Essential Communications Corp.
Character Device Interface
User's Guide and Reference

Document Number 15-0167-00.04

Contributors
Written by Philip Cameron
Illustrated by Philip Cameron
Engineering contributions by Philip Cameron
Reviewers: Jay Bhat, Roger Frye, John Gibbon, Dave Gold, Masato Maeda, and Bob Pearson

Revision History

Rev 00.04 14Nov1997 Add multiple packet connection changes
Rev 00.03 02Apr1997 third review draft
Rev 00.02 26Nov1996 second review draft
Rev 00.01 20Jun1996 first review draft

Introduction

About this manual...

This manual is a combined User's Guide and Reference for the Essential Communications Corp. Character Device Interface. The User's guide describes how application programs access the HIPPI NIC (network interface card) using the Unix character device interface. The Reference provides detailed information about the CDI.

What's inside...

User's Guide
- Section 1 is an overview of the character device interface.
- Section 2 describes device management features
- Section 3 describes connection management features
- Section 4 describes receive processing
- Section 5 describes send processing
- Section 6 describes system select() support
- Section 7 discusses endian issues
- Section 8 discusses compatibility issues with Version 2.0 (xx-xxxx-xx.xx)
- Section 9 presents some application portability tips
- Section 10 mentions some useful utilities

Reference
- Section 1 Header file
- Section 2 discusses CDI specific features of the standard character device functions
- Section 3 details the CDI support ioctls
- Section 4 discusses special file naming conventions and minor device usage

The Glossary provides definitions of technical terms used throughout the manual.

References lists the relevant standards and RFCs supported by the NIC.

Standards Conformance

This interface conforms to accepted use of a Unix character device interface.

The interface conforms to the protocols that are described in the HIPPI standards for HIPPI-SC, HIPPI-PH, and HIPPI-LE.

Editorial Conventions

Terms that are proper names of interfaces and coding examples are printed in courier to avoid possible confusion.

A number of conditions, sequences, parameters, Events, states, or similar terms are printed with the first letter of each word in uppercase and the rest in lowercase (e.g., Source). Any lowercase uses of these words have the normal technical English meaning.

Utility names are in italics.

The Glossary provides definitions for commonly used terms.

Contents

Figures

User's Guide

Section 1 Overview

This CDI (Character Device Interface) permits 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 are used to 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.

The User's Guide assumes that you are familiar with HIPPI. The annexes, especially Annex A, provides additional information about the HIPPI protocols.

1.1 General Operation

The HIPPI NIC includes two independent HIPPI devices: a Source (transmitter) and a Destination (receiver). HIPPI establishes a connection between a Source and a Destination prior to passing data packets. When data transmission is complete the connection is broken.

The connection is routed through the HIPPI network by an I-Field that precedes the first packet that will be passed over the connection (the I-Field is not part of the packet). The Destination (receiver) discards the I-Field since it is not part of a packet and it is no longer needed.

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

A NIC that is operational will always accept a connection request and will accept packets even if there is no application or an unexpected application 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 there is no application 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. Since any Source can send to any Destination. It is up to the receiving application to verify the source of the data before it is processed by the application. The application must be able to discard unwanted data.

HIPPI has defined a low level transfer protocol, HIPPI-PH, which is frequently used as a dedicated link between systems. HIPPI-FP is layered on HIPPI-PH. It provides a packet transfer protocol that passes packets among users of the HIPPI network. Applications should consider using HIPPI-PH in a direct connect configuration where the resources can be dedicated to the connection. HIPPI-FP should be used in a general network configuration. The administrator uses the *esshippi* 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.

The CDI uses blocking IO (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. Transmit uses ioctl()s to manage headers and control the transfer and `write()`s or `writev()`s to send the data. Receive uses `ioctl()`s to control the transfer and `read()`s or `readv()`s 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 ULP (upper layer protocol). 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). Since 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 the `read()`s that are queued 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()` or `writev()` return –1 and errno is set to EIO. An `ioctl()` is provided to get 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 (e.g., NIC0, NIC1, NIC2...).

The *esshippi* utility controls the operating mode of the NIC (e.g., FP vs. PH). It also starts and stops the NIC and provides operating statistics. The CDI provides an `ioctl()` to get the current mode settings.

Since 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 there are timeouts on both the Receive and Transmit channels. Once a packet starts to transfer it must continue to transfer. If no data is transferred for the configured timeout period, the packet is truncated and the connection is closed. The timeout may be configured by *esstune*.

1.2 Unexpected Packets

When an application starts it (usually) opens a HIPPI Transfer Device and binds to a ULP that it wishes to use for receiving packets. It then sends and receives packets. The method of determining the destination NIC address and ULP that the application should use is not defined in this 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.

There is no way of establishing a session (as is typical when using TCP/IP). Connections are (usually) made and broken for each packet instead of being made for the duration of the application execution, an application may receive packets that it is not expecting.

The application must validate the Source of every received packet before it is processed. It must make sure 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.

1.3 Undelivered Packets

The Destination NIC will (usually) accept a packet whether or not there is an application ready to receive it. When the application is not running (and the ULP is not bound), the packet is discarded by the NIC. The Source is not notified of this error.

1.4 NIC State

The NIC is a programmable device that executes a firmware program called RunCode. The *esshippi* utility is used to manage the NIC including loading RunCode into NIC local memory (SRAM), starting the RunCode and setting the operating state. See the *esshippi* manual page for more information.

The operating states include:

on	- NIC is executing RunCode
off	- NIC is stopped
long	- CDI can send packets that are longer than 64k
short	- CDI is limited to sending 64k packets
fp	- CDI is in HIPPI-FP mode
ph	- CDI is in HIPPI-PH mode

In fp mode the Network Driver can concurrently operate with the CDI. When the Network Driver is configured it is good to limit the maximum packet size to the network MTU size (64k) by setting "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 and results in lengthy delays in sending network packets. Since the NIC can receive a single packet at a time, receiving long packets delays the arrival of network packets. When "short" is set, the CDI prevents long packets from being sent. 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.

1.5 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, that are used by applications to access the NIC for data transfer (`read()`s and `write()`s).

1.6 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 problem 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, `open()`, `close()`, `ioctl()`, `read()`, and `write()` on a Transfer Device while the NIC is off, ENODEV is returned.

Since `read()` and `write()` are not supported on the Control Device, EINVAL is returned.

Parameter errors return EINVAL.

Invalid buffer pointers return EFAULT.

Insufficient permission returns EPERM.

The response to `read()` IO 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.

1.7 Byte Order

The HIPPI network is Big Endian. In multi-byte objects, the lowest byte address contains the most significant byte of data.

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

1.8 Data Buffers

The CDI restricts buffers to be aligned on 8 byte boundaries. When multiple `write()`s or `read()`s 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 long. The CDI does not restrict packet length. Host systems may further restrict length or alignment.

1.9 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.

1.10 Data Movement Timeouts

The NIC uses separate receive and transmit data movement timeouts to monitor data movement. Once a connection is opened and packet data starts to flow, the timeout is active. If no data is passed for the timeout period, the packet is truncated and the connection is dropped. An error is returned. The timeout value is set by the administrator using the *esstune* utility.

1.11 ULPs

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 128-255 (decimal) are available for general application use.

ULPs in the range 0-127 (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 assigned for specific uses. The specific ULPs that are reserved by HIPPI-FP and must not be used by the application are:

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

The CDI does not permit access to the above defined ULPs.

Section 2
Device Management

This section describes the process that an application uses to access the HIPPI NIC. Annex-C describes the HIPPI-FP header.

2.1 Device Open

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()` before `fork()` results 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). Since `read()` and `write()` processing are independent, each is discussed in turn. An application can have more than one Transfer Device open at a time.

The application may call HIPIOC_GET_NICS `ioctl()` to determine the number of configured NICs.

2.2 Binding a Read ULP

The NIC (while in HIPPI-FP mode) 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, there is no ULP. However, the application must still bind to establish shared or exclusive access.

Multiple Transfer Devices may be bound to the same ULP. When this occurs the read()s 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 to another ULP.

A Transfer Device that is open for write-only does not bind to a ULP.

It is up to the sending and receiving applications to negotiate a suitable ULP. This 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.

2.3 HIPPI-FP and HIPPI-PH Modes

The NIC operating mode is set by the *esshippi* 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 and then `read()`s the transfer device file. 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).

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.

2.4 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 `ioctl()`s to process headers and `read()`s or `write()`s 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 and are prepended 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 `read()`.

Receive processing and send processing differ in the following ways:
- Send includes an I-Field before each packet, receive does not include the I-Field.
- Send permits passing a short first burst, receive doesn't notice burst length.

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

2.5 Device Close

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. Future packets to the ULP are discarded.

On close a packet that is partially transmitted is truncated. The connection, including 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.

2.6 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, "*esshippi status*" or "*essstat -x*" may be invoked by the user to display NIC statistics.

2.7 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 using `ioctl()`s. Since a multiple packet connection ties up the two NICs and paths through the switches it is good to either dedicate resources or limit its 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. Once the connection is open, it remains open until:

- it is closed by a `HIPIOCW_DISCONN ioctl()`
- an error (including Tx Idle) occurs
- the transfer file is closed
- the NIC stops

Once the connection is open 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 when between packets. It fails if the application is between `write()`s of a multiple `write()` packet.

Section 3
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 and destination NICs. It is not a connection as is established by 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.

3.1 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 HIPPI device using information in the I-Field to control processing (see Annex-B).

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 HIPPI device may be:

- Available to accept the connection,
- Already connected to another Source, or
- 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. Since 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 *esstune* utility is used to set this timeout.

When the Destination is busy and CAMP_ON is not set, the connection is immediately rejected. *esstune* may 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.

3.2 Specifying Destination Device

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. (See Annex-B for more details).

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 (see Annex B).

3.3 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.

There is no standard method of finding a Destination ULP using HIPPI packets. It is up to the application to determine the Destination.

All packets that are sent to a destination are accepted. There is no error indication when the ULP is not valid or in use by another application.

3.4 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 HIPIOCW_DISCONN. The data movement timeouts are in effect.

Any error closes the connection.

Section 4
Processing Received Packets

The received packets may be 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 (D2 area) and `ioctl()`s can read headers and track the progress of packet processing.

A single `read()` will never return more than one packet.

A single packet may be read using one or more `read()` calls. When multiple `read()`s are used all buffers must be 8 byte aligned and all but the last buffer must be a multiple of 8 bytes long. 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 applications choose.

The application must completely process each packet that arrives (both expected and unexpected packets). `ioctl()`s are provided to track the progress of receiving packets and to discard unwanted packets.

This section describes several different approaches to receiving packets.

4.1 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 using the HIPIOCR_GET_FP `ioctl()`. HIPIOCR_GET_FP is used before the `read()` so it will block until the packet arrives instead of the `read()` blocking.

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 indicates the size of the D2 area (if any). Using the D1 and D2 sizes the application can determine the overall length of the packet. When the D2 size contains 0xFFFFFFFF the length is not known. See section 4.3 for processing details.

When the D1 size is not 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()`s may be used to read the D2 data area. Headers must be processed before the D2 area.

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

4.2 HIPPI-FP Combined Headers and Data

When (the default) combined header mode is set by the HIPIOCR_SEP_HDR `ioctl()`, a series of `read()`s 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 indicates the size of the D2 area (if any). Using the D1 and D2 sizes the application can determine the overall length of the packet. When the D2 size contains 0xFFFFFFFF the length is not known. See section 4.3 for processing details.

The `read()` blocks until data is available to be read.

4.3 Unknown Size Packets

In HIPPI-FP mode when the D2 size of a packet is not known when the packet is sent, D2 is set to D2SIZE_UNKNOWN (-1 or 0xFFFFFFFF). The application is expected to read however much data is in the packet or 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 long.

In HIPPI-PH mode there are no headers so all packets are of unknown length.

When the NIC is in PH mode, the packet must be a multiple of 4 bytes long.

When a short first burst is used, the remainder of the packet must be a multiple of 1024 bytes.

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*4096 bytes or 0x7FFFFFFF bytes.

When the whole packet has been read, the counter is reset to 0 and the ioctl returns 0 byte offset.

4.4 HIPPI-PH Packet Read

HIPPI-PH mode does not use headers. The header related `ioctl()`s are not valid and return 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 determine packet boundaries (see section 4.3 above).

4.5 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()` may be used to discard the remainder of the packet. This useful when processing packets with multiple `read()`s.

4.6 Packet Read Errors

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

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()` may be used to get a more detailed analysis of the error.

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

4.7 Process Interrupt (^C)

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

4.8 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 when it takes more than one read 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.

Section 5
Send Processing

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 (see Annex B). 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 an (optional) D1 header, followed by (optional) D2 data (see Annex C). The FP header and D1 can be sent as a short first burst.

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

When an application communicates with another application, it is common for the I-Field, FP header and D1 to not change much from packet to packet. When this is the case the application can cache header information in the CDI and the CDI will format packets using cached headers.

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

5.1 Setting the I-Field

Every connection is established using a HIPPI-SC I-Field. The I-Field is set using the HIPIOCW_I or HIPIOCW_CONNECT `ioctl()`s as described in section 3. The I-Field is cached in the CDI. Once 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 and 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.

5.2 Managing the FP Header

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 using HIPIOCW_SEP_HDR `ioctl()`. When the FP header is in the application buffer it must have the correct length fields and it must be in Big Endian order (see Annex C for the FP header format). When the FP header is cached by the the CDI, the application uses a set of `ioctl()`s 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 header and data mode clears the cached FP header.

HIPIOCW_SET_ULP sets the Destination ULP. Observe the rules for reserved ULPs described in section 1.10 above.

HIPIOCW_SHBURST forces the first burst to be short (see section 5.3). All remaining bursts must be a full 1024 bytes.

HIPIOCW_D1_SIZE sets the size of the D1 area. The D1 area is a multiple of 8 bytes long up to

1016 bytes. When there is no D1 data present, the D1 length is set to 0. The application manages the content of the D1 space. When the D1 size is 0 there is no D1 header.

HIPIOCW_D1_AREA_PTR tells the CDI the location of the D1 buffer in the application. It is used when the application manages the D1 area in a private buffer. When this option is used the first `write()` 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 user D1 buffer is no longer used and the D1 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 cached D1 data whenever it likes. 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 is part of the application buffer.

The application has the flexibility to:

- not use a D1 header,
- place the D1 area before the data in the data buffer,
- place the D1 area in a separate buffer, or
- cache the data in the kernel.

Cached data is used in multiple packets until it is changed. When neither HIPIOCW_D1_AREA or HIPIOCW_D1_AREA_PTR are not used, the write buffer contains the D1 area.

There is no support for setting the D2 offset field.

The D2 size field is set indirectly by one of the following:

Call `write()` without calling the HIPIOCW_START_PKT `ioctl()` creates a packet with a single `write()`. When the D1 is in the user buffer, the write length 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 are included in the `write()`. In this case the D2 size is the write length less the headers.

Use HIPIOCW_START_PKT to start a new packet of the specified D2 length. Multiple `write()`s 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 must be at least as long as the D1 area. When in separate header mode the first write must be at least the FP and D1 headers. Extra `write()` bytes 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 `write()`. None of the header manipulation `ioctl()`s are available. The application must ensure that the FP header is in Big Endian and the ULP is valid. The I-Field is always set by an `ioctl()`.

5.3 HIPPI-PH Mode

In HIPPI-PH mode there are no headers. There are two approaches to writing packets:

Use the HIPIOCW_START_PKT `ioctl()` to specify the length of the packet followed by one or more writes to send the packet, or

Write the packet using one or more writes and terminate the packet by calling the HIPIOCW_END_PKT `ioctl()`.

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

5.4 Unknown Size Packets

In HIPPI-FP mode the sender may start to send a packet before determining its final length. The FP header D2 field is set to D2SIZE_UNKNOWN (-1 or 0xFFFFFFFF) by the HIPIOCW_START_PKT `ioctl()`. The other FP header manipulation ioctl()s still operate. The first write 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 long.

When the NIC is in PH mode, the packet must be a multiple of 4 bytes long.

When a short first burst is used, the first write writes the short first burst. The remainder of the packet must be a multiple of 1024 bytes.

5.5 Short Burst

HIPPI permits up to one short burst in a packet. The short burst may be either the first burst or the last burst. The short burst is usually at the end of the packet. However, the application may 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 then pass the remainder of the data. Unless there is a hardware assist or some specific reason to use short burst don't use it. Essential NICs perform the needed header analysis without requiring a short first burst.

Also, don't use short first burst when the first burst is a full size burst (e.g., D1 is 1016 bytes).

The short burst may be used in both HIPPI-PH and HIPPI-FP modes. In HIPPI-FP mode, the burst size is not used, since 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 and less than or equal to 1016.

A standard HIPPI burst is 1024 bytes. When short first burst is used, the D2 area must be a multiple of 1024 bytes.

5.6 Transmit Queue

The CDI has a single transmit queue. Packets are placed at the tail of the queue by applications. The NIC takes packets from the head of the queue and sends them.

Since the queue is shared by all applications, access is restricted to one application at a time. The first (or only) `write()` for a packet waits for the queue 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 using multiple `write()`s and multiple packets over the connection.

The application blocks from the time that the `write()` is called until the NIC sends the data and the `write()` completes. When sending packets using multiple `write()`s, the application should do the `write()`s as quickly as possible. The Transmit Idle Timeout limits that delay between `write()`s. If no data is transmitted for the timeout period the connection is broken and the packet is discarded. The transmit idle timeout may be set by *esstune.*

If the application does not want to wait for the transmit queue to become available, `select()` may be used (see section 6).

5.7 `write() & writev()`

`write()` and `writev()` may be used to send packets. The packet processing details are set before calling `write()`. 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 the last buffer must be a multiple of 8 bytes long. The last write can be any number of bytes (when sending a known length packet).

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

In HIPPI-FP mode, the packet must be the length that is specified in the D2 size field of 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 `write()` (return value from `write()` is not the same as the length).

One or more `write()`s may be used to send a single packet. When multiple `write()`s are used, the packet length is set by a 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 `write()` returns the actual number of bytes that were sent. Multiple packets may not be sent in a single `write()`.

One or more packets may be sent 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 at the start of the next packet). When HIPIOCW_CONNECT is used, HIPIOCW_DISCONN is used to close the connection when the last packet ends. If the packet has already ended the connection is closed immediately.

When multiple `write()`s are used or unknown length packets are sent, the length of each `write()` must follow these rules:

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

5.8 Process Interrupt (^C)

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 the connection is closed. No further packets may be sent (since the Transfer Device is closed).

5.9 Transmit Errors

Errors in setting up the `write()`, (e.g., bad buffer pointer, invalid parameter, etc) are returned immediately. Write 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*), the packet is truncated and a disconnect is performed. errno is set to EIO. The write cannot be restarted.

Section 6
Select

The `select()` system call may be used to monitor the CDI for available packets and/or busy send queue.

All `read()`s and `write()`s to this CDI block while being processed. (The CDI implements blocking IO). The application may use `select()` to avoid calling `read()` until there is data to be received. It may also use `select()` to avoid blocking while another process is sending.

The call to `select()` is described in the host system man page. The CDI does not change `select()` processing.

6.1 Select Read Devices

The `select()` for `read()` completes when data is available to be read. When multiple read()s are used to process a packet, `select()` should be used only to wait for the arrival of the packet. It should not be called between successive `read()`s 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 select for the same bound ULP, the process that has been waiting longest will complete the select. The other processes will continue to wait. This eliminates races for the receive queue.

6.2 Select Write Devices

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

When the application uses multiple `write()`s per-packet or multiple packets per-connection, `select()` should only be called 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.

6.3 Select Exception Devices

There is no support for exception processing using `select()`.

Section 7
Endian

Host systems use one of two approaches for addressing multiple byte data objects (e.g., ints, shorts, longs). The most significant byte of the data object may be at the lowest byte address (Big Endian) or it may be at the highest byte address (Little Endian).

Little Endian hosts must make sure that network objects (e.g., I-Field and FP header) are in Big Endian order.

The HIPPI network, HIPPI-SC switches and HIPPI-FP headers are all Big Endian.

7.1 I-Field

The I-Field is always Big Endian. Little Endian machines must be careful to convert the I-Field to Big Endian before calling the HIPIOCW_I or HIPIOCW_CONNECT `ioctl()`s.

7.2 HIPPI-FP Header

The FP header is always Big Endian. When using the `ioctl()`s to construct the FP header, endian conversion is automatically performed. When the application manages the FP header in the user buffer, it must be careful to convert the header to Big Endian before calling `write()`. When a Little Endian host receives a packet in combined header mode the FP header will usually need to be converted to little endian before being processed by the application.

7.3 HIPPI-FP D1 and D2 Areas

These areas are managed by the application. They can be in either Big Endian or Little Endian. When the application interoperates with the same endian machine, no conversions are needed. When the application interoperates with an opposite endian machine, these areas must be converted by the application based on application needs.

Section 8
Compatibility with 2.0

The CDI is an evolution of the Essential 2.0 API. The differences are noted here.

8.1 Binding of ULPs

The 2.0 revision binds both the receive and send ULP with a single call to the HIPIOC_BIND_ULP `ioctl()`. Once bound the ULP can not be unbound. The CDI binds the receive ULP only. The send ULP is assigned as long as the HIPIOCW_SET_ULP `ioctl()` has not been called.

Applications that open a Transfer Device for O_WRONLY should not call HIPIOC_BIND_ULP. They should use HIPIOCW_SET_ULP `ioctl()` instead.

The CDI permits receive ULPs to be unbound. It permits receive ULPs to be bound for exclusive or shared use.

The CDI also permits the send ULP to be redefined as needed by calling the HIPIOCW_SET_ULP `ioctl()`.

8.2 `HIPIOC_SET_ONOFF`

This is an administrative `ioctl()`. Even though it is still available in the CDI. It is not documented and should not be used by the application. The `ioctl()` requires super user privilege.

8.3 `ESS_ALLOW_LARGE_TRANSFERS`

This is an administrative `ioctl()`. Even though it is still available in the CDI. It is not documented and should not be used by the application. The `ioctl()` requires super user privilege.

8.4 `ESS_DISALLOW_LARGE_TRANSFERS`

This is an administrative `ioctl()`. Even though it is still available in the CDI. It is not documented and should not be used by the application. The `ioctl()` requires super user privilege.

8.5 `HIPIOC_ACCEPT_FLAG`

This is an administrative `ioctl()`. Even though it is still available in the CDI. It is not documented and should not be used by the application. The `ioctl()` requires super user privilege.

8.6 `HIPIOCR_ONE_READ`

This is `ioctl()` is obsolete. The CDI default behavior is the same as executing this `ioctl()`. The `ioctl()` exists in 3.0 and returns 0. Applications should no longer call it.

8.7 HIPIOC_STIMEO

This is an administrative `ioctl()`. It is not properly implemented in 2.0. It should not be used by the application. The `ioctl()` requires super user privilege, returns 0 and does nothing. This has been replaced by tuning parameters that may be set by *esstune*.

8.8 HIPIOC_GET_STATS

This `ioctl()` is used to obtain statistics about the device. It is an administrative `ioctl()` and is not documented for application use.

8.9 HIPIOCW_CONNECT

In rev 2.0 the `ioctl()` returns EINVAL when a connect is already requested. In the CDI EBUSY is returned.

8.10 HIPIOCW_DISCONN

In rev 2.0 and the CDI the `ioctl()` immediately disconnects.

Section 9
Application Portability

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

9.1 Maximum read() and write() length

Host systems place restrictions on the maximum length of a single `read()` or `write()`. It is common for systems to pass as much as 2MB (2*1024*1024) byte in `read()` and `write()` calls. Since multiple `read()`s and `write()`s may be used for a single packet, this doesn't restrict packet size.

9.2 Buffer Alignment

For some hosts it is efficient to align buffers to cache line boundaries (usually 32 bytes)

For large transfers, it is efficient to align buffers to page boundaries. Pages are usually 4096, 8192, or 16384 bytes.

9.3 Endian

The D1 and D2 areas in HIPPI-FP packets and the entire HIPPI-PH packet may be in either Big Endian or Little Endian.

When both Source machine and Destination machine are the same endian, there is no problem. When they are of opposite endian, byte swapping may be needed. The applications need to coordinate the packet contents and agree to the needed conversions.

9.4 Maximum Packet Length

Some systems limit the maximum packet length to 4096*4096-8 (0xFFFFFFF8) bytes.

Section 10
Utilities

The following utilities are not really part of the CDI. However they are useful in managing the NIC interface. The text is from the man page for the utility.

10.1 esshippi

This utility permits the superuser to set the operating mode of the NIC. See the *esshippi*() manual page for more details.

10.2 esstune

This utility permits the superuser to set the tuning parameters for the NIC. See the *esstune*() manual page for more details.

Character Device Interface Reference

The Reference provides detailed information about the various features of the CDI. It is divided into several sections:

Section 1 - Header file

Section 2 - Interface Functions

Section 3 - Ioctls

Section 4 - 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 may be found in /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.

Section 1
Header File, Sample Files

The essioctl.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* and *sink* are distributed as source. These utilities provide examples of how the CDI can be used.

Digital Unix - Digital Equipment Corp.

/usr/include/io/hippi/essioctl.h
/usr/opt/HIPxxx/usr/examples/blast.c
/usr/opt/HIPxxx/usr/examples/sink.c
HIPxxx is the current kit number (e.g., HIP106)

HP-UX 10.xx - Hewlett Packard

/usr/include.....???

IRIX 6.2,6.3 - Silicon Graphics

/usr/include/essioctl.h

Section 2
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.

2.1 HIP_APP_OPEN()

This 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 all Transfer Devices have been tried. The macro calls `open()` and HIPIOC_GET_DEV `ioctl()`.

Usage:

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

int fd_hippi, flags;

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

The arguments:

`NIC0`	is the name of the Control Device special file for the NIC. NIC0 indicates the first NIC. essioctl.h defines the first 16 NICs (NIC0..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`	is 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.

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

2.2 close()

close() is used to close an open CDI device. See the close(2) manual page for more details on the general operation of close().

When close() is called any outstanding read()s or write()s 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 connection) is closed. Any packets for the special file that are in the transmit queue are returned with EIO.

If this is the last UNBIND for a receive ULP, the ULP becomes invalid and future packets to the ULP are discarded.

Usage:

```
close(fd_hippi)
```

The argument:

fd_hippi is the file descriptor of an open CDI special file

Failures and errors:

No special failures.

2.3 `ioctl()`

The `ioctl()`s that control the operation of the CDI are described in the section 3.

2.4 `open()`

The `open()` call is used to open a Control Device or open a Transfer Device for reading or writing (see section 4).

`open()` returns a file descriptor that is used for future calls to `read()`, `write()`, `ioctl()`, and `close()`. See the manual page for open(2) for more details.

`open()` of the Control Device (sub-device 0) for a NIC always succeeds as long as the Control Device name is valid and accessible. `open()` of Transfer Devices for the NIC fail if the NIC is stopped or the file is already open.

`open()` 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);
```

The arguments:

`fd_hippi`	is the file descriptor of an open CDI special file
`flags`	may contain O_RDONLY, O_WRONLY, or O_RDWR. All other flags (in particular O_NDELAY) are ignored.

Failures and errors:

ENODEV	- NIC is stopped
EBUSY	- device is already open
ENOMEM	- system ran out of memory in initializing device
EINVAL	- invalid minor number was specified.

2.5 `read() & readv()`

`read()` or `readv()` is used to read data from the HIPPI device. See the read(2) manual page for more details on `read()` and `readv()`.

The supplied buffers must be 8 byte aligned. The last `read()` of a packet may be any length, all other `read()`s must be a multiple of 8 bytes.

The application must validate the received packet since any Source can send to any Destination.

The application supplied buffer may be any length and start at any byte offset. There is 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 byte) for large packets. Some hosts run faster using buffers that are a multiple of cache line long.

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

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

In separate header and data mode, `read()` 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 there is no `read()` pending, the NIC flow controls 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 packet.

If an EIO error occurs, the HIPIOCR_ERRS `ioctl()` may 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);
```

The argument:

`fd_hippi`	is the file descriptor of an open CDI special file
`buf`	is the user space buffer that will receive the data
`BUF_SIZE`	is the size of the buffer

`len` is the actual number of bytes that were read or -1 if an error occurred.

Failures and errors:

ENODEV	- NIC is stopped
EINVAL	- read of sub-device 0
EINVAL	- invalid minor number was specified
EINVAL	- device not open for O_RDONLY or O_RDWR
EINVAL	- HIPPI-PH mode and length not multiple of 8 and > 0
EIO	- Data reception error occurred (use HIPIOCR_ERRS for more details)
EIO	- process kill (^C)

2.6 `select()`

`select()` allows the application to determine if data is waiting to be read or if the transmit queue is empty for `write()`. See the select(2) manual page for more details.

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

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

The `FD_SET()` macro (in sys/select.h) may be used to set the file descriptor bits in readfds and writefds.

HIPPI file descriptors can be intermixed with other file descriptors (e.g., files, sockets, ...)

The details of calling `select()` are host system specific. 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);
```

The arguments:

`width`	is the maximum number of descriptors to check (highest file descriptor +1)
`readfds`	is the FD_SET that refers to all the file descriptors to check for data available
`writefds`	is the FD_SET that refers to all the file descriptors to check for transmit queue empty
`timeout`	is the timeout value for the select(). If conditions do not change during this time, select() returns 0
`fds`	is the number of file descriptors that are ready to be processed.

Failures and errors:

No special error cases

2.7 write() & writev()

write() and writev() are used to write data to the HIPPI device. See the write(2) manual page for more details on general operation. write() accepts a single buffer and writev() accepts a vector of buffers.

The supplied buffers must be 8 byte aligned. The last write() of a packet may be any length, all other write()s must be a multiple of 8 bytes.

There is 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 byte) for large packets. There is also a minor performance advantage in making buffers a multiple of cache line size.

write() and writev() generally write the data immediately to the device and then return. However, there are cases where internal states, network flow control, and long transfers from other applications can cause write() to block. write() does not return until the data is written or an error has occurred.

write() operates based on the ioctl()s that have been used to condition packet transmission.

The packet length is the length of the write() unless the HIPIOCW_START_PKT ioctl() is used to specify a length. If the application attempts to send too much data, write() 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 for the packet must be a multiple of 8 bytes.

When short first burst is used, the application must write a multiple of 1024 bytes bursts for the remainder of the packet. This is because a packet can have only one short burst.

In HIPPI-PH mode, the write length must be greater than 0 and a multiple of 4 bytes.

Fill bytes at the end of the D2 area are undefined (not 0).

The Transmit Idle Timeout is active from the start of the first write() 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, HIPIOCW_ERR may be used to get more detail.

Usage:

```
#include <unistd.h>
int fd_hippi, len;
char buf[BUF_SIZE];

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

The argument:

`fd_hippi`	is the file descriptor of an open CDI special file
`buf`	is the user space buffer that will receive the data
`BUF_SIZE`	is the size of the buffer in bytes
`len`	is the actual number of bytes that were written or -1 if an error occurred.

Failures and errors:

ENODEV	- NIC is stopped
EINVAL	- write to sub-device 0
EINVAL	- invalid minor number was specified
EINVAL	- device not open for O_WRONLY or O_RDWR
EINVAL	- HIPPI-PH mode and length not multiple of 4 and > 0
EIO	- Data reception error occurred (use HIPIOCW_ERR for more details)
EIO	- process kill (^C)

Section 3
Ioctls

These `ioctl()`s are used to control the CDI.

3.1 HIPIOC_BIND_ULP

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. There may be multiple devices bound to the same ULP. The device file must be open for O_RDONLY or O_RDWR. 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 more than one Transfer Device. A positive ULP_id requests an exclusive bind; 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. A Transfer Device can accept packets to a single ULP.

ULPs in the range 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 0-127 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, 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.

When HIPIOCW_SET_ULP has not been executed, the ULP is also used as the transmit ULP.

A maximum of 31 different ULPs may be concurrently bound in FP mode.

Usage:

```
#include <essioctl.h>
int fd_hippi, ret, ULP_id;

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

The argument:

`fd_hippi`	is the file descriptor of an open CDI Transfer Device.
`ULP-id`	is the ulp that is being bound. When ULP-id is >= 0 the bind is exclusive. No other devices can be concurrently bound. When ULP < 0 the bind is not exclusive, other devices may be bound.

Failures and errors:

EINVAL	- ULP > 255
EPERM	- ULP < 128 and not superuser

EPERM	- ULP is reserved and defined (2, 3, 4, 6, 7, 8, 10)
EINVAL	- ULP is already bound to this device
EBUSY	- ULP is already bound to another device and exclusive bind is requested.
ENODEV	- NIC not running
EINVAL	- Device not open for O_RDONLY or O_RDWR
EINVAL	- fd_hippi is a Control Device (sub-device 0)
EINVAL	- NIC is in HIPPI-PH mode and ulp is not HIPPI_ULP_PH, HIPPI_ULP_PH_EXCL or HIPPI_ULP_PH_SHR
EINVAL	- Maximum number of ulps are already bound.

3.2 `HIPIOC_GET_DEV`

This `ioctl()` is not called directly by the application. It is called by the `HIP_APP_OPEN()` macro.

The `ioctl()` returns an unused Transfer Device for the NIC that can be opened by the application. The Transfer Device may be opened by another application by the time that `open()` can be tried so the caller must be prepared to retry.

Usage:

```
#include <essioctl.h>
int fd_hippi, ret, device;

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

The argument:

`fd_hippi`	is the file descriptor of an open CDI Transfer Device.
`device`	contains the returned device number to try to open.

Failures and errors:

EINVAL	- fd_hippi is not an open Controlling Device
EBUSY	- No devices are free.
ENODEV	- NIC not running

3.3 HIPIOC_GET_DEVICE_STATE

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

Usage:

```
#include <essioctl.h>
int fd_hippi, ret, status;

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

The arguments:

`fd_hippi` is the file descriptor of an open CDI Transfer Device.
`status` contains the operating mode of the device.

HIP_ON	0x01	NIC is running (or stopped)
HIP_LONG	0x02	long packets (or 64k short packets)
HIP_PH	0x04	HIPPI-PH mode (or HIPPI-FP mode)
HIP_LOOPBACK	0x08	Internal loopback mode (or network mode)
HIP_DIRECT	0x10	Direct connect mode (or switched mode)

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.

ENODEV - NIC not running (and this is not the Controlling Device)

3.4 HIPIOC_GET_NICS

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);
```

The arguments:

`fd_hippi`	is the file descriptor of any open CDI Control Device or CDI Transfer Device.
`nics`	contains the number of NICs in the configuration.

Failures and errors:

No `ioctl()` specific errors.

3.5 HIPIOC_UNBIND_ULP

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. (Between reads of a multiple read packet.)

When "esshippi off" is executed all ULPs are unbound. When `close()` is executed the ULP is unbound.

Usage:

```
#include <essioctl.h>
int fd_hippi, ret;

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

The argument:

`fd_hippi` is the file descriptor of an open CDI Transfer Device.

Failures and errors:

EINVAL	- ULP is not bound to this device
ENODEV	- NIC not running
EINVAL	- Device not open for O_RDONLY or O_RDWR
EINVAL	- fd_hippi is a Controlling Device (sub-device 0)

3.6 HIPIOCR_EIO

This `ioctl()` sets the receive error processing mode. When the argument is 0 (the default) a `read()` error results in `read()` returning -1 and errno being set to EIO. This results in the loss of whatever data had be received up to the error. When the argument is not 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()` to determine the `read()` completion code.

Usage:

```
#include <essioctl.h>
int fd_hippi, ret, mode;

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

The argument:

`fd_hippi` is the file descriptor of an open CDI Transfer Device.
`mode` determines processing when data errors occur. When mode is 0, EIO is generated on data error. When mode is not zero EIO is not generated on data error.

Failures and errors:

ENODEV	- NIC not running
EINVAL	- Device not open for O_RDONLY or O_RDWR
EINVAL	- fd_hippi is a Controlling Device (sub-device 0)

3.7 HIPIOCR_ERRS

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

Usage:

```
#include <essioctl.h>
int fd_hippi, ret, err_status;

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

The argument:

`fd_hippi` is the file descriptor of an open CDI Transfer Device.

`err_status` contains the result of the last read(). Multiple bits may be set. A value of 0 means no error.

HIP_DSTERR_PARITY	0x001	Destination parity error
HIP_DSTERR_LLRC	0x002	Destination LLRC error
HIP_DSTERR_FRAME	0x004	Glink - Frame error
HIP_DSTERR_SYNC	0x008	Glink - Flag sync error
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	Glink - Lost Link ready
HIP_DSTERR_RXTO	0x100	Receive timeout
HIP_DSTERR_PKTLEN	0x200	Packet length error

Failures and errors:

The ioctl return 0 when no EIO 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	- NIC not running

3.8 HIPIOCR_GET_D1

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 in the FP header is used to control the transfer. The application should provide a 1016 byte buffer for the (maximum size) D1 area.

The FP header must be read first. HIPIOCR_GET_D1 can't be called after read().

Usage:

```
#include <essioctl.h>
int fd_hippi, ret;
char buf[1016];

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

The argument:

`fd_hippi`	is the file descriptor of an open CDI Transfer Device.
`buf`	is the buffer that will receive the D1 area

Failures and errors:

EINVAL	- fd_hippi is a Controlling Device
EINVAL	- fd_hippi is not open for O_RDONLY or O_RDWR
EINVAL	- ULP is not bound
EINVAL	- Not in HIPPI-FP mode
ENODEV	- NIC not running

3.9 HIPIOCR_GET_FP

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.

This (when called) must be called before `read()`.

This ioctl blocks until a packet arrives for the bound ULP.

Usage:

```
#include <essioctl.h>
int fd_hippi, ret;
unsigned int fp[2];

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

The argument:

`fd_hippi`	is the file descriptor of an open CDI Transfer Device.
`fp`	is the buffer that will receive the FP header

Failures and errors:

EINVAL	- fd_hippi is a Controlling Device
EINVAL	- fd_hippi is not open for O_RDONLY or O_RDWR
EINVAL	- ULP is not bound
EINVAL	- Not in HIPPI-FP mode
ENODEV	- NIC not running

3.10 HIPIOCR_PKT_OFFSET

This `ioctl()` monitors the number of bytes that have been read for the packet. In separate header mode it contains the number of D2 bytes that have been read. In combined mode it contains the total number of bytes that have been read. When the whole packet has been read 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 though the packet is longer).

The file must be open for O_RDONLY or O_RDWR and a ULP must be bound.

Usage:

```
#include <essioctl.h>
int fd_hippi, ret;
unsigned int offset;

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

The argument:

`fd_hippi`	is the file descriptor of an open CDI Transfer Device.
`offset`	contains the byte offset into the packet. The maximum reported value is MAX_PKT_OFFSET bytes. If the packet is longer than that the counter stops advancing. When the whole packet has been read, offset is 0.

Failures and errors:

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	- NIC not running

3.11 HIPIOCR_SEP_HDR

This `ioctl()` sets separate header and data mode or combined headers and data (default) mode for the Transfer Device. This is effective in HIPPI-FP mode only.

In separate header mode, the FP header is read using the `HIPIOCR_GET_FP ioctl()` and the D1 area is read using the `HIPIOCR_GET_D1 ioctl()`.

In combined header and data mode, the application buffer contains the complete packet, FP header, (optional) D1 area, and (optional) D2 area.

Usage:

```
#include <essioctl.h>
int fd_hippi, ret, mode;

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

The argument:

`fd_hippi`	is the file descriptor of an open CDI Transfer Device.
`mode`	when 0 sets separate header mode. When not 0 (the default) , combined header and data mode is set.

Failures and errors:

EINVAL	- fd_hippi is a Controlling Device
EINVAL	- fd_hippi is not open for O_RDONLY or O_RDWR
ENODEV	- NIC not running
EINVAL	- NIC is in HIPPI-PH mode

3.12 HIPIOCR_TRUNCATE_PKT

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 there is no packet available to truncate, 0 is returned.

Usage:

```
#include <essioctl.h>
int fd_hippi, ret;

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

The argument:

`fd_hippi` is the file descriptor of an open CDI Transfer Device.

Failures and errors:

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	- NIC not running

3.13 HIPIOCW_CONNECT

The I-Field is cached in the CDI and is used with every packet that is sent until it is changed again. This ioctl() and HIPIOCW_I sets the I-Field in the cache. HIPIOCW_I cannot be called while in a multiple packet connection. The default I-Field is 0.

This ioctl() causes a multiple packet connection to be established when the next write() is executed. The argument is the I-Field that is to be placed in the CDI cache. The I-Field is described in Annex B. The application must make sure 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 or 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.

NOTE: When using a single packet connection, use HIPIOCW_I to set the I-Field.

Usage:

```
#include <essioctl.h>
int fd_hippi, ret;
unsigned int IField;

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

The argument:

fd_hippi	is the file descriptor of an open CDI Transfer Device.
IField	contains the I-Field that is to be used with packets that are sent over the connection. IField must be in Big Endian order.

Failures and errors:

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 has not been done.
EBUSY	- Packet is currently being sent
ENODEV	- NIC not running

3.14 HIPIOCW_D1_AREA

This HIPPI-FP `ioctl()` copies the D1 area from the supplied user buffer to the CDI D1 cache. The length is taken from the currently cached FP header. The `ioctl()` is valid in HIPPI-FP mode only. HIPIOCW_D1_SIZE must be called before this ioctl() to set the D1 length. This ioctl() clears a cached D1 area pointer.

Both `HIPIOCR_D1_AREA ioctl()` and `HIPIOCR_D1_AREA_PTR ioctl()` manage the D1 area for a packet. The last `ioctl()` called is the one that is used on all future packets until it is changed.

The `ioctl()` must not be called while a packet is being sent.

Usage:

```
#include <essioctl.h>
int fd_hippi, ret;
char d1_buf[1016], d1_ptr = &d1_buf;

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

The argument:

`fd_hippi`	is the file descriptor of an open CDI Transfer Device.
`d1_buf`	is a pointer to application supplied D1 buffer. NULL causes the cached D1 image to be discarded (D1 area is set to 0).

Failures and errors:

EINVAL	- fd_hippi is a Controlling Device
EINVAL	- fd_hippi is not open for O_WRONLY or O_RDWR
EINVAL	- D1 size is 0
EBUSY	- Packet is being sent
EINVAL	- NIC is in HIPPI-PH mode
ENODEV	- NIC not running

3.15 HIPIOCW_D1_AREA_PTR

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 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 the one that is used on all future packets until it is changed. The size of the D1 area is set using the `HIPIOCW_D1_SIZE ioctl()`. When `write()` 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 <essioctl.h>
int fd_hippi, ret;
char d1_buf[1016], d1_ptr = &d1_buf;

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

The argument:

`fd_hippi`	is the file descriptor of an open CDI Transfer Device.
`d1_buf`	is a pointer to application supplied D1 buffer. NULL causes the cached D1 pointer to be discarded.

Failures and errors:

EINVAL - fd_hippi is a Controlling Device
EINVAL - fd_hippi is not open for O_WRONLY or O_RDWR
EINVAL - D1 size is 0
EBUSY - Packet is being sent
EINVAL - NIC is in HIPPI-PH mode
ENODEV - NIC not running

3.16 HIPIOCW_D1_SIZE

This `ioctl()` is valid in HIPPI-FP mode only. It sets the size of the D1 area and, when D1 is not 0, it also sets the HIPPI-FP P bit. See Annex C for details of the HIPPI-FP header. The D1 area may be located in 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 this `ioctl()`.

The D1 size must be a multiple of 8 and it must be <= 1016.

This `ioctl()` must be called before `HIPIOCW_D1_AREA` or `HIPIOCW_D1_AREA_PTR` is called.

The `ioctl()` must not be called while a packet is being sent.

Usage:

```
#include <essioctl.h>
int fd_hippi, ret, size;

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

The argument:

`fd_hippi`	is the file descriptor of an open CDI Transfer Device.
`size`	contains 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	- NIC is in HIPPI-PH mode
EBUSY - Packet is being sent	
EINVAL	- size is invalid
ENODEV	- NIC not running

3.17 HIPIOCW_DISCONN

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 no packets have been sent, this `ioctl()` just returns to one packet per connection mode.

NOTE: to terminate a packet without terminating the connection use `HIPIOCW_END_PKT`.

Usage:

```
#include <essioctl.h>
int fd_hippi, ret;

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

The argument:

`fd_hippi` is the file descriptor of an open CDI Transfer Device.

Failures and errors:

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	- NIC not running

3.18 HIPIOCW_END_PKT

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, the resulting packet has an invalid D2 size which will appear to the Destination as a truncated packet.

The device must be open for `O_WRONLY` or `O_RDWR`.

Usage:

```
#include <essioctl.h>
int fd_hippi, ret;

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

The argument:

`fd_hippi` is the file descriptor of an open CDI Transfer Device.

Failures and errors:

EINVAL	- fd_hippi is a Controlling Device
EINVAL	- fd_hippi is not open for O_WRONLY or O_RDWR
EINVAL	- There is no packet being sent.
ENODEV	- NIC not running
EIO	- Packet was truncated

3.19 HIPIOCW_ERR

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

Usage:

```
#include <essioctl.h>
int fd_hippi, ret, err_status;

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

The arguments:

fd_hippi is the file descriptor of an open CDI Transfer Device.
err_status contains the result of the last `read()`.

HIP_SRCERR_NONE	0	No error
HIP_SRCERR_SEQ	1	HIPPI source sequence error
HIP_SRCERR_DSIC	2	Source lost DSIC
HIP_SRCERR_TIMEO	3	Source timed out transmit
HIP_SRCERR_CONNLS	4	Source lost CONNECT signal
HIP_SRCERR_RE	5	Connect request was rejected
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.

EINVAL - fd_hippi is a Controlling Device
EINVAL - fd_hippi is not open for `O_WRONLY` or `O_RDWR`
ENODEV - NIC not running

3.20 HIPIOCW_I

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

The argument is the new I-Field. The I-Field is described in Annex B. The application must make sure 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 0.

The file must be open for `O_WRONLY` or `O_RDWR`.

The application must not be in the process of sending a packet. The application must not have a multiple packet connection active.

NOTE: When using a many packet connection, use `HIPIOCW_CONNECT` to set the I-Field.

Usage:

```
#include <essioctl.h>
int fd_hippi, ret;
unsigned int IField;

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

The arguments:

`fd_hippi`	is the file descriptor of an open CDI Transfer Device.
`IField`	contains the I-Field that is to be placed in the CDI cache. IField must be in Big Endian order.

Failures and errors:

EINVAL	- fd_hippi is a Controlling Device
EINVAL	- fd_hippi is not open for `O_WRONLY` or `O_RDWR`
ENODEV	- NIC not running
EBUSY	- Sending a packet
EBUSY	- Multiple packet connection active
EINVAL	- currently in multiple packet connection

3.21 HIPIOCW_SEP_HDR

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.

In separate header mode, the FP header is taken from the CDI FP header cache. When the D1 size is not 0, the D1 header is placed in the packet. If the CDI D1 cache contains a header (`HIPIOCW_D1_AREA` has been called) the header is included in the packet. Alternatively, the D1 area may be in application buffer and `HIPIOCW_D1_AREA_PTR` may 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, (optional) D1 area, and (optional) D2 area. 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 a more extensive analysis and may fail the operation.

The `ioctl()` must not be called while a packet is being sent.

Usage:

```
#include <essioctl.h>
int fd_hippi, ret, mode;

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

The argument:

`fd_hippi`	is the file descriptor of an open CDI Transfer Device.
`mode`	when not 0 (the default) sets separate header mode. When 0, combined header and data mode is set.

Failures and errors:

EINVAL	- fd_hippi is a Controlling Device
EINVAL	- fd_hippi is not open for O_WRONLY or O_RDWR
EINVAL	- NIC is in HIPPI-PH mode
EBUSY	- Packet is being sent
ENODEV	- NIC not running

3.22 HIPIOCW_SET_ULP

This 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 this `ioctl()`. This `ioctl()` is only valid in HIPPI-FP mode.

If HIPIOC_BIND_ULP is called before this `ioctl()`, the bind ULP will be placed in the FP header cache.

The device file must be open for `O_WRONLY` or `O_RDWR`. The NIC must be running in HIPPI-FP mode.

The CDI must not be actively sending a packet and the CDI must be in separate header mode for write.

ULPs in the range 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 application. The remaining ULPs in the range 0-127 are reserved by the HIPPI-FP Standard for future use. They may be bound to applications that have root privilege.

Usage:

```
#include <essioctl.h>
int fd_hippi, ret, ULP_id;

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

The argument:

`fd_hippi`	is the file descriptor of an open CDI Transfer Device.
`ULP_id`	is the ulp that is being set.

Failures and errors:

EINVAL	- Not in separate header mode
EBUSY	- Packet is being sent
EINVAL	- Reserved ULP
EINVAL	- ULP > 255
EPERM	- ULP < 128 and not superuser
ENODEV	- NIC not running
EINVAL	- Device not open for `O_WRONLY` or `O_RDWR`
EINVAL	- fd_hippi is a Controlling Device
EINVAL	- NIC is in HIPPI-PH mode

3.23 HIPIOCW_SHBURST

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. This `ioctl()` must be called for every packet that is 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 and D1 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 or `D2SIZE_UNKNOWN`. The byte_count is ignored.

The byte_count must be a multiple of 8 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 0 yields a standard size (1024 byte) first burst (cancels previous call to this ioctl()).

A short first burst is occasionally required by Destination hardware. The Essential NICs do not require the use of a short first burst.

Usage:

```
#include <essioctl.h>
int fd_hippi, ret;
int byte_count;

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

The arguments:

`fd_hippi` is the file descriptor of an open CDI Transfer Device.
`byte_count` is used in HIPPI-PH mode. It contains the length of the short first burst. In HIPPI-FP mode set it to 0. byte_count must be a multiple of 8 and <= 1016. The first write for the packet must be at least as large as the first burst.

Failures and errors:

EINVAL	- fd_hippi is a Controlling Device
EINVAL	- invalid byte_count is specified
EINVAL	- fd_hippi is not open for O_WRONLY or O_RDWR
ENODEV	- NIC not running

3.24 HIPIOCW_START_PKT

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

The byte_count is interpreted differently in different situations.

In HIPPI-FP mode, when separate headers are used, byte_count is the length of the D2 area.

In HIPPI-FP mode, when combined header and data is used, the byte_count is the length of the entire packet.

In HIPPI-PH mode, the byte_count is the length of the entire packet

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

Usage:

```
#include <essioctl.h>
int fd_hippi, ret;
int byte_count;

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

The arguments:

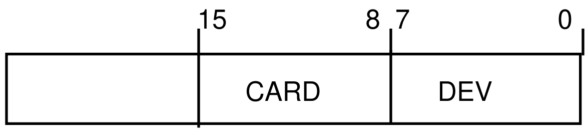
fd_hippi is the file descriptor of an open CDI Transfer Device.
byte_count is described above.

Failures and errors:

EINVAL	- fd_hippi is a Controlling Device
EINVAL	- invalid byte_count is specified
EINVAL	- fd_hippi is not open for O_WRONLY or O_RDWR
EINVAL	- Already processing multiple write packet
ENODEV	- NIC not running

Section 4
Special Files

The CDI supports multiple NICs in the Host system and multiple Devices on each NIC. The device minor field (last field in the mknod utility invocation) is divided into an 8-bit card field and an 8-bit sub-device field.



4-1 Device Minor Field

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. `read()` and `write()` 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 `read()` and `write()`.

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 0. The naming convention for special files is:

h<card>
h<card>.<sub-device>

where <card> is the card number (as determined during system bootup). The card number is represented as decimal digits. The <sub-device> is used on all Transfer Devices (1..31). It is also represented as decimal digits. For example:

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 and 31 Transfer Devices for each NIC.

The essioctl.h header file defines NIC0 through NIC15.

Annex - A
Understanding HIPPI

This Annex provides background information that is helpful in understanding the underlying HIPPI technology.

A.1 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 completed, the connection is terminated. The sender can wait for a connection (CampON) or give up if the Destination is busy.

Each NIC provides a Source (transmitter) and a separate Destination (receiver). The Source and Destination are independent and may operate concurrently. Since HIPPI transfers 800,000,000 bits/sec independently on the Source and Destination, the NIC is capable of (burst) transferring 1,600,000,000 bits/sec over the HIPPI media. HIPPI provides a hardware flow control mechanism when the maximum rates cannot be sustained due to host system constraints such as IO 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 Annex B 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 host computer systems and other peripherals. The HIPPI fabric (usually) assigns (using administrator intervention) a unique logical address to each End Point.

HIPPI media may be either copper cables or optical cables. Essential provides NICs that are based on the RoadRunner ASIC and provide optical media connections. The Essential NICs include both Source and Destination interfaces. Essential also provides HIPPI switches that can be configured with a combination of optical and copper media cards.

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.

A.2 HIPPI Connection Processing

HIPPI passes data packets over a point to point connection. The connection must be established before packets can be transferred. Once a connection is established, one or more packets may 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.

A.3 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) (see Annex D). 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 make sure 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 requires 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 Annex C for more information.

A.4 HIPPI I-Field

An I-Field (see Annex B) introduces each connection. 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.

A.5 HIPPI-FP Operation

The HIPPI-FP header, see Annex C, contains information about the packet. 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. It is in the first burst. When short first burst is indicated, the FP header and D1 area are in the first burst.

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

A.6 HIPPI-PH Operation

HIPPI-PH does not use headers. The packet starts with a start packet indicate and ends with a end packet indicate. The packet consists of one or more bursts. A full size burst is 1024 bytes. There may be exactly one short burst 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.

Annex - B
HIPPI-SC Excerpts

This 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.

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 long. 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 0. The next 4 bytes are as described below.

The CCI is in Big Endian on the media. Little Endian Hosts must byte swap.

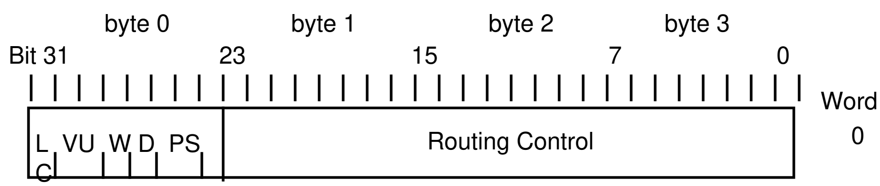


Figure B-1 CCI and I-Field Format

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 (30-0) are locally defined and are not defined by the standard.

VU - Vendor Unique (bits 30-29) - 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/sec data rate option. The NIC requires that this bit be set to 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 current sub-field.

PS - Path Selection (bits 26-25) - Used to select either (1) a source route, i.e., 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 shall reply with a connection reject sequence if unable to complete the connection. C=1 specifies that the switch shall attempt to establish a connection until either the connection is completed or the Source aborts the connection request.

Note that the NIC requires that the W bit be 0.

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

ANNEX - C
HIPPI-FP Excerpts

This 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 for proper send operation through the NIC. Fields in this header are used by the NIC 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 Host systems must perform the needed byte swaps.

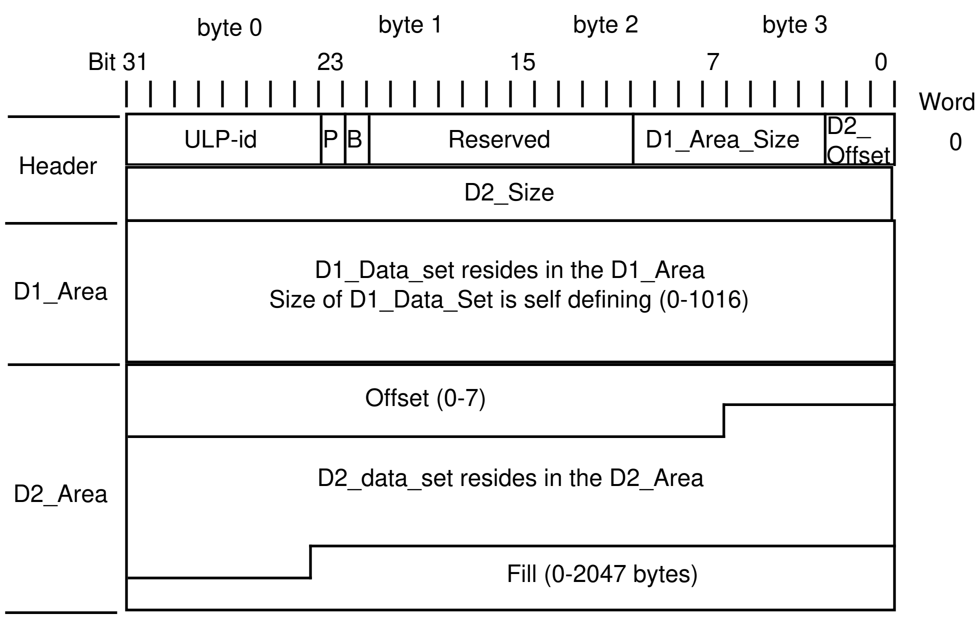


Figure C-1 FP Packet Format

The packet data presented by the Source ULP with a FP_TRANSFER Request primitive shall be transferred to the Destination with a HIPPI-FP header as shown in figure C-1. The HIPPI-FP packets shall be composed of three areas, (1) Header, (2) D1_Area, and (3) D2_Area, each starting and ending on a 64 bit boundary. If the 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 shall be the first 64 bits of the packet, and shall be completely contained in the first burst.

ULP-id - (8-bits) designates the Destination ULP to which the packet is to be delivered. ULPs in the range 0-128 are reserved by the HIPPI standards organization. ULPs in the range 128 through 255 inclusive are available to the application. The following 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 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) is 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, 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 shall immediately follow the Header_Area, shall be completely contained in the first burst, shall contain an integral number of 64 bit Double Words, and shall contain the D1_Data_Set (if present).

If Present, the D1_Data_Set shall be the first information in the D1_Area. If the P bit of the Header is 0, then the D1_Data_Set is not present. The D1_DataSet 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. The maximum size is 1016 bytes.

D2_Area

The D2_Area, if D2_Size is not zero, shall immediately follow the D1_Area, shall start and end on 64 bit Double Word boundaries, and shall contain the D2_Data_Set. If the B bit in the Header is 1, the D2_Area shall start 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_DataSet and the end of the D2_Area. When D2_Size is all binary ones, then there is no fill.

Annex - D
HIPPI-PH Excerpts

The following Physical framing hierarchy describes the data flow over HIPPI media.

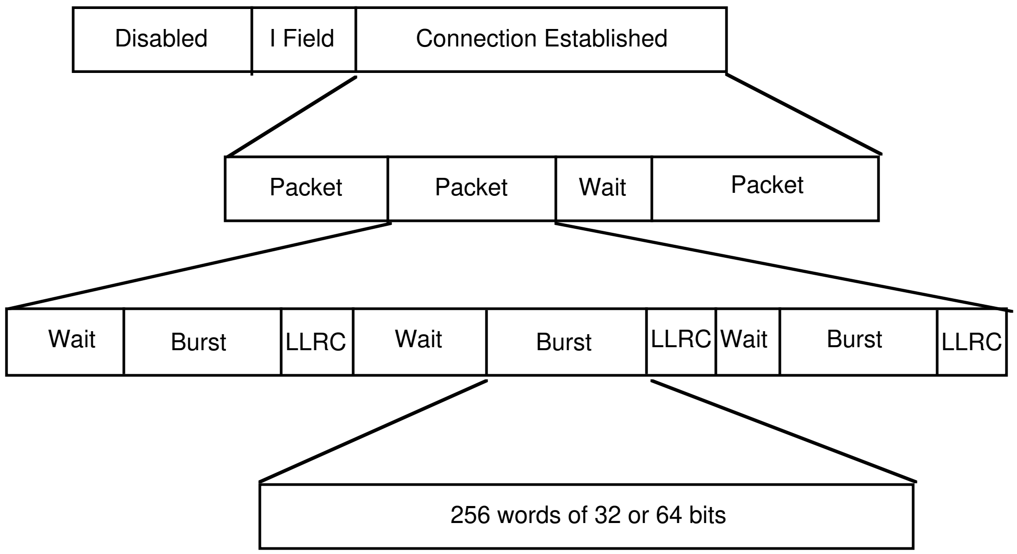


Figure D-1 Physical Framing Hierarchy

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 which permits 128k packets being 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.

Annex - E
Troubleshooting

Problems arise in communicating using HIPPI. This Annex outlines some techniques that are helpful in isolating the problems.

The installation provides two utilities *blast* and *sink* that may be used to verify the ability to send and receive packets. *Blast* sends packets and *sink* receives packets (see the man pages for more details).

On Digital Unix and HP-UX systems the first (or only) HIPPI device is hip0. On Silicone Graphics systems the HIPPI device is ess0.

E.1 Test the NIC

The installation process results in a properly operating NIC. (See the installation instructions for the Host system)

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. Packets that are sent are electrically looped back to the receive electronics. These administrative activities are run as "super user" or "root".

```
# su root
# esshippi off
# ifconfig hip0 down
# esshippi on loopback
# sink -n 10 &
# blast -n 10
# esshippi status
```

On SGI, ess0 instead of hip0.

The "off" turns off RunCode.

The "ifconfig" is only needed if the NIC as been configured for network use. "ifconfig hip0" will indicate whether or not the interface is configured.

The "on loopback" places the NIC in internal loopback mode.

The "*sink*" sets up to receive 10 default (4k) packets in the background.

The "*blast*" sends 10 default packets. The blast and the background jobs for *sink* should complete.

The "status" reports that 10 packets have been send (Source) and received (Destination). There should not be any errors.

If the above fails, either the installation failed or the NIC is defective. In either case call Essential support for assistance.

E.2 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.

It is very important that the optical surfaces at the ends of cables and in the optical modules remain clean. Erratic behavior can result from surface contamination. A cleaning kit is available through Essential.

E.3 Optical Connections

An optical cable connects two HIPPI interfaces. The HIPPI interfaces may be:

- 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 loopback cable. A loopback 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.

```
# su root
# esshippi on network switched
# esshippi status
```

The "on" command turns on RunCode and sets the NIC to communicate over the optical cable. The "status" command indicates 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 try the "status" command again.

E.4 Optical Loopback Test

The optics module of the NIC may be tested using a loopback cable. Use the loopback cable to connect both connectors on the NIC. When the optical loopback test passes, the NIC is installed and operating properly. It can both send and receive packets over an optical cable.

```
# su root
# esshippi off
# ifconfig hip0 down
# esshippi on network switched
```

```
# sink -n 10 &
# blast -n 10
# esshippi status
```

On SGI, ess0 instead of hip0.

The "off" turns off RunCode.

The "ifconfig" is only needed if the NIC as been configured for network use. "ifconfig hip0" will indicate whether or not the interface is configured.

The "on network switched" configures the NIC to use the optics module.

The "*sink*" sets up to receive 10 default (4k) packets in the background.

The "*blast*" sends 10 default packets. *Blast* and the background job for *sink* should complete.

The "status" reports that 10 packets have been send (Source) and received (Destination). There should not be any errors.

If the above fails, optics module on the NIC is defective. Call Essential support for assistance.

E.5 Optical Loopback 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 it is logical address 0x223.

Verify that the HIPPI interfaces are properly connected as described above.

```
# su root
# esshippi off
# ifconfig hip0 down
# esshippi on network switched
# sink -n 20 &
# blast -n 10 -I0x0100000a
# blast -n 10 -I0x07000223
  (wait for the background jobs to complete)
# esshippi status
```

On SGI, ess0 instead of hip0.

The "off" turns off RunCode.

The "ifconfig" is only needed if the NIC as been configured for network use. "ifconfig hip0" will indicate whether or not the interface is configured.

The "on network switched" configures the NIC to use the optics module.

The "*sink*" sets up to receive 20 default (4k) packets in the background.

The first "*blast*" sends 10 default packets using source routing (with CampON). The packets are sent to the address of the NIC (packets are looped back through the switch to the NIC). The background *blast* job completes when the 10 packets have been sent.

The second "*blast*" sends 10 default packets using destination routing (Logical Address) (with CampON). The packets are sent to the address of the NIC (packets are looped back through the switch to the NIC). Both background jobs complete when the 10 packets have been sent.

The "status" reports that 20 packets have been send (Source) and received (Destination). There should not be any errors.

If the above fails, the switch is improperly configured. The switch administrator should be able to fix the problem.

E.6 Optical Testing Between NICs

The NICs may be connected through a switch or directly connected. This test makes sure that the 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 this 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.

Verify that the HIPPI interfaces are properly connected as described above.

On NIC A

```
# su root
# esshippi off
# ifconfig hip0 down
# esshippi on network switched
# sink -n 20 &
```

(wait for the background job to complete)

```
# esshippi status
```

On NIC B

```
# su root
# esshippi off
# ifconfig hip0 down
# esshippi on network switched
# blast -n 10 -I0x01000003
# blast -n 10 -I0x07000143
```

(wait for the background jobs to complete)

```
# esshippi status
```

On SGI, ess0 instead of hip0.

The "off" turns off RunCode.

The "ifconfig" is only needed if the NIC as been configured for network use. "ifconfig hip0" will indicate whether or not the interface is configured.

The "on network switched" configures the NIC to use the optics module.

The "*sink*" on NIC A sets up to receive 20 default (4k) packets in the background.

The first "*blast*" (from NIC B) sends 10 default packets using source routing (with CampON). The packets are sent to the address of the NIC (packets are looped back through the switch to the NIC). The background *blast* job completes when the 10 packets have been sent.

The second "*blast*" (from NIC B) sends 10 default packets using destination routing (Logical Address) (with CampON). The packets are sent to the address of the NIC (packets are looped back through the switch to the NIC). Both background jobs complete when the 10 packets have been sent.

The "status" on NIC A reports that 20 packets have been received (Destination). The "status" on NIC B reports that 20 packets have been send (Source). There should not be any errors.

On NIC B

```
# su root
# sink -n 20 &
   (wait for the background jobs to complete)
# esshippi status
```

On NIC A

```
# su root
# blast -n 10 -I0x01000008
# blast -n 10 -I0x07000846
   (wait for the background jobs to complete)
# esshippi status
```

The "*sink*" on NIC B sets up to receive 20 default (4k) packets in the background.

The first "*blast*" (from NIC A) sends 10 default packets using source routing (with CampON). The packets are sent to the address of the NIC (packets are looped back through the switch to the NIC). The background *blast* job completes when the 10 packets have been sent.

The second "*blast*" (from NIC A) sends 10 default packets using destination routing (Logical Address) (with CampON). The packets are sent to the address of the NIC (packets are looped back through the switch to the NIC). Both background jobs complete when the 10 packets have been sent.

The "status" on NIC B reports that 20 packets have been received (Destination). The "status" on NIC A reports that 20 packets have been send (Source). There should not be any errors.

If the above fails, the switch is improperly configured. The switch administrator should be able to fix the problem. If the above fails with directly connected NICs contact Essential support for assistance.

When this test passes, the NICs and switch ports are properly configured.

E.7 Long Packets Between NICs

The default operation of the driver is to limit packet size to 64K as recommended by the network standards. When the CDI is used with applications that require larger packet size the NIC can be configured to permit longer packet to be sent.

The above is effective as long as the host is running. When the host is rebooted, operation reverts to parameters that are set in the EEPROM on the NIC.

```
# su root
# esshippi off
# ifconfig hip0 down
# esshippi on long
# ifconfig hip0 up
```

On SGI systems the NIC is ess0 not hip0.

The "off" turns off RunCode.

The "ifconfig down" is only needed if the NIC as been configured for network use. "ifconfig hip0" will indicate whether or not the interface is configured.

The "on long" configures the NIC to permit sending packets that are longer than 64K.

The "ifconfig up" is only needed if the NIC as been configured for network use.

The above is effective as long as the host is running. When the host is rebooted, operation reverts to parameters that are set in the EEPROM on the NIC. The EEPROM can be changed so that the NIC will automatically come up in the desired mode when the host is booted.

```
# su root
# esshippi off
# esshippi on long
# esshippi off
# esstune -l -e
# esshippi on
```

The "off" turns off RunCode.

The "on long" configures the NIC to permit sending packets that are longer than 64K.

The second "off" is needed since the RunCode must be off before accessing the EEPROM.

The "esstune" writes the current mode in to the EEPROM.

The "on" restarts the NIC, the "long" is remembered until the host system is rebooted.

References

The following documents are useful in understanding the context in which the NIC operates. It is always encouraged to use the latest revision of each document.

ANSI X3.183-1991, High-Performance Parallel Interface, Mechanical, Electrical, and Signaling Protocol Specification (HIPPI-PH).

Serial-HIPPI Specification, Revision 1.0, Serial HIPPI Implementors Group, May 17,1991

ANSI X3.218-199x, High-Performance Parallel Interface - Framing Protocol (HIPPI-FP)

ANSI X3.218-199x, High-Performance Parallel Interface - Encapsulation of ISO 8802-2 (IEEE Std 802.2) Logical Link Control Protocol Data Units (HIPPI-LE)

ANSI X3.222-199x, High-Performance Parallel Interface - Physical Switch Control (HIPPI-SC)

IEEE Standard 802.1A

Glossary

Big Endian: most significant bit of data of a multiple byte object is at the lowest byte address

burst: unit of transfer on a HIPPI media. Full size burst is 1024 bytes

Card: Essential Communications Corp. HIPPI/PCI Network Interface Card (NIC)

CDI: Character Device Interface

connection: path from a HIPPI source to a HIPPI destination over which packets are passed

Controlling Device: special file that is used to control a NIC

Destination: HIPPI receiver that can accept Packets

DMA: Direct Memory Access. Data is moved directly between the NIC and Host memory without passing through the CPU

Double Word: 8 byte object

duplex: Device 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: Device can simultaneously send and receive data

HIPPI: High Performance Parallel Interface 800 or 1600 M bit/sec communication media

Half Word: 2 byte object

Host: System in which the NIC is operating

Host Software: Software that is executing on the Host system

IO: input/output

K: 1024

KB: 1024 bytes

logical address: HIPPI-SC address of a HIPPI destination

Little Endian: 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 are initiated and controlled by the NIC

MB: 1024*1024 bytes

msec: millisecond, 1.0E-3 seconds

NIC: Network Interface Card

nsec: nanosecond, 1.0E-9 seconds

Packet: A Packet as defined in HIPPI-PH

Road Runner: ASIC that is the heart of NIC. It controls processing and the host interface

RunCode: Firmware used during normal operation of the NIC

sec: second

simplex: One way communication path. Device can either send or receive

Single Wide: HIPPI operating at 800 M bit/second

Source: HIPPI transmitter that can generate Packets

Transfer Device: special file opened by an application for reading and writing data

ULP: HIPPI-FP upper layer protocol identifier

usec: microsecond, 1.0E-6 seconds

Word: 4 byte object