Chapter 2

Basic CM5-HIPPI Concepts

2.1 Introduction

This chapter discusses a number of concepts that underly the creation and use of sockets in a CM5-HIPPI environment. The concepts covered here are:

- Socket Associations — Section 2.2
- Address domains — Section 2.3
- AF_HIPPI Socket types and Protocols — Section 2.4
- Socket address structures — Section 2.5
- Connection Control: the `ifield` — Section 2.6
- Wildcards — Section 2.7
- Connecting Sockets — Section 2.8
- Reading from HIPPI — 2.9.1
- Writing to HIPPI — 2.9.4

While many of these concepts will be very familiar to those experienced in the UNIX and/or Internet environments, they are covered here with a focus on some CM5-HIPPI particulars. For a more comprehensive discussion of

Copyright © 1994 Thinking Machines Corporation

sockets, refer to the Sun Microsystems *Network Programming Guide* or other UNIX networking documentation.

2.2 Socket Associations

A prerequisite for socket-based communication between two processes is for each process to create a local socket and establish an *association* between them. A central feature of this association is the sharing of address information between sending and receiving sockets. In the CM5-HIPPI environment, this address sharing is accomplished via the address structure, `struct sockaddr_hippi`.

`sockaddr_hippi` defines, among other socket attributes, address-pair associations. Each address pair represents the two end-points (sockets) in a communication path between two processes. For any given packet transfer, one address is regarded as the source address and the other the destination address. Because socket connections can be bi-directional, these address pairs may trade source−destination roles.

In addition to the socket address association, `sockaddr_hippi` defines a number of other socket attributes that influence how communcation between sockets is carried out. The various socket attributes are described in Section 2.5, *Socket Address Structures*. The user initializes `sockaddr_hippi` with the applicable socket attributes, including the appropriate address association. Address structure initialization is described in Section 3.3.

A socket is conditioned to *receive* packets by being *bound* to a name, which serves as a pointer to the address structure containing its source−destination address pair. Another socket can then *connect* to the bound socket and/or *send* message packets to it by name.

Socket binding is described in Sections 2.5 and 3.4. Socket connections are described in Sections 2.8 and 3.5. Receiving (reading) packets is described in Section 2.9.1. Sending (writing) packets is described in Section 2.9.4.

Copyright © 1994 Thinking Machines Corporation

2.3 Address Domains

When a socket is created, it exists within a particular *address domain*. A socket's domain defines the manner in which other sockets within the domain will reference it — that is, each domain supports a specific addressing scheme that is used to steer message packets between network endpoints.

For example, in the UNIX domain, referred to as AF_UNIX, sockets are referenced by pathnames — just as if they were files. An AF_UNIX socket might be called `/dev/my_socket`. Sockets in the Internet domain (AF_INET) are defined by a host IP address (a 32-bit value) and a port number.

NOTE

Both of these domains are well documented in other network literature and are described here only in passing. For a broader discussion of these domains and socket programming in general, refer to the Sun Microsystems *Network Programming Guide* or other UNIX networking documentation.

In the CM5-HIPPI address domain (i.e., AF_HIPPI), sockets are referenced by an `ifield`, a 32-bit value that contains a 24-bit address field and a set of low-level control bits. `ifield`s are discussed more fully in Section 2.6. For some socket types, the message can be passed to a higher-layer protocol using an Upper-Layer Protocol ID (`ulp-id`). This mechanism for passing a message to another protocol at a higher layer is described in Section 2.4.2.

FP packets may also reference higher-level protocol sockets, such as IP, by means of an embedded Upper Layer Protocol ID (`ulp-id`). This

Copyright © 1994 Thinking Machines Corporation

2.4 AF_HIPPI Socket Types and Protocols

When a socket is created, its type must be specified. The AF_HIPPI domain supports two socket types, `SOCK_RAW` and `SOCK_DGRAM`. A single protocol is associated with each.

The protocol used for `SOCK_RAW` sockets is HIPPIPROTO_PH. This protocol conforms to the physical layer protocol defined in the American National Standards Institute (ANSI) document X3.183−199x, Revision 8.1, *HIPPI−PH*. This protocol is discussed more fully in Section 2.4.1.

The protocol used for `SOCK_DGRAM` sockets is HIPPIPROTO_FP. This conforms to the framing packet protocol defined in the ANSI document X3.210−199x, Revision 4.3, *HIPPI−FP*. This protocol is discussed more fully in Section 2.4.2.

NOTE: The Internet Protocol, IP, is layered on top of HIPPIPROTO_FP (per RFC 1374). It can be accessed using the standard AF_HIPPI IP mechanism. This subject is discussed in Section 2.4.3.

Both socket types support datagram-style communication. This is a packet-based communication mode in which the connection between the sender and receiver lasts only for the duration of a single packet transfer. A new connection must be established for each packet, and the connection is automatically broken at the end of the packet.

2.4.1 HIPPIPROTO_PH

HIPPIPROTO_PH provides the lowest level of communication over the HIPPI link. The only header information used by this protocol is a 32-bit `ifield`, which precedes the data portion of a message packet. See Figure 3.

The `ifield` is described by the `hippi_cci_addr` structure in `<sys/nethippi/hippi.h>`. It is discussed more fully in Section 2.6, *Connection Control: the ifield*

NOTE: HIPPPIPROTO_PH does not include provision for encapsulating any higher level protocol. Consequently, PH and FP sockets must be kept separate in the CM5-HIPPI address space. FP packets must not be sent to PH sockets and vice-versa.

CMOST Version 7.3, April 1994 — Preliminary Edition
Copyright © 1994 Thinking Machines Corporation

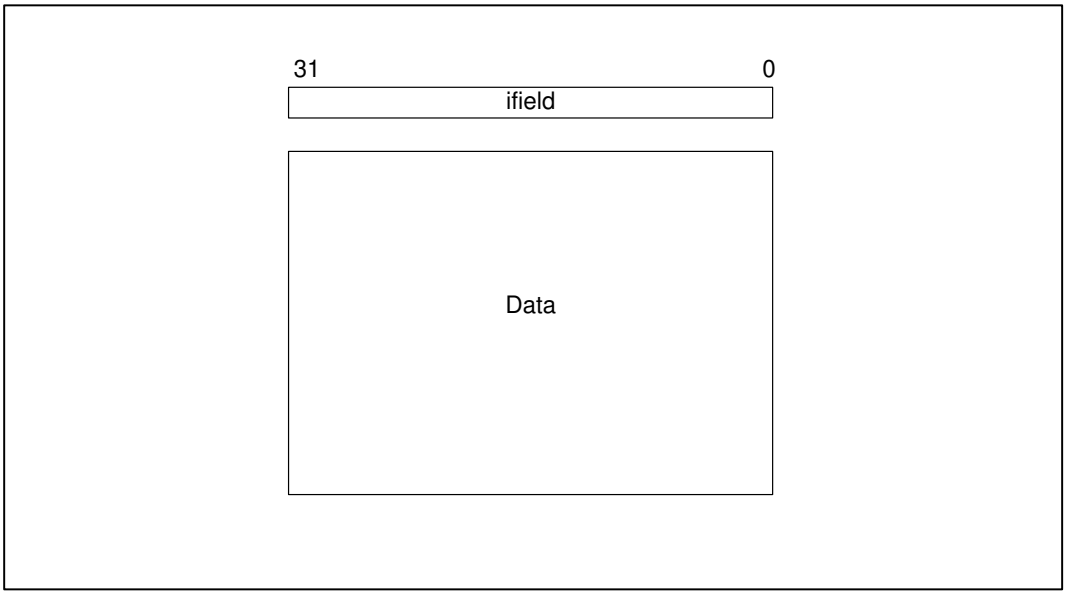


Figure 3. HIPPIPROTO_PH data packet.

2.4.2 HIPPIPROTO_FP

HIPPIPROTO_FP is layered on top of HIPPIPROTO_PH. Its message packets are divided into three areas: a header, a D1_area, and a D2_area. The D1_area begins and ends on a 64-bit boundary. The D2_area is byte-aligned. This layout is illustrated in Figure 4.

Copyright © 1994 Thinking Machines Corporation

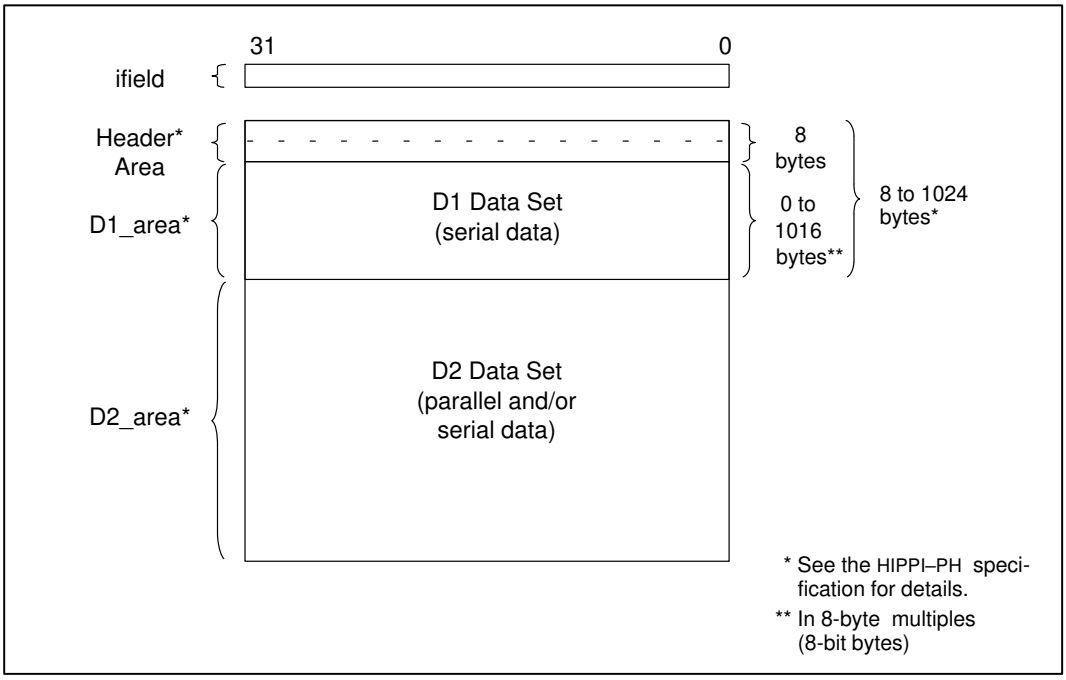


Figure 4. HIPPIPROTO_FP data packet.

The HIPPIPROTO_FP header is defined by the `hippi_fp_header` structure; see Figure 5. The various fields that comprise the header are described below.

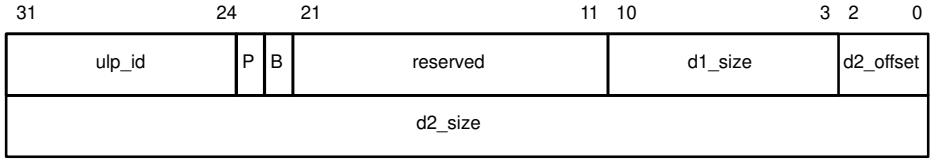
Field	Bits	Definition
ulp_id	31—24	This 8-bit value identifies the upper level protocol to which the packet is to be delivered. Any value in the range 0x80—8F (> 01111111) can be used. See ANSI document X.210−199x, Revision 4.3, HIPPI−FP for additional information.
P	23	Set to 0 if D1 data set is not included.
B	22	This bit controls whether or not to start D2_area on a burst boundary. It is always set to 0 on output (i.e., no user control) and ignored on input.
reserved	21—11	This field is unused. It is set to all 0's.
d1_size	10—3	This is the size of the D1_area in **8−byte** words.

Copyright © 1994 Thinking Machines Corporation

d2_offset	2—0	This specifies the number of bytes from the start of the D2_area to the start of the D2_data_set. It is set to 0 on output. Any non-zero value on input is strongly discouraged and may have adverse impact on performance.
d2_size	31—0	This is the size of the D2 data set in bytes.

```
struct hippi_fp_header {
  unsigned  ulp_id        :8,   /*  Destination ULP */
  P                       :1,   /*  D1_Data_Set_Present */
  B                       :1,   /*  Start_D2_on_Burst_Boundary, */
                                /*  always 0 */
   reserved               :11,  /*  unused */
  d1_size                 :8,   /*  D1_Area_Size in 64-bit words */
  D2_offset               :3 ;  /*  bytes between start of D2_Area */
                                /* and D2_Data_Set*/
  unsigned long d2_size   ;     /*  size of D2_Data_Set in bytes */
};

typedef struct {
  struct sockaddr_hippi  sa ;
  struct hippi_fp_header fph ;
} Hippi_fp_hdr_t ;
```



FP Header Format

Figure 5. HIPPIPROTO_FP header structure.

The various read and write operations follow different rules for how they deal with the header, D1, and D2_areas. These differences are described in Sections 2.9.1 and 2.9.4.

Copyright © 1994 Thinking Machines Corporation

2.4.3 IP over HIPPI

Datagram IP (IP-over-HIPPI) is invoked when the value of a received FP header's `ulp` field is 4. This will turn message handling control for the socket over to the IP protocol handler.

2.5 Socket Address Structures

In order for a socket to receive any data, it must be associated with an address structure that other sockets in its domain can reference. An application makes this association using the system call, **`bind()`**. The arguments to **`bind()`** associate **`*name`** with the specified socket. **`*name`** is a pointer to an address structure that uniquely identifies the named socket within its address domain.

This section describes the address structure used by AF_HIPPI sockets. Instructions for using **`bind()`** are provided in Section 3.4.

The AF_HIPPI address structure, `sockaddr_hippi`, is defined in `<sys/nethippi/hippi.h>`. Figure 6 shows the `sockaddr_hippi` portion of the `<sys/nethippi/hippi.h>` file. The complete `<sys/nethippi/hippi.h>` file is illustrated in Appendix A.

```
struct sockaddr_hippi {
  u_short           family    ;   /* AF_HIPPI */
  u_char            if_unit   ;   /* HIPPI interface number */
  u_char            ulp       ;   /* upper layer protocol id
                                    /* (FP only) */
  Hippi_cci_addr_t  ifield    ;   /* ifield */
  u_char            protocol  ;   /* HIPPIPROTO_PH or HIPPIPROTO_FP */
  u_char            wildcards ;   /* wildcard field flags */
  char              unused[6] ;
} ;
```

Figure 6. AF_HIPPI address structure.

Copyright © 1994 Thinking Machines Corporation

The various fields `sockaddr_hippi` are explained below.

`family`	This field identifies the protocol family to which the socket belongs. It is synonymous with *domain*. This value is always AF_HIPPI.
`if_unit`	This is the number that the network administrator assigns to the CM5-HIPPI. It is 0 for a single CM5-HIPPI. In system's with multiple CM5-HIPPI interfaces, the unit numbers comprise a contiguous range, beginning with 0.
`ulp`	This field specifies an upper layer protocol ID for HIPPIPROTO_FP protocol transfers. It is not used for HIPPIPROTO_PH transfers.
`ifield`	This is a 32-bit field that includes a 24-bit address mask and an 8-bit control field. The address mask has two modes of use, logical and source route. These modes are discussed more fully in Section 2.6.
`protocol`	This specifies the protocol to be used for processing messages. The two protocols supported for CM5-HIPPI interfaces are HIPPIPROTO_PH and HIPPIPROTO_FP. See Section 2.4 for a detailed discussion of these protocols.
`wildcards`	This field can be used to set the address mask to match *any* value. For FP sockets, the `ulp` field can be wildcarded as well. A `ulp` field can be wildcarded in either logical address mode or source route addressing. All other uses of wildcards can only be done in logical mode addressing. See Section 2.7 for a discussion of wildcard use.
`unused`	These 6 bytes are not used.

Copyright © 1994 Thinking Machines Corporation

2.6 Connection Control — the ifield

When a process forwards a packet to a remote socket, it associates a 32-bit value, called an `ifield`, with the packet. The ANSI standard for the HIPPI-SC (switch control) layer defines the `ifield` as a connection control parameter for routing packets across the HIPPI network.

The system administrator defines `ifield`s for a HIPPI network when the network's address maps are set up. The application developer then uses these `ifields` when initializing the `struct sockaddr_hippi` for a local socket. This initialization is done to enable the socket to exchange data with a remote socket..

Once the address structures for a HIPPI transfer's two end points have been initialized, `ifield`s are transparent to the user application. The following summary of their implementation is provided for background interest only.

An `ifield` can be used in either of two modes — *source route addressing* or *logical addressing*.

- In source route addressing, the source explicitly prescribes how each switch along a packet's source-destination route is to behave — i.e., it specifies which path the packet will take as it leaves each switch. This has the effect of setting up a point-to-point connection between the source and destination end-points.

- For logical addressing, the `ifield` carries the logical address of the destination end-point as well as the address of the source end-point. The receiving system then compares the logical address contained in the `ifield` with its own address. It can also compare the source address with the remote address binding table to determine whether or not it should accept a message from that source.

  If a wildcard has been specified in the destination socket's address structure, the applicable address matching function(s) will be overridden.

Source route and logical addressing are discussed more fully in Sections 2.6.1 and 2.6.2.

The `ifield`'s 8 most-significant bits contain control information, including the addressing mode control bits. The remaining 24-bit field carries address information, whose format depends on the addressing mode in effect. See Figure 7.

Copyright © 1994 Thinking Machines Corporation

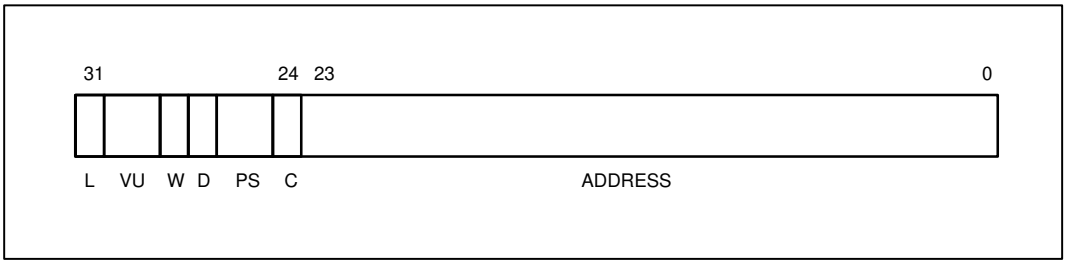


Figure 7. ifield control bits.

The control bits, 31—24, are defined below.

Field	Bits	Definition
L	31	`Local` — This bit must be 0 to ensure that the `ifield` is interpreted according to the standard and not locally defined.
VU	30—29	`Vendor Unique` — These bits are defined by the vendor; they are meaningful to some HIPPI devices, such as frame buffers. They may be set to any value on output and are passed to the application on input.
W	28	`Width` — Always set to 0; this represents the 32-bit wide HIPPI channel, or 800 Mbit/second data rate.
D	27	`Direction` — This bit controls the interpretation of the 24-bit address field. For source route addressing, bit 27 controls the direction in which the address bits are shifted as the packet travels through the switch network. See Section 2.6.1. For logical addressing, it controls which set of 12 bits (left or right) is interpreted as the source address and which is the destination address. See Section 2.6.2.

Copyright © 1994 Thinking Machines Corporation

PS	6—25	`Path Selection` — These bits control the address mode as follows: 00 = source routing; see Section 2.6.1 01 = logical address; see Section 2.6.2 10 = reserved 11 = logical address
C	24	`Camp On` — 0 causes the switch to notify the source if it cannot complete a connection. 1 causes the switch to continue trying to complete a connection until either the connection is successful or the source aborts the connection request.

2.6.1 Source Route Addressing

For a socket to be able to receive packets by source route addressing, it must be bound to a `struct sockaddr`, whose `ifield` may be interpreted as a `struct cci_src_rt_addr`. Figure 8 shows the `ifield` format and the address structure for this addressing mode. Figures 9 and 10 illustrate the changes a source route `ifield` undergoes as it moves through a switched network.

This `ifield` address structure is defined in `<sys/nethippi/hippi.h>`.

Copyright © 1994 Thinking Machines Corporation

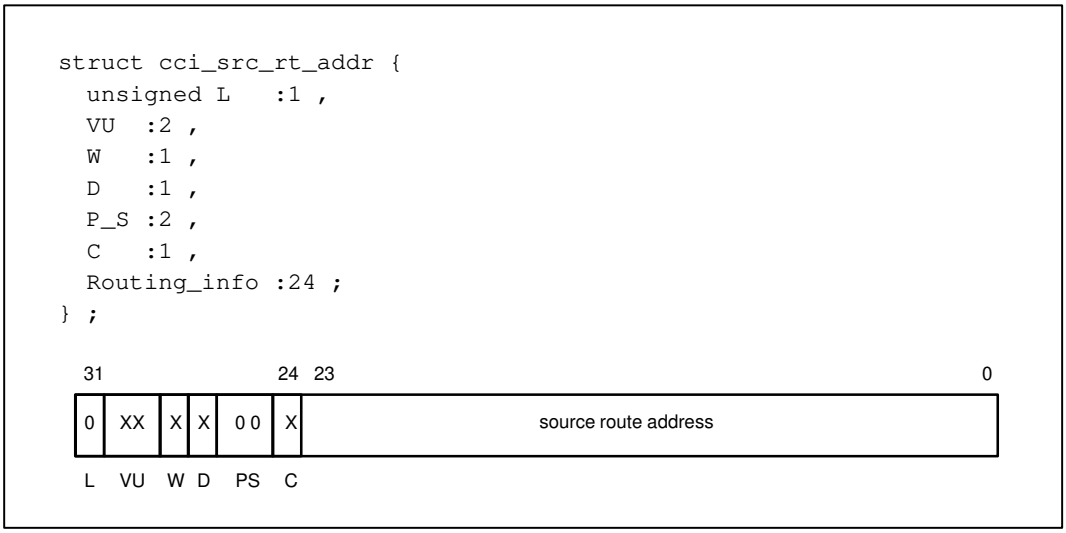


Figure 8. ifield address structure for source route addresses.

Source route addressing creates a point-to-point connection between source and destination end-points in a multi-switch network. It does this by explicitly steering a packet to the appropriate output of each switch it passes through.

Source route addressing also automatically creates *return-path* switch control information while the packet is in transit from the source end-point to the destination end-point. It does this by recording in the `ifield` the port by which the packet enters each switch. By the time the `ifield` reaches the destination end-point, it contains a complete description of the path back to the source.

This means a reply can be sent to the previous source without having to construct its address. Simply invert the direction bit (27) to reverse the source and destination end-point roles and the `ifield` is ready to be retransmitted.

Copyright © 1994 Thinking Machines Corporation

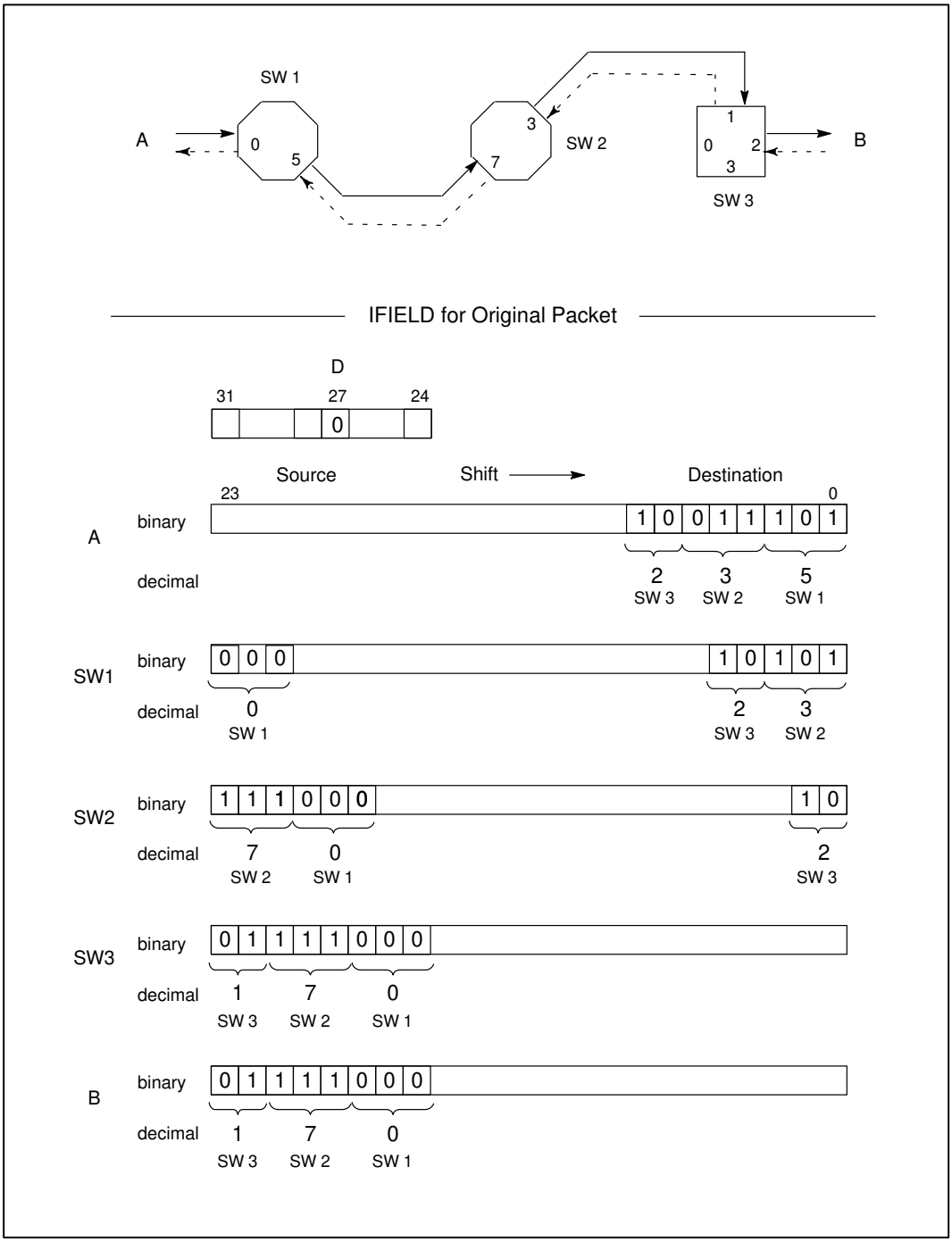


Figure 9. Source route addressing example — ifield changes along original path.

Copyright © 1994 Thinking Machines Corporation

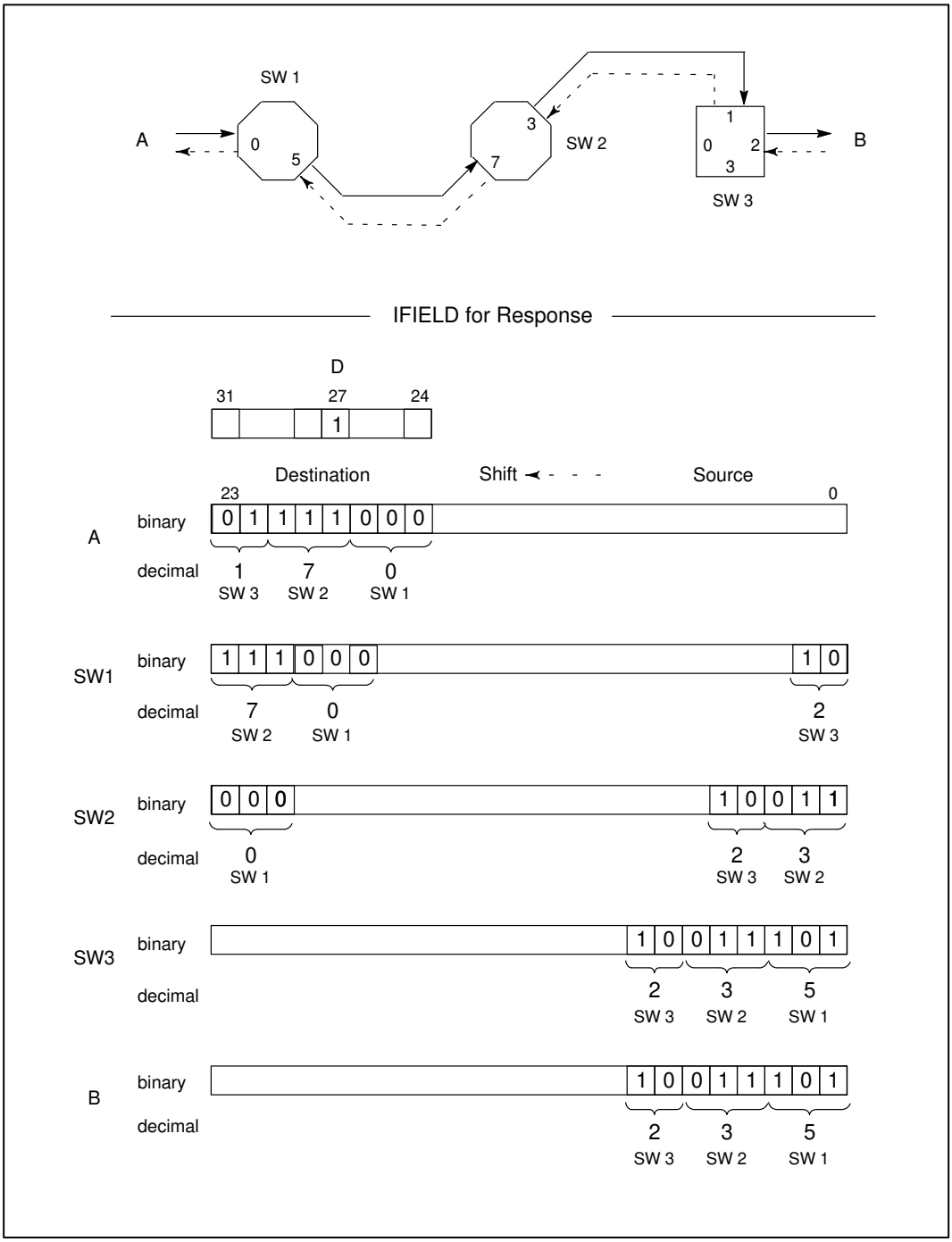


Figure 10. Source route addressing example — ifield changes along response path..

Copyright © 1994 Thinking Machines Corporation

2.6.2 Logical Addressing

In logical addressing, the 24-bit address field is divided into two 12-bit fields, one specifying the destination end-point and the other the source end-point. The sense of the direction bit (27) determines which half contains source information and which contains the destination. See Figure 11.

The following logical address ranges are reserved. Any addresses outside these ranges are allowed.

1111 110x xxxx reserved for local use

and

1111 111x xxxx globally assigned

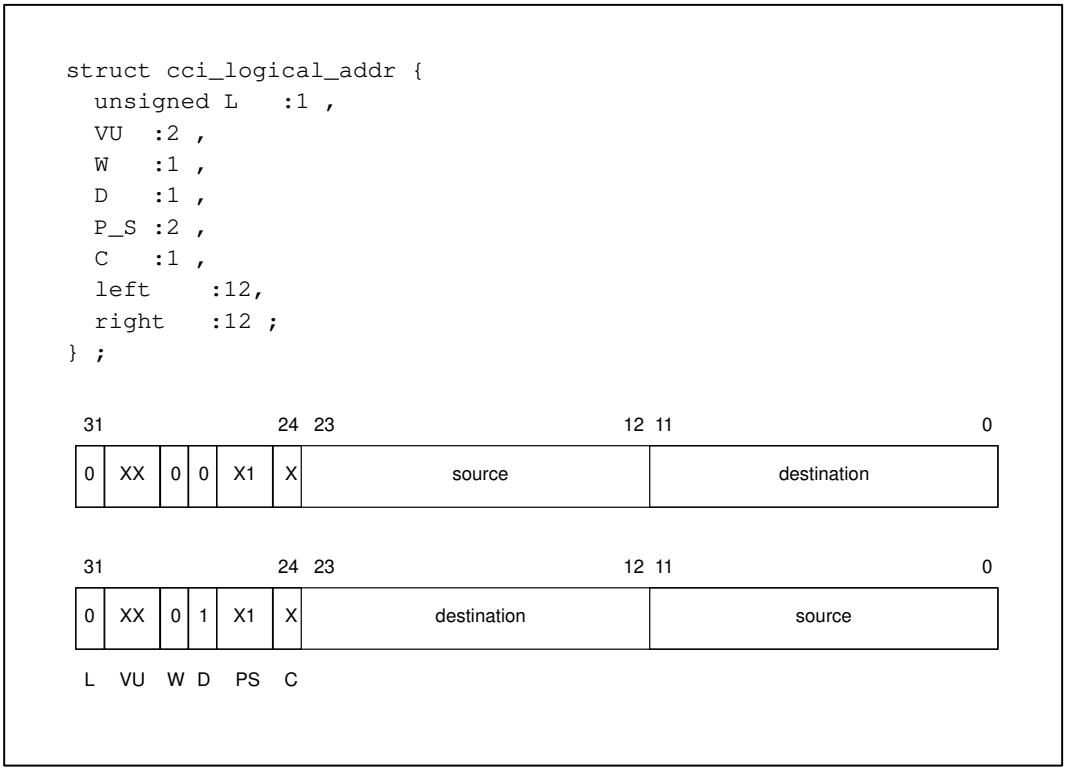


Figure 11. ifield address structure for logical addresses.

Copyright © 1994 Thinking Machines Corporation

2.7 Wildcards

An address structure's wildcard field contains three flag bits, each controlling a particular wildcard. The three wildcard flags are defined in `<sys/nethippi/hippi.h>` as follows.

Flag Name	Description
`HIPPI_LOCAL_ANY`	Enables the local socket to receive all messages sent from a given remote address.
`HIPPI_REMOTE_ANY`	Enables the local socket to receive all messages sent to the local address, regardless of the sender's address.
`HIPPI_ULP_ANY`	Enables the local socket to receive messages having any ULP ID. **NOTE:** HIPPI_ULP_ANY applies only to HIPPIPROTO_FP sockets; it cannot be used on HIPPIPROTO_PH sockets.

Setting a flag bit enables the corresponding wildcard attribute. The default for each flag is the not-set state.

Flag bits can be set in individually, or in combinations. The only combinations supported are:

PH Sockets

`HIPPI_LOCAL_ANY` • `HIPPI_REMOTE_ANY`

FP Sockets

`HIPPI_LOCAL_ANY` • `HIPPI_REMOTE_ANY` • `HIPPI_ULP_ANY`

Because of the dynamic, switch-specific nature of source route addressing, the HIPPI_LOCAL_ANY and HIPPI_REMOTE_ANY wildcards are irrelevant. The only wildcard of any use during FP transfers with source route addressing is HIPPI_ULP_ANY.

2.8 Connecting Sockets for `write`, `writev`, and `send`

Some packet transfer operations, namely **`write()`**, **`writev()`**, **`send()`**, and their **`CM_xxxx`** equivalents, require the source socket to be connected to the

Copyright © 1994 Thinking Machines Corporation

destination socket before the packet is transmitted. This is done with a **connect()** system call issued on the source host.

connect() establishes a point-to-point connnection between the local and remote socket addresses specified in the argument list. Use of the **connect()** system call is described in Section 3.5.

2.9 Reading from and Writing to the HIPPI Network

All of the standard UNIX read and write calls, as well as their corresponding parallel (**CM_xxxx**) read and write calls are supported. These are:

`read()`	`CM_read()`
`readv()`	`CM_readv()`
`recv()`	`CM_recv()`
`recvmsg()`	`CM_recvmsg()`
`recvfrom()`	`CM_recvfrom()`
`write()`	`CM_write()`
`writev()`	`CM_writev()`
`send()`	`CM_send()`
`sendmsg()`	`CM_sendmsg()`
`sendto()`	`CM_sendto()`

NOTE: From this point on, wherever a standard UNIX call is referenced, its **CM_xxxx** counterpart is implied.

recvmsg and **sendmsg** calls use a message header structure to convey needed address information. **recvfrom** and **sendto** calls communicate address information via a `sockaddr_hippi` structure. This structure mechanism is used to minimize the number of parameters that need to be supplied via command arguments.

This header structure is essentially the same for both serial and parallel calls; the only difference between them is in the names used for the attribute fields. Figures 12 and 13 illustrate the two sets of names used for serial and parallel message header structures in the AF_HIPPI domain.

Copyright © 1994 Thinking Machines Corporation

```
struct msghdr {
  caddr_t       msg_name;          /* buffer address */
  int           msg_namlen;        /* size of buffer */
  struct iovec  *msg_iov;          /* scatter/gather array */
  int           msg_iovlen;        /* # of elements in msg_iov */
  caddr_t       msg_accrights;     /* access rights sent/received */
  int           msg_accrightslen;  /* size of rights buffer */
} ;

structiovec {
  caddr_t       iov_base;
  int           iov_len;
};
```

Figure 12. AF_HIPPI message header structure — serial I/O

```
struct CM_msghdr {
  caddr_t       cm_msgname;        /* buffer address */
  int           cm_msgnamelen;     /* size of buffer */
  struct iovec  *cm_msgiov;        /* scatter/gather array */
  int           cm_msgiovcnt;      /* # of elements in msg_iov */
  caddr_t       cm_aux;            /* pointer to D1_area */
  int           cm_auxlen;         /* size of D1_area */
} ;

struct cmiov {

/* Required parts - must be supplied by calling program */
  cmaddr_t      cmv_addr;
  size_t        cmv_len;           /* # of bytes starting at cmv_addr */

/* Default parts - filled in by kernel at syscall interface */
  u_short       cmv_flags;         /* see below */
  u_short       cmv_vlen;          /*  sizeof this cmiov */
  offset_t      cmv_foff;          /* file position for this buffer */
  u_int         cmv_eltsize;       /* consecutive bytes per PN in buffer */
  u_short       cmv_npn;           /* number of PNs in whole– or sub–partition */
  u_short       cmv_partbase;      /* first PN in whole– or sub–partition */
};
```

Figure 13. AF_HIPPI message header structure — parallel I/O

Copyright © 1994 Thinking Machines Corporation

The message header fields used for **serial I/O** are described below.

`msg_name`
: For **`recvmsg`** calls, this points to a buffer that will receive **`struct sockaddr_hippi`**; for FP packets the buffer will receive a **`struct hippi_fp_header`** as well.

  For **`sendmsg`** and **`sendto`** calls, **`msg_name`** points to a buffer containing **`struct sockaddr_hippi`**.

`msg_namelen`
: For **`recvmsg`** calls, this specifies the size of **`msg_name`**, which must be 16 bytes for PH packets and 24 bytes for FP packets.

  For **`sendmsg`** calls, **`msg_namelen`** is 16 bytes, the size of **`struct sockaddr_hippi`**.

`*msg_iov`
: For **`recvmsg`** calls, this specifies the scatter/gather locations that will receive the data contained in the PH packet's Data section or FP packet's D2_area.

  For **`sendmsg`** calls, **`*msg_iov`** specifies the array that will supply data to the PH packet's Data section or FP packet's D2_area.

`msg_iovlen`
: This specifies the number of elements in the **`*msg_iov`** array.

`msg_accrights`
: For **`recvmsg`** calls, this field points to a buffer that will receive the data contained in an FP packet's D1_area.

  For **`sendmsg`** calls, **`msg_accrights`** points to a buffer that will supply data to the FP packet's D1_area.

  The **`msg_accrights`** field is not used for PH packets.

NOTE: The msg_accrights field is used by UNIX file systems for buffer access control. Because access rights are not relevant to AF_HIPPI transactions, it is used instead for this buffer pointer.

Copyright © 1994 Thinking Machines Corporation

`msg_accrightslen` — This specifies the size of the buffer referenced by **`msg_accessrights`**. For **`recvmsg`** calls, it must be 1024 bytes, the maximum size of the D1_area. For sendmsg calls, it should be set to the actual size of **`msg_accrights`**.

The message header fields used for **parallel I/O** are described below.

`cm_msgname` — For **`recvmsg`** calls, this points to a buffer that will receive **`struct sockaddr_hippi`**; for FP packets the buffer will receive a **`struct hippi_fp_header`** as well.

For **`sendmsg`** calls, **`cm_msgname`** points to a containing **`struct sockaddr_hippi`**.

`cm_msgnamelen` — For **`recvmsg`** calls, this specifies the size of **`cm_msgname`**, which must be 16 bytes for PH packets and 24 bytes for FP packets.

For **`sendmsg`**, **`cm_msgnamelen`** is 16 bytes, the size of **`struct sockaddr_hippi`**.

`*cm_msgiov` — For **`recvmsg`** calls, this specifies the scatter/gather locations that will receive the data contained in the PH packet's Data section or FP packet's D2_area.

For **`sendmsg`** calls, **`*cm_msgiov`** specifies the array that will supply data to the PH packet's Data section or FP packet's D2_area.

`cm_msgiovcnt` — This specifies the number of elements in the **`*cm_msgiov`** array.

`cm_aux` — For **`recvmsg`** calls, this field points to a buffer that will receive the data contained in an FP packet's D1_area.

For **`sendmsg`** calls, **`cm_aux`** points to a buffer that will supply data to the FP packet's D1_area.

The **`cm_aux`** field is not used for PH packets.

Copyright © 1994 Thinking Machines Corporation

`cm_auxlen` This specifies the size of the buffer referenced by **`cm_aux`** For **`recvmsg`** calls, it must be 1024 bytes, the maximum size of the D1_area. For sendmsg calls, it should be set to the actual size of **`cm_aux`**.

2.9.1 Reading from HIPPI

Table Table 1 lists the read calls and identifies the **`bind`** and **`connect`** requirements for each.

Table Table 1. CM5-HIPPI read calls — bind and connect requirements.

Read Call	socket	bind	connect
read	●	●	○
readv	●	●	○
recv	●	●	○
recvmsg	●	●	○
recvfrom	●	●	○
CM_read	●	●	○
CM_readv	●	●	○
CM_recv	●	●	○
CM_recvmsg	●	●	○
CM_recvfrom	●	●	○

● = required
○ = optional

The various read calls treat incoming packets differently, depending in part on which read call is issued and in part on the type of packet, PH or FP. These differences are described in Sections 2.9.2 and 2.9.3. Tables Table 2 and Table 3 summarize these differences.

Copyright © 1994 Thinking Machines Corporation

Table Table 2. PH Packet packet processing differences.

Read Call	Packet Elements Returned
read	Data
readv	Data
recv	Data
recvmsg	sockaddr_hippi and Data
recvfrom	sockaddr_hippi and Data
CM_read	Data
CM_readv	Data
CM_recv	Data
CM_recvmsg	sockaddr_hippi and Data
CM_recvfrom	sockaddr_hippi and Data

Table Table 3. FP Packet packet processing differences.

Read Call	Packet Elements Returned
read	D2
readv	D2
recv	D2
recvmsg	sockaddr_hippi, hippi_fp_header, D1, and D2
recvfrom	sockaddr_hippi, hippi_fp_header, and D2
CM_read	D2
CM_readv	D2
CM_recv	D2
CM_recvmsg	sockaddr_hippi, hippi_fp_header, D1, and D2
CM_recvfrom	sockaddr_hippi, hippi_fp_header, and D2

Copyright © 1994 Thinking Machines Corporation

2.9.2 Reading PH Packets

The following describes the differences in buffer allocation required for reading PH packets.

`read`, `CM_read`	Allocate buffer space for the packet's data section only.
`readv`, `CM_readv`	Allocate buffer space for the packet's data section only.
`recv`, `CM_recv`	Allocate buffer space for the packet's data section.
`recvmsg`, `CM_recvmsg`	Allocate buffer space for the contents of **`sockaddr_hippi`** and for the packet's data section.
`recvfrom`, `CM_recvfrom`	Allocate buffer space for the contents of **`sockaddr_hippi`** and for the packet's data section.

2.9.3 Reading FP Packets

The following describes the differences in buffer allocation required for reading FP packets.

`read`, `CM_read`	Allocate buffer space for the D2_area.
`readv`, `CM_readv`	Allocate buffer space for the D2_area.
`recv`, `CM_recv`	Allocate buffer space for the D2_area.
`recvmsg`, `CM_recvmsg`	Allocate buffer space for the contents of **`sockaddr_hippi`**, **`hippi_fp_header`** and for the D1 and D2 areas. **NOTE:** Allocate 1016 bytes for D1.
`recvfrom`, `CM_recvfrom`	Allocate buffer space for the contents of **`sockaddr_hippi`**, **`hippi_fp_header`** and the D2 area.

Copyright © 1994 Thinking Machines Corporation

2.9.4 Writing to HIPPI

Table Table 4 lists these calls and identifies the **`bind`** and **`connect`** requirements for each.

Table Table 4. CM5-HIPPI write calls — bind and connect requirements.

Read Call	socket	bind	connect
write	●	○	●
writev	●	○	●
send	●	○	●
sendmsg	●	○	○
sendto	●	○	○
CM_write	●	○	●
CM_writev	●	○	●
CM_send	●	○	●
CM_sendmsg	●	○	○
CM_sendto	●	○	○

● = required
○ = optional

Sections 2.9.5 and 2.9.6 describe the differences in how PH and FP packets are handled by the various write calls. Tables Table 5 and Table 6 summarize these differences.

Copyright © 1994 Thinking Machines Corporation

Table Table 5. PH Packet packet transmittal differences.

Read Call	Packet Elements Transmitted
write	Data
writev	Data
send	Data
sendmsg	sockaddr_hippi and Data
sendto	sockaddr_hippi and Data
CM_write	Data
CM_writev	Data
CM_send	Data
CM_sendmsg	sockaddr_hippi and Data
CM_sendto	sockaddr_hippi and Data

Table Table 6. FP Packet packet transmittal differences.

Read Call	Packet Elements Transmitted
write	D2
writev	D2
send	D2
sendmsg	sockaddr_hippi, D1, and D2
sendto	sockaddr_hippi, and D2
CM_write	D2
CM_writev	D2
CM_send	D2
CM_sendmsg	sockaddr_hippi, D1, and D2
CM_sendto	sockaddr_hippi, and D2

Copyright © 1994 Thinking Machines Corporation

2.9.5 Writing PH Packets

The various write calls handle PH packets as follows.

`write`, `CM_write`	Transmit the data section only.
`writev`, `CM_writev`	Transmit the data section only.
`send`, `CM_send`	Transmit the data section only.
`sendmsg`, `CM_sendmsg`	Transmit **`sockaddr_hippi`** and the packet's data section.
`sendto`, `CM_sendto`	Transmit **`sockaddr_hippi`** and the packet's data section.

2.9.6 Writing FP Packets

The various write calls handle FP packets as follows.

`write`, `CM_write`	Transmit an FP header constructed in accordance with the **`sockaddr_hippi`** that was used during the previous connect followed by the D2_area.
`writev`, `CM_writev`	Transmit an FP header constructed in accordance with the **`sockaddr_hippi`** that was used during the previous connect followed by the D2_area.
`write`, `CM_write`	Transmit an FP header constructed in accordance with the **`sockaddr_hippi`** that was used during the previous connect followed by the D2_area.
`sendmsg`, `CM_sendmsg`	Transmit an FP header constructed in accordance with the **`sockaddr_hippi`**, that was used during the previous connect, followed by the D1_area. and the D2_area.

Copyright © 1994 Thinking Machines Corporation

`sendto`, `CM_sendto`	Transmit an FP header constructed in accordance with the **`sockaddr_hippi`**, that was used during the previous connect, followed by the D2_area.

The **`sendmsg`** and **`CM_sendmsg`** calls use **`struct msghdr`** and **`struct CM_msghdr`** to simplify the parameter input requirements. Figures 12 and 13 explain **`msghdr`** and **`CM_msghdr`** usage for write operations. **`struct msghdr`** is defined in **`<sys/socket.h>`**. **`struct CM_msghdr`** is defined in **`<cmsys/cmio.h>`**.

Copyright © 1994 Thinking Machines Corporation