Chapter 3

# CM5-HIPPI Socket Programming

This chapter explains how to use sockets to exchange datagram packets between local and remote processes through the CM5-HIPPI. The description is organized into six main themes:

- Header files — Section 3.1
- Creating a socket — Section 3.2
- Initializing the address structure — Section 3.3
- Binding a socket — Section 3.4
- Connecting a socket — Section 3.5
- Socket options — Section 3.6

For information about programming sockets for stream communication or for other general information about socket programming, refer to UNIX networking documentation such as the Sun Microsystems *Network Programming Guide*.


*Copyright © 1994 Thinking Machines Corporation*

## 3.1 Header Files

The following files should be included to support the various system calls.

| | |
|---|---|
| `<sys/types.h>` | basic system type definitions |
| `<cmsys/cm_types.h>` | parallel system type definitions |
| `<sys/socket.h>` | socket definitions |
| `<nethippi/hippi.h>` | AF_HIPPI socket definitions |

## 3.2 Creating a Socket

The first step in any socket-based communication is socket creation. An application creates a socket with the system call, **`socket()`** as follows.

```
s = socket(domain, type, protocol)
```

| | |
|---|---|
| `domain` | Specify `AF_HIPPI` to set up a socket for datagram communication. |
| `type` | Within the `AF_HIPPI` domain, specify either `SOCK_RAW` for a PH-layer socket or `SOCK_DGRAM` for an FP-layer socket. |
| `protocol` | Currently, a single protocol is available for each socket type. Consequently, this entry can be **`0`** the default protocol value. If you choose to specify the protocol by name use either `HIPPIPROTO_PH` or `HIPPIPROTO_FP`, as applicable. |

The two `AF_HIPPI` socket call possibilities are:

| | |
|---|---|
| `s = socket(AF_HIPPI, SOCK_RAW, 0)` | PH sockets |
| `s = socket(AF_HIPPI, SOCK_DGRAM, 0)` | FP socket |



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

Once a socket has been created, a number of operations can be performed on it:

- **setsockopt** and **getsockopt** can be used to manipulate options affecting socket behavior.
- **sendto** and **sendmsg** can be used to transmit serial messages to another socket. The **bind** and **connect** system calls are not required, as **sendto** or **sendmsg** provide the needed address information in their argument lists.
- **CM_sendto** and **CM_sendmsg** are parallel versions of **sendto** and **sendmsg**.

## 3.3 Initializing the Address Structure

Under CMOST 7.3 Beta, the user is responsible for initializing a socket's address structure before issuing a **bind()** or **connect()** system call for that socket.

This section describes initialization of the AF_HIPPI address structure, `struct sockaddr_hippi`. Refer tothe appropriate UNIX network documentation, such as the Sun Microsystems *Network Programming Guide* for information about AF_UNIX and AF_INET address structures.

Section 3.3.1 describes the `struct sockaddr_hippi` attributes that must be specified for a HIPPIPROTO_PH protocol socket. Section 3.3.2 provides the same information for sockets using HIPPIPROTO_FP protocol.

### 3.3.1 Initializing Address Structures for PH Sockets

The following components of `struct sockaddr_hippi` must be initialized for a HIPPIPROTO_PH socket before a **bind()** or **connect()** system call is issued for that socket.

`family` Always specify AF_HIPPI.


Copyright © 1994 Thinking Machines Corporation

`if_unit` — This is the number that the network administrator assigns to the CM5-HIPPI interface you plan to use. If D you do not know this number, ask your network administrator.

`ifield` — This is a 32-bit hexadecimal value that controls packet routing between the source and destination sockets at the physical level. Some portion bits 0—23 contain socket routing information. Bits 24—31 contain either control information or are not used. The exact implementation of the address field depends on the address mode to be used as well as the network topology. See Section 2.6 for a fuller discussion of `ifield`s.

`wildcards` — Wildcards can be specified for either the local (receiving) socket or the remote (sending) socket, but not both. See Section 2.7 for wildcard rules.

#### HIPPIPROTO_PH EXAMPLE 1

```
bzero(&hippi_sa, sizeof (hippi_sa);
hippi_sa.family = AF_HIPPI;
hippi_sa.if_unit = 0;
hippi_sa.ifield.c_ifield = 0x0203A142;
hippi_sa.wildcards=HIPPI_REMOTE_ANY;
```

This example initializes the address structure for interface unit 0. The `ifield`'s two high-order nibbles (0x02) specify logical addressing (PS bits = x1). Because D = 0, the logical destination address is contained in the three least signifcant nibbles and the source address in the next three higher order nibbles — in this example, destination = 0x#####142 and source = 0x##03A###. This example also uses the wildcard field to enable the socket to accept messages addressed to it that are sent from any remote address.

#### HIPPIPROTO_PH EXAMPLE 2

```
bzero(&hippi_sa, sizeof (hippi_sa);
hippi_sa.family = AF_HIPPI;
hippi_sa.if_unit = 0;
hippi_sa.ifield.c_ifield = 0x00000164;
```




Copyright © 1994 Thinking Machines Corporation

The second example initializes the address structure for interface unit 1 and specifies source route addressing. In this example, the `ifield`'s PS bits = 00 and the direction bit = 0. With D = 0, the path to the destination socket is described by the three least significant nibbles (0x#####164). Wildcards are not available for source route addressing of PH sockets.

### 3.3.2 Initializing Address Structures for FP Sockets

The same set of fields is initialized for HIPPIPROTO_FP sockets as for HIPPIPROTO_PH sockets, plus a ULP ID field.

To initialize the upper layer protocol for an FP socket, enter the following.

```
hippi_sa.hs_ulp = ulp_id;
```

where *ulp_id* is the value assigned to the upper layer protocol of interst.

On HIPPIPROTO_FP sockets, wildcards can also be applied to the ulp field for the local (receiving) socket. Wildcards cannot be applied to the upper layer protocol of a remote (sending) socket. See Section 2.7 for a description of wildcard use on ULP IDs.

#### HIPPIPROTO_FP EXAMPLE 1

```
bzero(&hippi_sa, sizeof (hippi_sa);
hippi_sa.family = AF_HIPPI;
hippi_sa.if_unit = 0;
hippi_sa.ulp = 0x82;
hippi_sa.ifield.c_ifield = 0x0203A142;
```

This example initializes an address structure for interface unit 0 and specifies a ULP ID of 0x82 residing at logical address `0x03A`.

#### HIPPIPROTO_FP EXAMPLE 2

```
bzero(&hippi_sa, sizeof (hippi_sa);
hippi_sa.family = AF_HIPPI;
hippi_sa.if_unit = 1;
hippi_sa.ifield.c_ifield = 0x00000164;
hippi_sa.wildcards = HIPPI_ULP_ANY;
```

The second FP example initializes an address structure for interface unit 1, specifies a source route address of `0x#####164`, and uses a wildcard to


Copyright © 1994 Thinking Machines Corporation

enable the socket to accept messages sent to the local addresss from a given remote address but having any ULP ID.

#### HIPPIPROTO_FP EXAMPLE 3

```
bzero(&hippi_sa, sizeof (hippi_sa);
hippi_sa.family = AF_HIPPI;
hippi_sa.if_unit = 1;
hippi_sa.ifield.c_ifield = 0x0203A142;
hippi_sa.wildcards = HIPPI_ULP_ANY;
hippi_sa.wildcards = HIPPI_REMOTE_ANY;
```

This example specifies interface unit 1, specifies a logical address and uses wildcards for the remote address and for the ULP ID. This enables the socket to accept all messages sent to the local addresss from any remote address having any ULP ID.

## 3.4 Binding a Socket

Socket binding is done with the system call, **`bind()`**, which has the following syntax.

```
bind(s, *name, namelen)
```

| | |
|---|---|
| s | This is the socket descriptor returned by a successful **`socket()`** call. |
| *name | This is a pointer to the address structure being bound to socket *s*. |
| namelen | This is the size of structure *name* in bytes. Under CMOST 7.3 Beta, this value is 16. |

Two examples are provided below showing HIPPIPROTO_PF and HIPPI-PROTO_FP sockets being created, initialized, and then bound.

#### HIPPIPROTO_PH EXAMPLE 1

The first example creates a PH socket, `sock`, initializes its address structure, and binds it.




Copyright © 1994 Thinking Machines Corporation

```
sock = socket(AF_HIPPI, SOCK_RAW, 0)
bzero(&hippi_sa, sizeof (hippi_sa);
hippi_sa.family = AF_HIPPI
hippi_sa.if_unit = 0
hippi_sa.ifield.c_ifield = 0x0203A142
bind(sock, (struct sockaddr *)&hippi_sa, 16)
```

#### HIPPIPROTO_FP EXAMPLE 2

The second example creates an FP socket, `sock`, initializes its address structure, and binds it. **NOTE:** In this example, the FP socket does not specify a ULP ID.

```
sock = socket(AF_HIPPI, SOCK_DGRAM, 0)
bzero(&hippi_sa, sizeof (hippi_sa);
hippi_sa.family = AF_HIPPI
hippi_sa.if_unit = 0
hippi_sa.ifield.c_ifield = 0x0203A142
bind(sock, (struct sockaddr *)&hippi_sa, 16)
```

## 3.5 Connecting a Socket

For a socket to use **`write()`**, **`writev()`**, or **`send()`**, it must be *connected* to the socket that will receive the packets. This is done with a **`connect()`** system call, which has the following syntax.

```
connect(s, *name, namelen)
```

| | |
|---|---|
| s | This is the socket descriptor returned by a successful **`socket()`** call. |
| *name | This is a pointer to the address structure being to be connected to socket *s*. |
| namelen | This is the size of structure *name* in bytes. Under CMOST 7.3 Beta, this value is 16. |

Two examples are provided below showing HIPPIPROTO_PF and HIPPIPROTO_FP sockets being created, initialized, and then connected.


Copyright © 1994 Thinking Machines Corporation

#### HIPPIPROTO_PH EXAMPLE 1

The first example creates a PH socket, sock1, initializes its address structure, and connects it to sock2.

```
sock1 = socket(AF_HIPPI, SOCK_RAW, 0)
bzero(&hippi_sa, sizeof (hippi_sa);
hippi_sa.family = AF_HIPPI
hippi_sa.if_unit = 0
hippi_sa.ifield.c_ifield = 0x0203A142
connect(sock2, (struct sockaddr *)&hippi_sa, 16)
```

#### HIPPIPROTO_FP EXAMPLE 2

The second example creates an FP socket, sock1, initializes its address structure, and connects it to sock2.

```
sock1 = socket(AF_HIPPI, SOCK_DGRAM, 0)
bzero(&hippi_sa, sizeof (hippi_sa);
hippi_sa.family = AF_HIPPI
hippi_sa.if_unit = 0
hippi_sa.ifield.c_ifield = 0x0203A142
connect(sock2, (struct sockaddr *)&hippi_sa, 16)
```


Copyright © 1994 Thinking Machines Corporation

## 3.6 Socket Options

All of the standard UNIX socket options are supported in AF_HIPPI, with the following modifications and additions.

| | |
|---|---|
| `SO_SNDBUF`<br>`SO_RCVBUF` | These options specify the send and receive queue buffer sizes for serial transfers only. |
| `SO_PAR_SNDBUF`<br>`SO_PAR_RCVBUF` | These options have been added to support parallel socket-based I/O. They enable the user to specify larger buffer sizes than are suported by by their serial counterparts. |
| `HIPPI_DONT_BLOCK_ON_WRITE` | This option allows writes to return without waiting for completion status. By default, writes will block until full completion status is returned. |
| `HIPPI_MAX_PACKET_SIZE` | This option is used for ”autofragmentation”, which instructs the protocol layer in the kernel to break the packet into smaller packets of the size specified by this sockopt. In all cases, each smaller packet will use the same ifield that was provided with the write call. For FP transfers that included a D1 dataset, each packet will contain the same D1 dataset that was provided with that write call.<br>**NOTE:** This behavior applies to all forms of write calls (write, send, sendto, sendmsg and their CM_xxx counterparts). |


Copyright © 1994 Thinking Machines Corporation