PKT);
```

Arguments

The following arguments are supported:

- fd_hippi – The file descriptor of a CDI transfer device that is open

Failures and Errors

The following list contains the failures and errors that can occur:

- EINVAL – fd_hippi is a controlling device.
- EINVAL – fd_hippi is not bound to a ULP.
- EINVAL – fd_hippi is not open for O_RDONLY or O_RDWR.
- ENODEV – The NIC is not running.

HIPIOCW_CONNECT Call

This ioctl(), and HIPIOCW_I, sets the I-Field in the cache. The I-Field is cached in the CDI and is used with every packet that is sent until it is changed again. HIPIOCW_I cannot be called while a multiple-packet connection is open. The default I-Field is zero (0). Note that when you are using a single-packet connection, you must use HIPIOCW_I to set the I-Field.

This ioctl() causes a multiple-packet connection to be established when the next write() call is executed. The argument is the I-Field, which is placed in the CDI cache. The application must ensure that the I-Field is in Big Endian byte order (Little Endian machines will likely have to byte swap the I-Field).

The connection remains open, and the send lock is held until HIPIOCW_DISCONN is executed, an error occurs, or the transfer device is closed.

The file must be open for O_WRONLY or O_RDWR. The Transmit-Idle-Timeout is in effect during this connection while sending packets and between successive packets. This ioctl() cannot be called while a connection is open for the transfer device.

Usage

```
#include <ioctl.h>

int fd_hippi, ret;

unsigned int IField;


ret = ioctl(fd_hippi, HIPIOCW_CONNECT, IField);
```

Arguments

The following arguments are supported:

- fd_hippi – The file descriptor of a CDI transfer device that is open
- IField – The I-Field that is to be used with packets that are sent over the connection

Failures and Errors

The following list contains the failures and errors that can occur:

- EINVAL – fd_hippi is a controlling device.
- EINVAL – fd_hippi is not open for O_WRONLY or O_RDWR.
- EBUSY – Multiple-packet connection is already requested. The connection is not established if the write() call has not been done.
- EBUSY – A packet is currently being sent.
- ENODEV – The NIC is not running.

HIPIOCW_D1_AREA Call

This HIPPI-FP ioctl() copies the D1 area from the user buffer to the CDI's D1 cache. The length is taken from the currently cached FP header. The ioctl() is valid in HIPPI-FP mode only. To set the D1 area's length, HIPIOCW_D1_SIZE must be called before HIPIOCW_D1_AREA. The HIPIOCW_D1_SIZE ioctl() clears the D1-area pointer that is currently cached.

Both HIPIOCR_D1_AREA ioctl() and HIPIOCR_D1_AREA_PTR ioctl() manage the D1 area for a packet. The last ioctl() called is used on all subsequent packets until it is changed. The ioctl() must not be called while a packet is being sent.

Usage

```
#include <ioctl.h>

int fd_hippi, ret;

char d1_buf[1016], d1_ptr = &d1_buf;


ret = ioctl(fd_hippi, HIPIOCW_D1_AREA, &d1_ptr);
```

Arguments

The following arguments are supported:

- fd_hippi – The file descriptor of a CDI transfer device that is open
- d1_buf – Pointer to the application-supplied, D1 buffer (NULL causes the cached D1 image to be discarded, causing the D1 area to be set to zero.)

Failures and Errors

The following list contains the failures and errors that can occur:

- EINVAL – fd_hippi is a controlling device.
- EINVAL – fd_hippi is not open for O_WRONLY or O_RDWR.
- EINVAL – D1 size is zero (0).
- EBUSY – A packet is being sent.
- EINVAL – The NIC is in HIPPI-PH mode.
- ENODEV – The NIC is not running.

HIPIOCW_D1_AREA_PTR Call

This HIPPI-FP ioctl() caches a pointer to the application-supplied, D1-area buffer. Whenever a packet is sent, the D1 area is copied to the CDI's D1 cache and included in the packet.

Both HIPIOCR_D1_AREA ioctl() and HIPIOCR_D1_AREA_PTR ioctl() manage the D1 area for a packet. The last ioctl() called is used on all future packets until it is changed. The size of the D1 area is set by using the HIPIOCW_D1_SIZE ioctl(). When the write() call is executed, the D1 area is copied to the D1 cache in the driver. The ioctl() must not be called while a packet is being sent.

Usage

```
#include <ioctl.h>

int fd_hippi, ret;

char d1_buf[1016], d1_ptr = &d1_buf;


ret = ioctl(fd_hippi, HIPIOCW_D1_AREA_PTR, &d1_ptr);
```

Arguments

The following arguments are supported:

- fd_hippi – The file descriptor of a CDI transfer device that is open
- d1_buf – Pointer to application supplied D1 buffer (NULL causes the cached D1 pointer to be discarded.)

Failures and Errors

The following list contains the failures and errors that can occur:

- EINVAL – fd_hippi is a controlling device.
- EINVAL – fd_hippi is not open for O_WRONLY or O_RDWR.
- EINVAL – D1 size is zero (0).
- EBUSY – A packet is being sent.
- EINVAL – The NIC is in HIPPI-PH mode.
- ENODEV – The NIC is not running.

HIPIOCW_D1_SIZE Call

This ioctl() is valid in HIPPI-FP mode only. It sets the size of the D1 area and, when D1 is not zero (0), it also sets the HIPPI-FP P bit (see Appendix C "HIPPI-FP Excerpts" for details about the HIPPI-FP header). The D1 area may be located in the user buffer (default), cached in the driver by calling the HIPIOCW_D1_AREA ioctl(), or maintained in a separate user-buffer by calling the HIPIOCW_D1_AREA_PTR ioctl(). In all cases, the size of the D1 area is set by HIPIOCW_D1_SIZE ioctl(). The D1 size must be a multiple of 8 bytes in length, and it must be less than or equal to 1016 bytes.

The HIPIOCW_D1_AREA ioctl() must be called before HIPIOCW_D1_AREA or HIPIOCW_D1_AREA_PTR is called. In addition, it must not be called while a packet is being sent.

Usage

```
#include <ioctl.h>

int fd_hippi, ret, size;


ret = ioctl(fd_hippi, HIPIOCW_D1_SIZE, size);
```

Arguments

The following arguments are supported:

- fd_hippi – The file descriptor of a CDI transfer device that is open
- size – The size of the D1 area in bytes

Failures and Errors

- EINVAL – fd_hippi is a controlling device.
- EINVAL – fd_hippi is not open for O_WRONLY or O_RDWR.
- EINVAL – The NIC is in HIPPI-PH mode.
- EBUSY – A packet is being sent.
- EINVAL – The size is invalid.
- ENODEV – The NIC is not running.

HIPIOCW_DISCONN Call

This ioctl() disconnects a connection that was opened by HIPIOCW_CONNECT. The disconnect happens immediately. If a packet is being transmitted, it is truncated. If packets have not been sent, this ioctl() returns to one-packet-per-connection mode.

Note – To terminate a packet without terminating the connection, use HIPIOCW_END_PKT.

Usage

```
#include <ioctl.h>

int fd_hippi, ret;


ret = ioctl(fd_hippi, HIPIOCW_DISCONN);
```

Arguments

The following arguments are supported:

- fd_hippi – The file descriptor of a CDI transfer device that is open

Failures and Errors

The following list contains the failures and errors that can occur:

- EINVAL – fd_hippi is a controlling device.
- EINVAL – fd_hippi is not open for O_WRONLY or O_RDWR.
- EINVAL – There is no multiple-packet connection pending.
- ENODEV – The NIC is not running.

HIPIOCW_END_PKT Call

This ioctl() terminates the current packet. It is usually used to terminate HIPPI-PH packets and HIPPI-FP packets that have a D2 size of D2SIZE_UNKNOWN. When it is used to prematurely terminate a HIPPI-FP packet that has a specified D2 size, it causes the packet to have an invalid D2 size that appears to the destination as a truncated packet. The device must be open for O_WRONLY or O_RDWR.

Usage

```
#include <ioctl.h>

int fd_hippi, ret;


ret = ioctl(fd_hippi, HIPIOCW_END_PKT);
```

Arguments

The following arguments are supported:

- `fd_hippi` – The file descriptor of a CDI transfer device that is open

Failures and Errors

The following list contains the failures and errors that can occur:

- `EINVAL` – `fd_hippi` is a controlling device.
- `EINVAL` – `fd_hippi` is not open for `O_WRONLY` or `O_RDWR`.
- `EINVAL` – A packet is not being sent.
- `ENODEV` – The NIC is not running.
- `EIO` – A packet was truncated.

`HIPIOCW_ERR` Call

This `ioctl()` returns the error status from the most recent `write()` call.

Usage

```
#include <ioctl.h>

int fd_hippi, ret, err_status;


ret = ioctl(fd_hippi, HIPIOCW_ERR, &err_status);
```

Arguments

The following arguments are supported:

- fd_hippi – The file descriptor of a CDI transfer device that is open
- err_status – The result of the last read()

The following list explains the values that appear in err_status:

- HIP_SRCERR_NONE (0) – No error
- HIP_SRCERR_SEQ (1) – HIPPI source-sequence error
- HIP_SRCERR_DSIC (2) – Lost DSIC (source)
- HIP_SRCERR_TIMEO (3) – Transmit time-out (source)
- HIP_SRCERR_CONNLS (4) – CONNECT-signal loss (source)
- HIP_SRCERR_RE (5) – Rejected connect request
- HIP_SRCERR_SHUT (6) – Interface shut down
- HIP_SRCERR_SW (7) – Unexpected driver failure

Failures and Errors

If ret is -1, one of the following errors occurred in processing the ioctl(). In this case err_status does not contain valid data.

The following list contains the failures and errors that can occur:

- EINVAL – fd_hippi is a controlling device.
- EINVAL – fd_hippi is not open for O_WRONLY or O_RDWR.
- ENODEV – The NIC is not running.

HIPIOCW_I Call

This ioctl(), and HIPIOCW_CONNECT, sets the I-Field in the cache. The I-Field is cached in the CDI and is used with every packet that is sent until it is changed again.

The argument is the new I-Field. The application must ensure that the I-Field is in Big Endian byte order (Little Endian machines will likely have to byte swap the I-Field). The default I-Field is zero (0).

The file must be open for O_WRONLY or O_RDWR. The application must not be in the process of sending a packet or have a multiple-packet connection active.

Note – If you are using a multiple-packet connection, use HIPIOCW_CONNECT to set the I-Field.

Usage

```
#include <ioctl.h>
```

```
int fd_hippi, ret;
unsigned int IField;


ret = ioctl(fd_hippi, HIPIOCW_I, IField);
```

Arguments

The following arguments are supported:

- `fd_hippi` – The file descriptor of a CDI transfer device that is open
- `IField` – The I-Field that is to be placed in the CDI cache (`IField` must be in Big Endian order.)

Failures and Errors

The following list contains the failures and errors that can occur:

- `EINVAL` – `fd_hippi` is a controlling device.
- `EINVAL` – `fd_hippi` is not open for `O_WRONLY` or `O_RDWR`.
- `ENODEV` – The NIC is not running.
- `EBUSY` – A packet is being sent.
- `EBUSY` – Multiple-packet connection is active.
- `EINVAL` – Multiple-packet connection is currently being used.

`HIPIOCW_SEP_HDR` Call

This `ioctl()` sets separate-header-and-data mode (default) or combined-headers-and-data mode for the transfer device. This is effective in HIPPI-FP mode only. This `ioctl()` must not be called while a packet is being sent.

In separate-header-and-data mode, the FP header is taken from the CDI FP header cache. When the D1 size is not zero (0), the D1 header is placed in the packet. If the CDI's D1 cache contains a header (`HIPIOCW_D1_AREA` has been called), the header is included in the packet. Alternatively, the D1 area can be in an application buffer, and `HIPIOCW_D1_AREA_PTR` can be called to cache the pointer in the CDI.

In combined-header-and-data mode, the application buffer contains the complete packet, and the driver header cache is cleared. The application-supplied buffers contain the FP header, the D1 area (optional), and the D2 area (optional). The application must ensure that the FP header is in Big Endian order. The driver validates the ULP field in the FP header. The RunCode does more extensive analysis and can cause the operation to fail.

Usage

```
#include <ioctl.h>

int fd_hippi, ret, mode;


ret = ioctl(fd_hippi, HIPIOCW_SEP_HDR, mode);
```

Arguments

The following arguments are supported:

- fd_hippi – The file descriptor of a CDI transfer device that is open
- mode – Header mode

Failures and Errors

The following list contains the failures and errors that can occur:

- EINVAL – fd_hippi is a controlling device.
- EINVAL – fd_hippi is not open for O_WRONLY or O_RDWR.
- EINVAL – The NIC is in HIPPI-PH mode.
- EBUSY – A packet is being sent.
- ENODEV – The NIC is not running.

HIPIOCW_SET_ULP Call

This ioctl() sets the ULP in the cached, FP header for the device. The value is used in successive packets until it is changed by another call to HIPIOCW_SET_ULP. This ioctl() is valid only in HIPPI-FP mode. If HIPIOC_BIND_ULP is called before HIPIOCW_SET_ULP, the bound ULP will be placed in the FP header cache.

The device file must be open for O_WRONLY or O_RDWR, and the NIC must be running in HIPPI-FP mode. In addition, the CDI must not be actively sending a packet, and it must be in separate-header-and-data mode for write() calls.

ULPs in the range of 128 to 255 can be bound by any application program. ULPs 2, 3, 4, 6, 7, 8, and 10 are reserved for specific purposes and can not be bound by a user application. The remaining ULPs in the range of 0 to 127 are reserved for future use. They can be bound to applications that have root privilege.

Usage

```
#include <ioctl.h>

int fd_hippi, ret, ULP_id;


ret = ioctl(fd_hippi, HIPIOC_SET_ULP, ULP_id);
```

Arguments

The following arguments are supported:

- `fd_hippi` – The file descriptor of a CDI transfer device that is open
- `ULP_id` – The ULP that is being set

Failures and Errors

The following list contains the failures and errors that can occur:

- `EINVAL` – The NIC is not in separate-header-and-data mode.
- `EBUSY` – A packet is being sent.
- `EINVAL` – The ULP is reserved.
- `EINVAL` – The ULP ID is greater than 255.
- `EPERM` – The ULP ID is less than 128 and the application does not have superuser privileges.
- `ENODEV` – The NIC is not running.
- `EINVAL` – The device is not open for `O_WRONLY` or `O_RDWR`.
- `EINVAL` – `fd_hippi` is a controlling device.
- `EINVAL` – The NIC is in HIPPI-PH mode.

`HIPIOCW_SHBURST` Call

This `ioctl()` causes the first burst of the next packet that is sent to be the short burst. The application must ensure that the remainder of the packet is a multiple of 1024 bytes in length. `HIPIOCW_SHBURST` must be called for every packet to be sent with a short, first burst.

In HIPPI-FP mode, the short, first burst consists of the FP header and the D1 area as set by a previous call to `HIPIOCW_D1_SIZE`. When the FP header and D1 area occupy less than a burst, the B-bit is set, and the short, first burst is sent. Otherwise, the standard, first burst is sent. The D2 size must be a multiple of 1024 bytes in length or `D2SIZE_UNKNOWN`. The `byte_count` is ignored.

The byte_count must be a multiple of eight in the range 0 through 1024, inclusive. The first write must be long enough to send the short, first burst. In HIPPI-PH mode, the byte_count is the length of the short, first burst. A byte_count of zero (0) causes a standard, first burst (1024 byte) to be used and cancels the previous call to HIPIOCW_SHBURST.

A short, first burst is occasionally required by the destination hardware. The Sun NICs do not require the use of a short, first burst.

Usage

```
#include <ioctl.h>

int fd_hippi, ret;

int byte_count;


ret = ioctl(fd_hippi, HIPIOCW_SHBURST, byte_count);
```

Arguments

The following arguments are supported:

- fd_hippi – The file descriptor of a CDI transfer device that is open
- byte_count – The length of the short, first burst

Failures and Errors

The following list contains the failures and errors that can occur:

- EINVAL – fd_hippi is a controlling device.
- EINVAL – An invalid byte_count is specified.
- EINVAL – fd_hippi is not open for O_WRONLY or O_RDWR.
- ENODEV – The NIC is not running.

HIPIOCW_START_PKT Call

This ioctl() is used to specify the length of a packet that will be sent with multiple write() calls. If HIPIOCW_START_PKT is not called before the first write() call of a packet, the packet is formed from a single write() call.

The byte_count is interpreted differently in different situations. In HIPPI-FP mode, when separate-headers-and-data mode is used, the byte_count is the length of the D2 area. In HIPPI-FP mode, when combined-header-and-data mode is used, the byte_count is the length of the entire packet.

When D2SIZE_UNKNOWN is used, the packet length is not limited. HIPIOCW_END_PKT must be used to terminate a packet of unknown length.

Usage

```
#include <ioctl.h>

int fd_hippi, ret;

int byte_count;


ret = ioctl(fd_hippi, HIPIOCW_START_PKT, byte_count);
```

Arguments

The following arguments are supported:

- fd_hippi – The file descriptor of a CDI transfer device that is open
- byte_count – The length of the short, first burst

Failures and Errors

The following list contains the failures and errors that can occur:

- EINVAL – fd_hippi is a controlling device.
- EINVAL – An invalid byte_count is specified.
- EINVAL – fd_hippi is not open for O_WRONLY or O_RDWR.
- EINVAL – A multiple-write packet is already being processed.
- ENODEV – The NIC is not running.

6

Troubleshooting

Problems could arise in communicating when you are using HIPPI. This chapter outlines some techniques that are helpful in isolating the problems.

The CDI contains two utilities, `blast(1M)` and `sink(1M)`, that can be used to verify the ability to send and receive packets. The `blast(1M)` utility sends packets, and the `sink(1M)` utility receives packets (see the man pages for more details).

NIC Installation and Operation

The installation can be verified by placing the NIC in internal loopback mode and passing some HIPPI packets. Internal loopback mode uses the entire NIC except the optical interface module. Sent packets are electrically looped back to the receive electronics. You must have superuser access to run these utilities.

▼ To Test the Installation and Operation of the NIC

1. **Log into the host as superuser.**
2. **Turn HIPPI off.**

```
# /etc/opt/SUNWconn/hippi/bin/hippi off
```

This step turns off RunCode.

Note – You should be aware that the command in the example above is not the same as the `hippi start/stop` script that exists in the `/etc/initd` directory.

---

3. **Unplumb the HIPPI interface.**

```
# ifconfig hipip0 down
```

This step is needed only if you have configured the NIC for network use. If you are unsure if the interface is configured, you can use the `ifconfig hipip0` command to check the interface's current state.

4. **Place the NIC into the internal loopback mode.**

```
# hippi on loopback
```

5. **Set up the NIC to receive packets.**

```
# sink -n 10 &
```

The `sink(1M)` command sets up the NIC to receive ten default packets (4 kilobytes in length) in the background.

6. **Send the packets.**

```
# blast -n 10
```

The `blast(1M)` command sends the ten packets. The `blast(1M)` command and the background job for the `sink(1M)` command should complete successfully.

7. **Check the status of the jobs.**

```
# hippi status
```

The status argument should report that ten packets have been sent from the source and received by the destination. You should not encounter any errors.

If the above test fails, either the installation failed or the NIC is defective. In either case, contact your Sun Service representative for assistance.

Optical Modules and Cables

HIPPI packets are passed over an optical cable that directly connects two HIPPI interfaces. There are two types of optical interfaces: multi-mode (short wavelength) and single-mode (long wavelength). Multi-mode is the most common. It supports cable runs of a few hundred meters. Single-mode is less common. It supports cable runs of several kilometers or more. The optical modules at both ends of the cable must be the same type, and the optical cable must be the correct type for the optical modules.

The optical surfaces at the ends of the cables and in the optical modules must remain clean; otherwise, erratic behavior can result from surface contamination.

Optical Connections

An optical cable connects two HIPPI interfaces, which can be one of the following items:

- A HIPPI NIC and a HIPPI switch port
- Two HIPPI NICs
- Two switch ports

Alternatively, both connectors of a HIPPI interface can be directly connected with a loop-back cable. A loop-back cable has a single connector at each end (it is usually half of an optical cable as described above).

Each HIPPI interface has two connectors: a transmitter and a receiver. In all cases, a transmitter must be connected to a receiver. When two HIPPI interfaces, A and B, are connected, the transmitter of A is connected to the receiver of B, and the transmitter of B is connected to the receiver of A.

When correctly connected, the bulkhead *link-LED* of each HIPPI interface will be on. When HIPPI NICs are connected, the RunCode must be running for the LED to show the proper state. When two switch ports are connected together, the link LEDs on both must illuminate.

▼ To Turn On RunCode and Check the Status

1. **Log in to the host as superuser.**

2. **Turn on RunCode and set the NIC to communicate over the optical cable.**

```
# hippi on network switched
```

3. **Check the status of the NIC.**

```
# hippi status
```

This command returns the state of the RunCode (`on` or `off`) and the state of the link (`on` or `off`). If the link is unexpectedly off, reverse the connectors at one end of the cable, and retry the `hippi status` command.

Optical Loopback Test

The optics module of the NIC may be tested by using a loop-back cable. Use the loop-back cable to connect both connectors on the NIC. When the optical loop-back test passes, the NIC is installed and operating properly.

▼ To Set Up the Loop-Back Test

1. **Log in to the host as superuser.**
2. **Turn off RunCode.**

```
# hippi off
```

3. **Use `ifconfig(1M)` to unplumb the network if needed.**

```
# ifconfig hipip0 down
```

Use this command only if the NIC as been configured for network use.

4. **Configure the NIC to use the optics module.**

```
# hippi on network switched
```

5. **Set up the NIC to receive ten default packets (4 kilobytes) in the background.**

```
# sink -n 10 &
```

6. **Set up the NIC to send the ten packets.**

```
# blast -n 10
```

7. **Check the status of the `sink(1M)` and `blast(1M)` commands.**

```
# hippi status
```

The `blast(1M)` and the background job for `sink(1M)` should complete successfully before you perform the next step.

The `hippi status` command should report that ten packets have been sent to the source and that ten packets have been received by the destination. No errors should occur.

If these commands fail, the optics module on the NIC is defective. Contact your Sun Service representative if the NIC is defective.

Optical Loop-Back Through a Switch

Connect a NIC to a HIPPI switch port as described above. Determine the port number and logical address of the port (the switch administrator should provide the needed information). Assume that the physical port is `0xA` and that the port has a logical address of `0x223`. Verify that the HIPPI interfaces are properly connected as described above.

▼ To Set Up Optical Loop-Back Through a Switch

1. **Log in to the host as superuser.**

2. **Turn off RunCode.**

```
# hippi off
```

3. **Unplumb the network if needed.**

```
# ifconfig hipip0 down
```

Use this command only if the NIC as been configured for network use.

4. **Configure the NIC to use the optics module.**

```
# hippi on network switched
```

5. **Set up the NIC to receive twenty default packets (four kilobytes each) in the background.**

```
# sink -n 20 &
```

6. **Set up the NIC to send the ten packets, using source routing (with CampON).**

```
# blast -n 10 - I-Field
```

The packets are sent to the address of the NIC (packets are looped back through the switch to the NIC). The background `blast(1M)` job should complete successfully when the ten packets have been sent.

Note – The value of the I-Field depends on the configuration of the NIC and takes the form `I0x07000223`. Refer to Appendix B "HIPPI-SC Excerpts" for more information on how to determine the value of the I-Field.

---

7. **Set up the NIC to send ten packets, using the destination routing (logical address with CampON).**

```
# blast -n 10 - I-Field
```

The packets are sent to the address of the NIC (packets are looped back through the switch to the NIC). Both background jobs should complete successfully when the ten packets have been sent.

Note – The value of the I-Field depends on the configuration of the NIC and takes the form `I0x07000223`. Refer to Appendix B "HIPPI-SC Excerpts" for more information on how to determine the value of the I-Field.

8. **Check the status of the NIC:**

```
# hippi status
```

The `status(1M)` command should report that twenty packets have been sent by the source and that twenty packets have been received by the destination. No errors should occur during this procedure. If this procedure fails, the switch is improperly configured.

Optical Testing Between NICs

The NICs may be connected through a switch or directly connected. The following test ensures that each NIC can send packets to the other NIC. For various reasons it is possible for NIC A to send packets to NIC B while NIC B cannot send packets to NIC A.

For the following example, assume NIC A is in port 3 at logical address `0x143` and NIC B is in port 8 at logical address `0x846`. When using a switch, test both the source route and the logical addresses. With directly connected NICs, the switch I-Field is not needed and can be omitted.

Before you perform the following procedure, verify that the HIPPI interfaces are properly connected as described in "Optical Connections" on page 79.

▼ To Test the Optical Connection Between NICs

1. **On NIC A, perform the following substeps:**

   a. **Log in to the host as superuser.**

   b. **Turn off RunCode.**

```
# hippi off
```

c. **Unplumb the network if needed.**

```
# ifconfig hipip0 down
```

d. **Configure the NIC to use the optical module.**

```
# hippi on network switched
```

e. **Set up the NIC to receive twenty packets in the background.**

```
# sink -n 20 &
```

You should wait for the background jobs to complete successfully before you perform the next substep.

f. **Check the status of the packets.**

```
# hippi status
```

The `hippi status` command should report that twenty packets have been received by the destination.

2. **On NIC B, perform the following substeps:**

a. **Log in to the host as superuser.**

b. **Turn off RunCode.**

```
# hippi off
```

c. **Unplumb the NIC if needed.**

```
# ifconfig hipip0 down
```

d. **Configure the NIC to use the optical module.**

```
# hippi on network switched
```

e. **Set up the NIC to send ten packets, using source routing (with CampON).**

```
# blast -n 10 - I-Field
```

The packets are sent to the address of the NIC (packets are looped back through the switch to the NIC). The `blast(1M)` job should complete successfully when all ten packets have been sent.

Note – The value of the I-Field depends on the configuration of the NIC and takes the form `I0x07000223`. Refer to Appendix B "HIPPI-SC Excerpts" for more information on how to determine the value of the I-Field.

---

f. **Set up the NIC to send ten packets, using destination routing (logical address with CampON).**

```
# blast -n 10 - I-Field
```

The packets are sent to the address of the NIC (packets are looped back through the switch to the NIC). The `blast(1M)` job should complete successfully when all ten packets have been sent. You should wait for the background jobs to complete before you perform the following substep.

Note – The value of the I-Field depends on the configuration of the NIC and takes the form `I0x07000223`. Refer to Appendix B "HIPPI-SC Excerpts" for more information on how to determine the value of the I-Field.

---

g. **Check the status of the packets.**

```
# hippi status
```

The `hippi status` command should report that twenty packets have been sent by the source. No errors should occur during this procedure.

If the above procedure fails, the switch is improperly configured. If the above procedure fails with directly connected NICs, contact your Sun Service representative for assistance. If this procedure passes, the NICs and switch ports are properly configured.

Long Packets Between NICs

The default operation of the driver is to limit packet size to 64 kilobytes, as suggested by the network standards. When the CDI is used with applications that require larger packet sizes, the NIC can be configured to allow longer packets to be sent.

If you configure the NIC to permit longer packets, the NIC allows long packets to be sent as long as the host is running. When the host is rebooted, the operating mode reverts to parameters that are set in the EEPROM on the NIC.

▼ To Set Up the NIC for Long Packets

1. **Log in to the host as superuser.**

2. **Turn off RunCode.**

```
# hippi off
```

3. **Unplumb the NIC if needed.**

```
# ifconfig hipip0 down
```

4. **Configure the NIC to use long packets.**

```
# hippi on long
```

5. **Plumb the NIC if needed.**

```
# ifconfig hipip0 up
```

This procedure can be used when the host is running. When the host is rebooted, the operating mode reverts to parameters that are set in the EEPROM on the NIC. You can use the following procedure to change the EEPROM so that the NIC will automatically come up in the desired mode when the host is booted.

▼ To Change the EEPROM for Long Packets

1. **Log in to the host as superuser.**
2. **Turn off RunCode.**

```
# hippi off
```

3. **Configure the NIC to allow long packets.**

```
# hippi on long
```

4. **Turn off RunCode to access the EEPROM.**

```
# hippi off
```

5. **Write the current mode to the EEPROM.**

```
# hippitune -l -e
```

6. **Restart RunCode on the NIC.**

```
# hippi on
```

A

Special Files

The CDI supports multiple NICs in the host and multiple devices on each NIC. Special files that may be opened to access the CDI are in `/dev/hippi` on the host system. Each card has a controlling device and one or more transfer devices. The controlling device is used by utilities and the application to manage the NIC. The `read()` and `write()` calls are not supported and the controlling device can be concurrently opened by more than one process. Transfer devices are opened for exclusive access. They support the `read()` and `write()` calls.

One controlling-device special file and 31 transfer-device special files are created for each NIC that is installed in the system. The control-device special file has sub-device zero (0). The following list contains the naming conventions for special files:

- h*card*
- h*card.sub-device*

In these examples, *card* represents the card number (as determined during system bootup) and is represented as decimal digits. The *sub-device* is used on all transfer devices from 1 to 31, and it is also represented as decimal digits. The following list contains examples of the conventions:

- `h0` – Control device for NIC 0
- `h0.1` – CDI transfer device 1 for NIC 0
- `h0.2` – CDI transfer device 2 for NIC 0

The installation process creates the control device file and the 31 transfer-device files for each NIC. The `essioctl.h` header file defines NIC0 through NIC15.

B

HIPPI-SC Excerpts

The following information is excerpted from the HIPPI-SC specification; for more information refer to the specification. When the HIPPI-SC option is set, the host software must construct valid CCI fields (I-Fields) for proper send operation through a HIPPI switch (see FIGURE B-1). The detailed settings must be determined after analysis of the switch configuration.

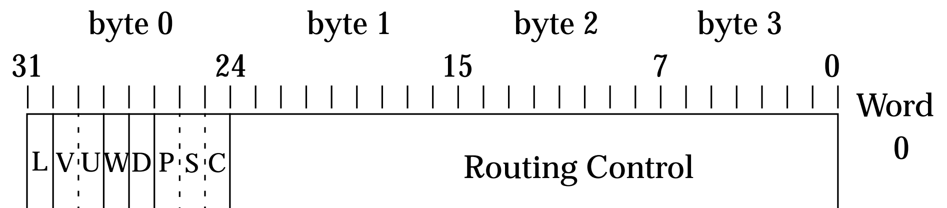


FIGURE B-1 CCI and I-Field Format

The HIPPI-SC CCI field describes the CCI field (I-Field) as it appears on the HIPPI media. In host memory and in the NIC, the CCI field is 8-bytes in length. This permits the packet to be 8-byte-aligned in an 8-byte or larger buffer (which is important for 64-bit CPUs). The first 4 bytes of the extended CCI contain zero (0). The CCI is in Big Endian on the media. Little Endian hosts must byte swap. The following list explains the next 4 bytes.

- `L` (Locally administered bit 31) – `L=0` designates that the I-Field is defined by the standard. `L=1` designates that rest of the bits (30 through 0) are locally defined and are not defined by the standard.
- `VU` (Vendor Unique bits 30 through 29) – The contents of these bits are not defined by the standard. Switches pass these bits through to the destination.
- `W` (Double Wide bit 28) – `W=0` designates that the source is using the 800-Mbit-per-second data-rate option. The NIC requires that this bit be set to zero (0).

- D (Direction bit 27) – D=0 designates that the right-hand end (least significant bits) of the routing control field shall be the current sub-field. D=1 designates that the left-hand end (most significant bits) of the routing control field shall be the current sub-field.
- PS (Path Selection (bits 26 and 25) – Used to select either (1) a source route (that is, a specific route through the switches) with output-port numbers specified for each switch, or (2) to specify the logical address.
  - PS=00 – Source routing (Source selects the route through the switches.)
  - PS=01 – Logical address (Switches select the first route from a list of possible routes.)
  - PS=10 – Reserved
  - PS=11 – Logical address (Switches select a route.)
- C (Camp-on bit 24) – C=0 specifies that the switch should reply with a connection-rejected sequence if it is unable to complete the connection. C=1 specifies that the switch should attempt to establish a connection until either the connection is completed or the source aborts the connection request.

Note – The NIC requires that the W-bit be zero (0).

The HIPPI-SC standard reserves certain logical addresses for specially defined uses. In general, logical addresses from 0xF90 through 0xFFF, inclusive, are reserved.

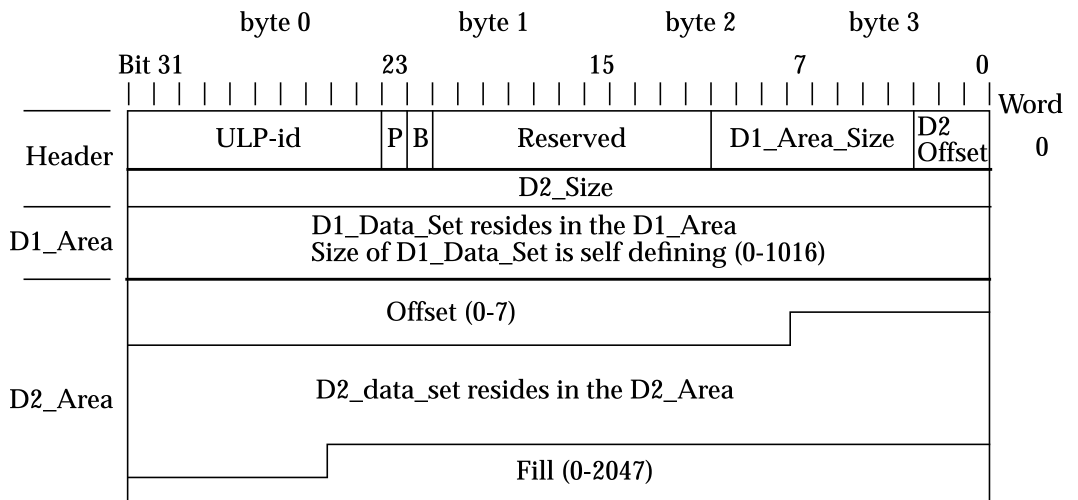
C

HIPPI-FP Excerpts

This appendix contains information that is excerpted from the HIPPI-FP specification (for more information refer to the specification). When the HIPPI-FP option is set, the host software must construct valid FP headers so that the packets are sent properly through the NIC. The NIC uses the fields in this header to direct processing. The detailed settings must be determined after analysis of processing requirements.

The header is always in Big Endian on the media. Little Endian systems must perform the needed byte swaps.

The packet data presented by the source ULP with an `FP_TRANSFER` request primitive shall be transferred to the destination with a HIPPI-FP header, as shown in the following figure.



HIPPI-FP Header

The HIPPI-FP packets are composed of three areas: (1) the header, (2) the D1_Area, and (3) the D2_Area. Each area starts and ends on a 64-bit boundary. If D1_Data_Set is used as control information, the D2_Data_Set is intended as the data associated with that control information.

Header

The header is the first 64 bits of the packet and should be completely contained in the first burst. The following list explains the subareas within the header:

- ULP-id (8-bits) – Designates the destination ULP to which the packet is to be delivered. ULPs in the range of 0 to 128 are reserved by the HIPPI standards organization. ULPs in the range of 128 through 255, inclusive, are available to the application. The following ULPs are defined:
  - 00000010b - Memory interface
  - 00000011b - Memory interface initialization
  - 00000100b - ISO 8802.2 link encapsulation
  - 00000110b - IPI-3 slave
  - 00000111b - IPI-3 master
  - 1xxxxxxxb - Locally assigned
- P (D1_Data_Set_Present=1) – Designates that a D1_Data_Set is present in this packet.
- B (Start_D2_on_Burst_Boundary=0) – Designates that the D2_Area starts at or before the beginning of the second burst. B=1 designates that the D2_Area starts at the beginning of the second burst.
- Reserved (11-bits) – Must be zero (0).
- D1_Area_Size (8-bits) – Designates the size of the D1_Area, the number of 64-bit double words that are between the header and the start of the D2_Area.
- D2_Offset (3-bits) – Designates the number of offset bytes from the start of the D2_Area to the first byte of the D2_Data_Set.
- D2_Size (32-bits) – Designates the length in bytes of the D2_Data_Set portion of the packet. The D2_Size does not include the bytes contained in the D2_Offset or the fill following the D2_Data_Set. A D2_Size of D2SIZE_UNKNOWN specifies that the D2_Data_set size is unknown. Packets with an unknown D2_DataSet size cannot be terminated at arbitrary byte boundaries. They can be terminated only at 64-bit, HIPPI-PH, burst boundaries. A D2_Size of zero specifies that the D2_Area and the D2_Data_Set do not exist.

D1 Area

The D1 area immediately follows the header and should be completely contained in the first burst. It should also contain an integral number of 64-bit double words and the `D1_Data_Set` (if one is present).

If present, the `D1_Data_Set` is the first information in the D1 area. If the `P`-bit of the header is zero (0), then the `D1_Data_Set` is not present. The `D1_Data_Set` is intended for control information that may be delivered to the destination ULP on receipt, without waiting for the arrival of other bursts of the packet.

The size of the `D1_Data_Set` is self defining; however, the maximum size is 1016 kilobytes.

D2 Area

If the D2-size field is not zero(0), the D2 area immediately follows the D1 area, and it starts and ends on 64-bit, double-word boundaries. The D2 area contains the `D2_Data_Set`. If the `B`-bit in the header is one, the D2 area starts at the beginning of the second HIPPI_PH burst.

The offset is the unused bytes from the start of the D2 area to the first byte of the `D2_Data_Set`. Fill is the unused bytes between the end `D2_Data_Set` and the end of the D2 area. When `D2_Size` is all binary ones, then there is no fill.

D

HIPPI-PH Excerpts

This appendix contains information excerpted from the HIPPI-PH specification (refer to the specification for more information). The following physical framing hierarchy describes the data flow over HIPPI media.

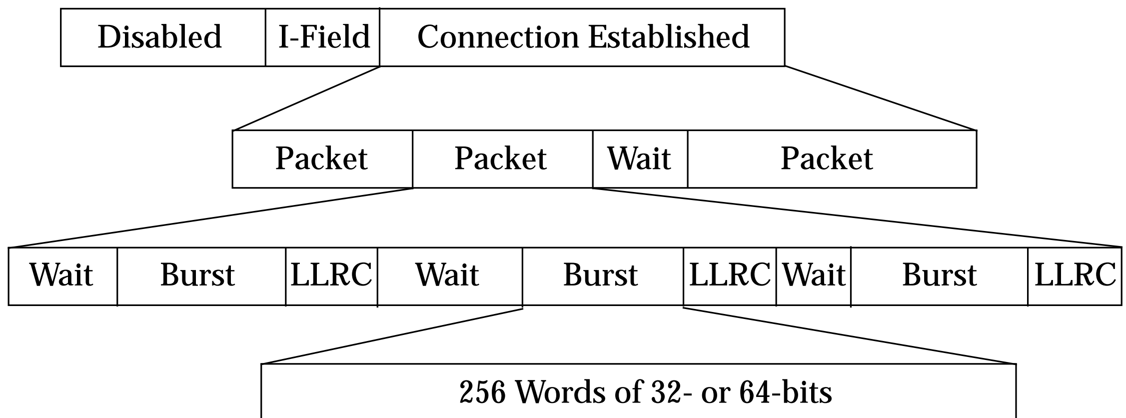


FIGURE D-1 Physical Framing Hierarchy

The NIC implements 32-bit words. Wait intervals are determined by the HIPPI flow control mechanism. The destination tells the source how many bursts it can receive by transmitting one ready pulse for each burst. When a connection is made the destination sends its first set of bursts (128 are in the first set that permits 128-Kbytes packets to be transferred at HIPPI media speed). When passing longer packets additional bursts are issued as the target system accepts packet data from the NIC. There is an error detection block, `LLRC`, after each burst. A packet can have, at most, one short burst. The short burst may be either the first or last burst.

E

File Locations and Examples

The files needed to write CDI applications are installed as part of the `SUNWhipc` package (see the *Sun HIPPI/P 1.0 Installation and User's Guide* for instructions on how to install the `SUNWhipc` package). By default, these files are located in the root directory `/opt`. The following files are included in the `SUNWhipc` package:

- `/opt/SUNWconn/include/sys/hippi.h`
- `/opt/SUNWconn/hippi/examples/blast.c`
- `/opt/SUNWconn/hippi/examples/sink.c`

The `hippi.h` file must be included in any CDI application. It contains definitions for the various HIPPI operations defined in this document. The `blast.c` file and the `sink.c` file contain functional, sample, CDI source code.

You can use the following commands to compile the `blast.c` and `sink.c` files can

```
% cc -o blast -I/opt/SUNWconn/include/sys blast.c
% cc -o sink -I/opt/SUNWconn/include/sys sink.c
```

The blast and sink options are fully described in the `blast(1M)` and `sink(1M)` man pages. Binaries for these two utilities are also installed as part of the `SUNWhip` package.

Glossary

Big Endian The most significant bit of data of a multiple-byte object is at the lowest, byte address.

burst A unit of transfer on a HIPPI media. A full-size burst is 1024 bytes.

CDI Character Device Interface

Connection The path from a HIPPI source to a HIPPI destination over which packets are passed.

Controlling Device A special file that is used to control a NIC.

Destination The HIPPI receiver that can accept packets

DMA Direct Memory Access. Data is moved directly between the NIC and the domain's memory without passing through a CPU.

Double Word An 8-byte object

Duplex A device that can both send and receive data

Endian The end of a multiple-byte object that has the low-order byte address (see Big Endian and Little Endian)

End point A HIPPI source or destination

Firmware Software that is running in the NIC

Full duplex A device that can simultaneously send and receive data

HIPPI High Performance Parallel Interface (800- or 1600-Mbits-per-second communication media)

Half word 2-byte object

Logical address A HIPPI-SC address of a HIPPI destination

Little Endian The least-significant bit of data of a multiple-byte object is at the lowest, byte address.

LLRC Error detection word at the end of each HIPPI burst

Master mode DMA DMA operations that are initiated and controlled by the NIC

NIC Network Interface Card

Road runner ASIC that is the heart of NIC, controlling processing and the domain interface

RunCode Firmware that is used during normal operation of the NIC

Simplex A one-way communication path on which a device can either send or receive

Single wide A HIPPI network operating at 800-Mbits per second

Source A HIPPI transmitter that can generate packets

Transfer device A special file opened by an application for reading and writing data

ULP Upper Layer Protocol identifier (HIPPI-FP)

Word 4-byte object