NetIPC 3000/XL
Programmer's Reference Manual

HP 3000 MPE/iX Computer Systems

Edition 3

Manufacturing Part Number: 5958-8600
E1089

U.S.A. October 1989

Notice

The information contained in this document is subject to change without notice.

Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability or fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or for direct, indirect, special, incidental or consequential damages in connection with the furnishing or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

Restricted Rights Legend

Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013. Rights for non-DOD U.S. Government Departments and Agencies are as set forth in FAR 52.227-19 (c) (1,2).

Acknowledgments

MS-DOS is a U.S. registered trademark of Microsoft Corp.

Hewlett-Packard Company
3000 Hanover Street
Palo Alto, CA 94304 U.S.A.

Contents

Contents

Contents

Contents

Contents

Figures

Figures

Tables

Tables

Preface

Network InterProcess Communication (NetIPC) is a set of programmatic calls that can be used to exchange data between peer-to-peer processes on the same or different nodes in an Hewlett-Packard NS network. Any process can initiate communication, and any process can send or receive messages by means of common intrinsics. NetIPC 3000/XL is a version of NetIPC that can be used in programs written for MPE XL based computer systems.

NetIPC provides programmatic access to the Transmission Control Protocol (TCP), which is the Transport-Layer protocol used by NS 3000/XL link products.

NetIPC 3000/XL is provided with the purchase of any NS 3000/XL link product. With the purchase of the X.25 XL System Access link product, NetIPC access to TCP (level 4) and X.25 (level 3) is provided.

Intended Audience of this Manual

In order for you to use NetIPC, you should be familiar with MPE XL, the operating system on which NetIPC 3000/XL can be used. You should also be familiar with the TCP protocol and a high-level language such as Pascal.

If you are using direct access to level 3 (X.25), you should be familiar with the X.25 protocol and the HP 3000 products that provide X.25 network communication.

Organization of the Manual

Following is a summary of how this manual is organized:

- Chapter 1 , “NetIPC Fundamentals,” explains how to establish connections, send and receive data over connections, and shutdown connections between processes using NetIPC TCP access or X.25 level 3. This chapter also introduces some of the NetIPC calls.
- Chapter 2 , “Cross-System NetIPC,” describes what NetIPC calls need to be considered for a cross-system application (using TCP access) between an HP 3000 Series 900 and either an HP 1000, HP 9000, or personal computer.
- Chapter 3 , “NetIPC Intrinsics,” provides a detailed description of each NetIPC intrinsic, in alphabetical order. This chapter also explains programming considerations, syntax, and the structure and function of several parameters that are common to multiple NetIPC intrinsics.
- Chapter 4 , “NetIPC Examples,” provides sample programs using NetIPC intrinsics for peer-to-peer process communication for both TCP access and X.25 level 3 access.

- Appendix A , “IPC Interpreter (IPCINT),” describes how to use the IPCINT software utility which provides an interactive interface to the NetIPC intrinsics used for programmatic access to X.25 level 3.
- Appendix B , “Cause and Diagnostic Codes,” lists the possible cause and diagnostic codes generated by NS X.25 packets.
- Appendix C , “Error Messages,” includes a list of SOCKERRs and the corresponding protocol module errors returned in the IPCCHECK intrinsic, and provides a table of NetIPC errors (SOCKERRs) returned in the result parameter of the NetIPC intrinsics.
- Appendix D , “Migration From PTOP to NetIPC and RPM,” explains how to translate programs written in the Program-to-Program communication service to NetIPC and RPM.
- Appendix E , “C Program Language Considerations,” describes program language differences that affect how NetIPC intrinsics are used in programs written in C programming language.

Related Publications

The following publications contain additional information that can assist you in using NetIPC.

NS 3000/XL

- *Using NS 3000/XL Network Services Manual*
- *NS 3000/XL Error Messages Reference Manual*
- *NS 3000/XL Configuration, Planning and Design Guide*
- *NS 3000/XL NMMGR Screens Reference Manual*
- *NS Cross-System NFT Reference Manual*
- *NS Cross-System Network Manager Reference Manual*

X.25 Networking

- *X.25 XL System Access Configuration Guide*
- *Using the OpenView DTC Manager*

MPE XL

- *MPE XL Commands Reference Manual*
- *MPE XL Intrinsics Reference Manual*
- *HP PASCAL Reference Manual*
- *HP COBOL II/XL Reference Manual*
- *FORTRAN 77/XL Reference Manual Supplement*

1 NetIPC Fundamentals

Network Interprocess Communication (NetIPC) is a facility that enables processes on the same or different nodes to communicate with each other using a series of programmatic calls.

NetIPC 3000/XL can be purchased as part of any NS 3000/XL link product. It provides access to the Transmission Control Protocol (TCP), the Transport Layer protocol used in NS 3000/XL link products. Over an NS X.25 network, NetIPC provides access to X.25 protocol features at level 3.

To communicate by means of NetIPC, processes must be executing concurrently. One or more users (or programs) can run these processes independently, or one process can initiate the execution of another by using the Remote Process Management (RPM) Network Service. In conjunction with NetIPC, RPM can be used to manage distributed applications. Refer to the *Using NS 3000/XL Network Services* manual for information about RPM.

Processes that use NetIPC calls gain access to the communication services provided by the network protocols of NS 3000/XL. NetIPC does not encompass a protocol of its own; rather, it acts as a generic interface to the protocols underlying the NS 3000/XL network services.

This chapter is organized into the following major sections:

- NetIPC Concepts
- Using NetIPC for Interprocess Communication
- Direct Access to Level 3, X.25
- Considerations for using NetIPC between MPE-V and MPE XL HP 3000 systems.

NetIPC Concepts

The following paragraphs describe the concept of sockets, and the NetIPC terms used to describe a NetIPC connection.

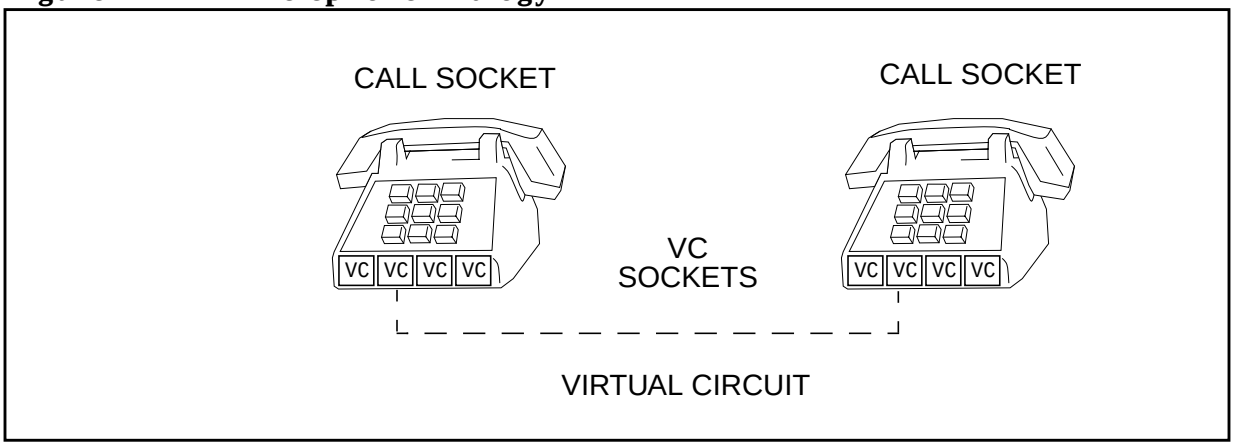
Sockets

NetIPC uses a data structure called a socket to create a connection to a NetIPC process on another system. Even though this process may reside upon the same node, the process that receives the NetIPC call is known as a remote system. The NetIPC calls are used to establish connections and manipulate sockets so that data can be exchanged with other processes. The Transmission Control Protocol (Transport Layer) (TCP) regulates the transmission of data to and from these data structures. When direct access to level 3 (X.25) is used, the X.25 protocol regulates the transmission of data between sockets. Although data must pass through the control of lower-level protocols, these details are transparent to NetIPC processes when they send and receive data.

Connections

Before a connection can be established between two NetIPC processes, each process must create a call socket. A call socket is a type of socket that is roughly analogous to a telephone handset with multiple buttons or extensions. NetIPC processes engage in a three-way handshake over the connection formed by their respective call sockets in order to create a virtual circuit (VC) socket at each process. A call socket can be thought of as one of the steps needed to build a VC socket.The VC sockets created by this dialogue are the endpoints of a new connection called a **virtual circuit** or **virtual circuit connection**. A call socket is analogous to a telephone with multiple extensions, and a VC socket is analogous to one of the telephone extensions as shown in Figure 1-1.

Figure 1-1 Telephone Analogy



Virtual circuits are the basis for interprocess communication. Once a virtual circuit is established, the two processes that created it may use it to exchange data. *Two processes pass data only through VC sockets, not through call sockets*. For example, a process may use one call socket to establish multiple VC sockets; these VC sockets are then used to communicate with different processes. A call socket may even be shut down once a virtual circuit connection is established without affecting communication between the processes. A virtual circuit has the following properties:

- It provides reliable service, guaranteeing that data will not be corrupted, lost, duplicated or received out of order.
- Sharing of connections is allowed for sends only. A process may allow up to 8 connections to be shared. There is no limit as to how many processes may send on a shared connection though only one at a time. Sharing a connection can only be done in Privileged Mode.

Naming, Socket Registry and Destinations

When a NetIPC process initiates a connection with a peer process, it must reference a call socket that was created by that peer process. In order to gain access to the call socket of another process, a NetIPC process must reference the socket name or the address of that call socket through `IPCDEST`.

NetIPC processes associate ASCII-coded names with the call sockets they create and insert this information into the socket registry of their node. Each NS 3000/XL node has a socket registry that contains a listing of all the named call sockets that reside at that node. In keeping with the telephone analogy begun earlier, the socket registry could be compared to a telephone directory: a call socket is associated with a name and inserted in the local socket registry in much the same way as a telephone number is associated with a person's name and placed in a local telephone directory.

NetIPC processes use the socket registry to access call sockets by passing a socket name and the corresponding node name to the socket registry software. The socket registry determines which socket is associated with the name specified and translates the address of that socket into a **destination descriptor** which it returns to the inquiring process.

A destination descriptor is a data structure which carries address information. Specifically, when a destination descriptor is returned to a process, it tells the process:

- how to get to the node where the referenced socket resides
- how to get to the referenced socket at that node. Using the socket registry to gain access to call socket of another process is similar to using directory assistance to find a person's phone number. The

resulting destination descriptor, like a telephone number, is then used to direct a caller to a particular destination.

Descriptors

NetIPC processes reference three types of descriptors: 1) call socket descriptors, 2) destination descriptors, and 3) virtual circuit socket descriptors. Descriptors are returned to processes when certain NetIPC calls are invoked (see Table 1-1). The following is an explanation of the descriptors, the NetIPC call, or calls, that are used to obtain them, and the terminology used to refer to these descriptors in the syntax and parameter statements:

- **Call Socket Descriptor**. A call socket descriptor describes a call socket. A process obtains a call socket descriptor by invoking `IPCCREATE` (to create a call socket) or `IPCGET` (to get a descriptor given away by another process). When a call socket descriptor is obtained with either method, the call socket is said to be **owned** by the calling process. The term *`calldesc`* refers to a call socket descriptor parameter.
- **Destination Descriptor**. A destination descriptor describes a destination socket. The descriptor points to addressing information that is used to direct requests to a specified call socket at a specified node. A process obtains a destination descriptor by invoking the command `IPCLOOKUP` (to look up the name of a call socket in a specific socket registry) or `IPCDEST` (if the address of the destination call socket is known). The term *`destdesc`* refers to a destination descriptor parameter.
- **VC Socket Descriptor**. A VC socket descriptor describes a VC socket. A VC socket is the endpoint of a virtual circuit connection between two processes. A VC socket descriptor is returned by `IPCCONNECT` and `IPCRECVCN` during the creation of a connection between two call sockets. A process can also obtain a VC socket descriptor given away by another process by invoking `IPCGET`. The term *`vcdesc`* refers to a VC socket descriptor parameter.

Table 1-1 Descriptor Summary

Type of Descriptor	Parameter Name	Description	Returned as Output From
Call socket descriptor	*calldesc*	Refers to a call socket. A call socket is used to build a VC socket.	IPCCREATE IPCGET
Destination descriptor	*destdesc*	Refers to a destination socket. A destination socket points to addressing information that is used to direct requests to a certain call socket at a certain node.	IPCLOOKUP IPCDEST
VC socket descriptor	*vcdesc*	Refers to a VC socket. A VC socket is the endpoint of a virtual circuit connection between two processes.	IPCCONNECT IPCRECVCN IPCGET

Using NetIPC for Interprocess Communication

The following paragraphs describe the tasks for using NetIPC for process-to-process communication in programs which are:

- Establish a connection.
- Send and receive data over the connection.
- Shut down the connection.

These discussions are based on access to level 4 (TCP) but most principles apply to direct access to level 3 (X.25). Information specific to X.25 is noted in the discussion.

After establishing a virtual circuit, you can use other NetIPC functions which are described in this chapter under the heading, “Additional NetIPC Functions”.

Establishing a Level 4 Connection

The following paragraphs are a call-by-call explanation of the dialogue through which a virtual circuit connection is built. This example uses the socket registry facility by establishing names for call sockets with IPCNAME and retrieving the names with IPCLOOKUP. Figure 1-9 shows the sequence of calls to use if the address of the socket is known (using IPCDEST).

Only two processes are shown in this example. Either or both of the processes shown can establish virtual circuit connections with other processes. Secondary or auxiliary connections can also be set up between the same two processes.

NOTE

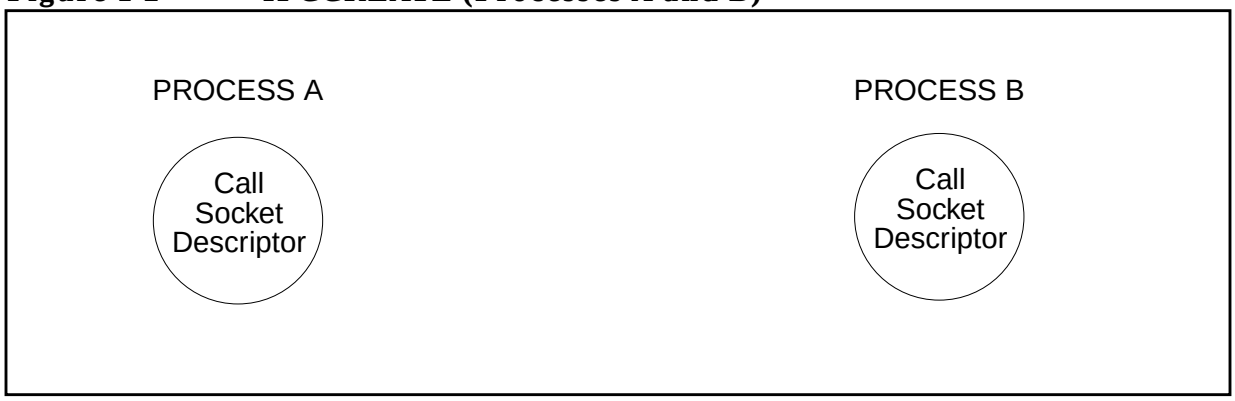
Both of the processes in the following dialogue are assumed to be created and running at their respective nodes. NetIPC does not include a call to schedule remote processes. Refer to the chapter, Remote Process Management, in the *Using NS 3000/XL Network Services* manual for more information about initializing remote processes with RPM.

1. Creating a Call Socket

Interprocess communication is initiated when Process A and Process B each create a call socket by invoking the NetIPC call IPCCREATE (see Figure 1-2). As explained previously, a call socket is roughly analogous to a telephone with multiple extensions (see Figure 1-1). IPCCREATE returns a **call socket descriptor** to the calling process in its *calldesc* parameter that describes the call socket, or “telephone extension,” that

the process has created. This call socket descriptor is then used in subsequent NetIPC calls.

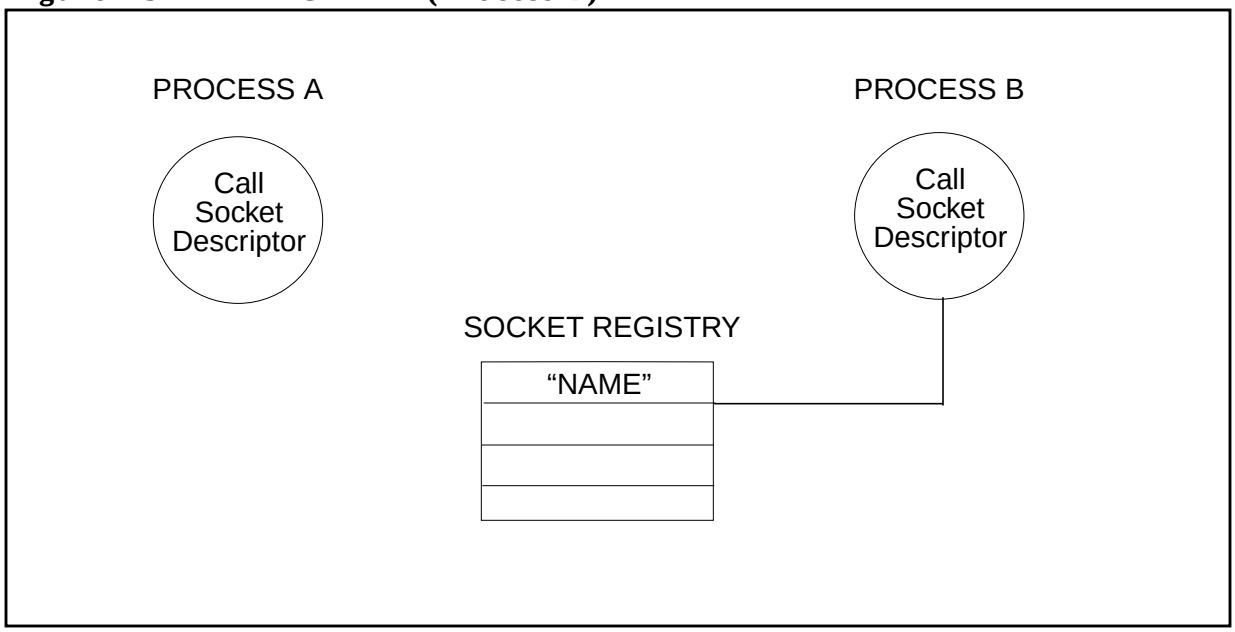
Figure 1-2 IPCCREATE (Processes A and B)



2. Naming a Call Socket

Process B associates a name with its call socket by calling `IPCNAME` (see Figure 1-3). When a call socket is named, this information is placed in the socket registry at the local node. The name Process B assigns to its call socket must also be known to Process A because Process A must reference it later in its `IPCLOOKUP` call. (When a socket name is known to both processes in this way, it is called a **well-known name**.) Although call sockets do not have to be named, a process cannot gain access to the socket of another process if the socket is not named (unless the address of that socket is known, in which case `IPCDEST` is used). The socket must be named and be recorded in the socket registry at the node of Process B when Process A calls `IPCLOOKUP`.

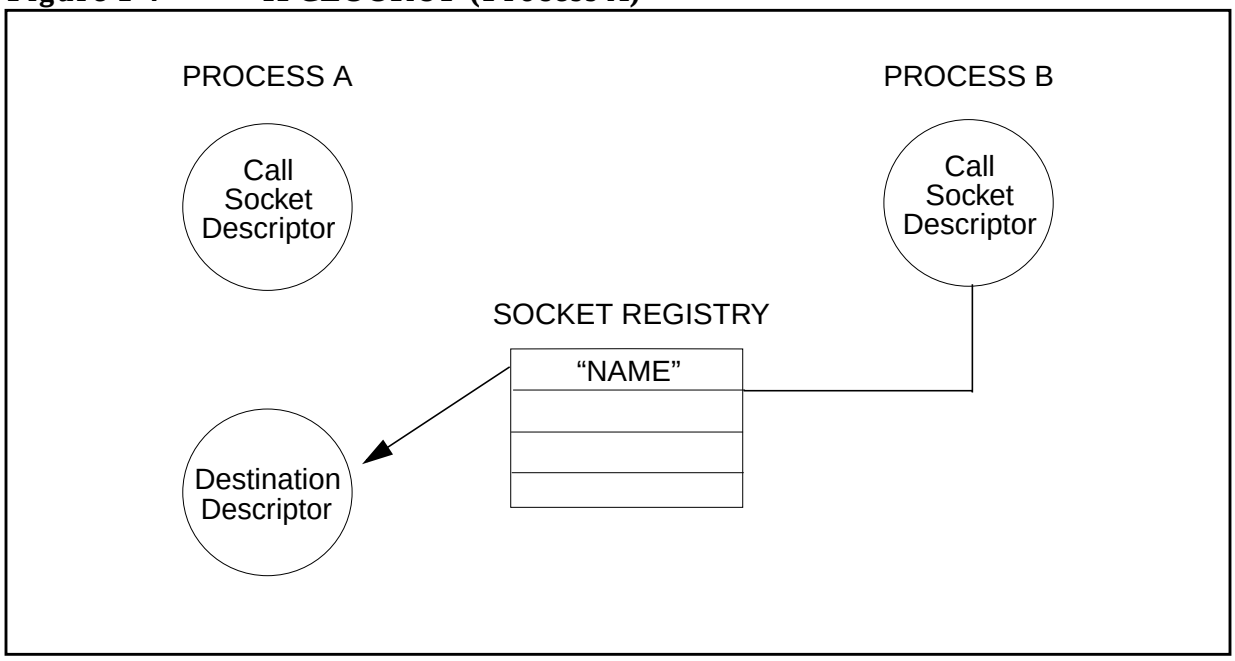
Figure 1-3 IPCNAME (Process B)



3. Looking Up a Call Socket Name

Process A must reference the call socket of Process B by its name in the call to IPCLOOKUP to "look up" the name of the call socket in the socket registry at the node where Process B resides. IPCLOOKUP returns a destination descriptor in its *destdesc* parameter (see Figure 1-4). The destination descriptor indicates the location of the destination call socket which is owned by Process B. IPCLOOKUP is similar to a telephone company's directory assistance service: Process A calls the "operator" (IPCLOOKUP), and gives him/her a "city" (*location* parameter) and a "name" (*socketname* parameter). Using the "city," that is, the node name or environment ID, the operator looks for the name in the proper "telephone directory" (socket registry). Once the name is found, the operator returns a "telephone number" (*destdesc* parameter) to the caller.

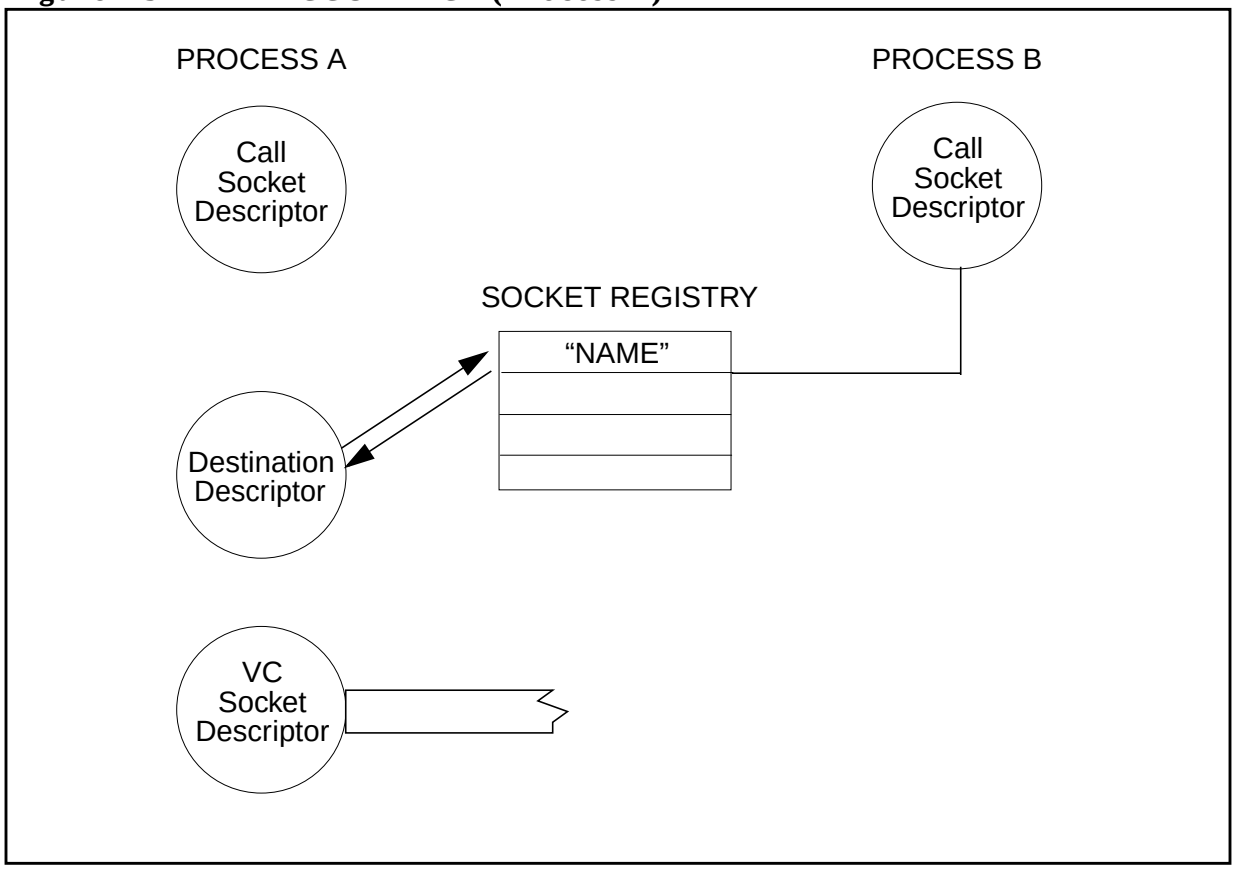
Figure 1-4 IPCLOOKUP (Process A)



4. Requesting a Connection

Process A specifies the destination descriptor returned by IPCLOOKUP and the call socket descriptor returned by IPCCREATE in its call. With these two parameters, IPCCONNECT requests a virtual circuit connection between Process A and Process B (see Figure 1-5). This could be compared to dialing a phone number. IPCCONNECT then returns a VC socket descriptor in its *vcdesc* parameter that describes the VC socket endpoint of the connection at Process A.

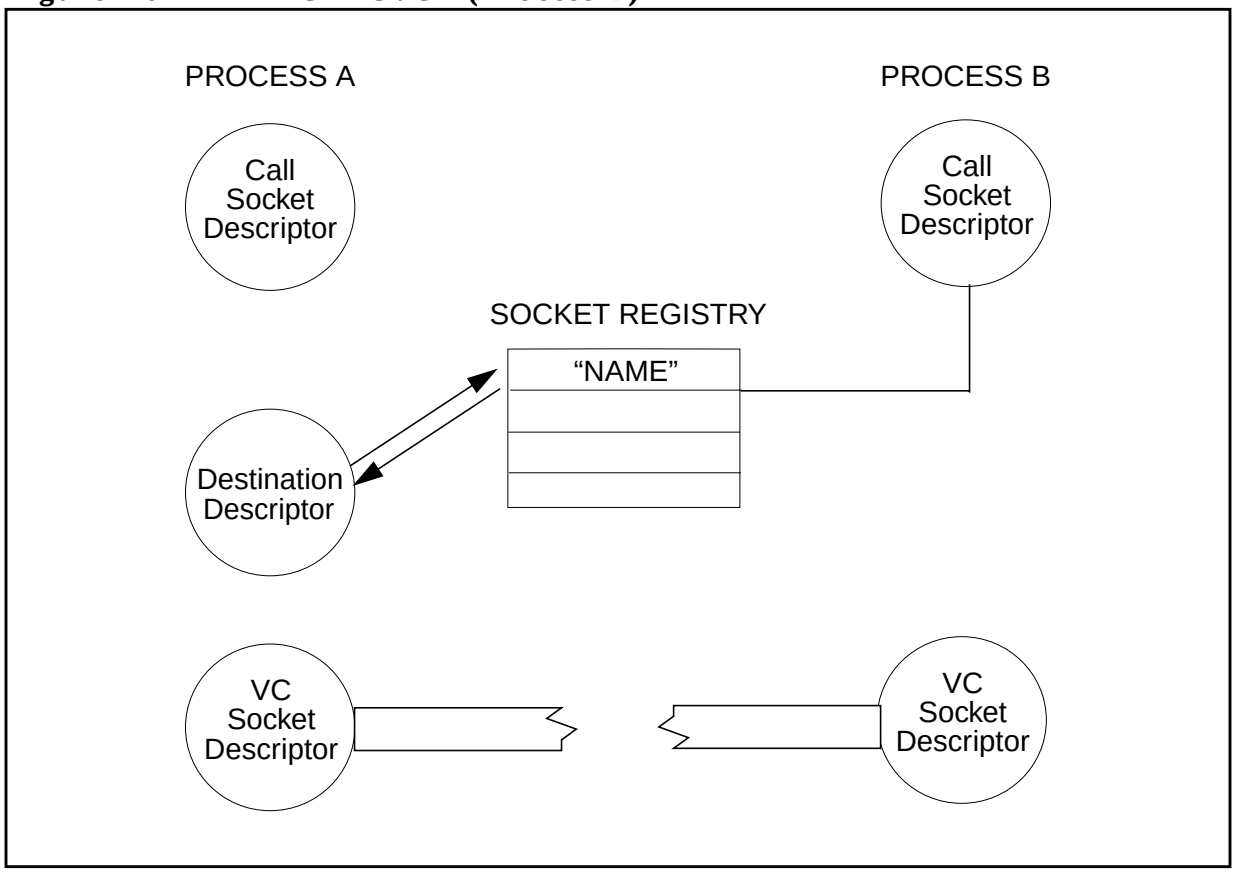
Figure 1-5 IPCCONNECT (Process A)



5. Receiving a Connection Request

Using the call socket descriptor returned by its `IPCCREATE` call, Process B calls `IPCRECVCN` to receive any connection requests. In this example, Process B receives a connection request from Process A. (Process A "dialed its telephone" to call Process B when it called `IPCCONNECT`.) `IPCRECVCN` returns a VC socket descriptor in its *vcdesc* parameter (see Figure 1-6). This VC socket is the endpoint of the virtual circuit at Process B. The connection will not be established, however, until Process A calls `IPCRECV`. In the telephone analogy, `IPCRECVCN` is similar to picking up a ringing phone and saying "hello".

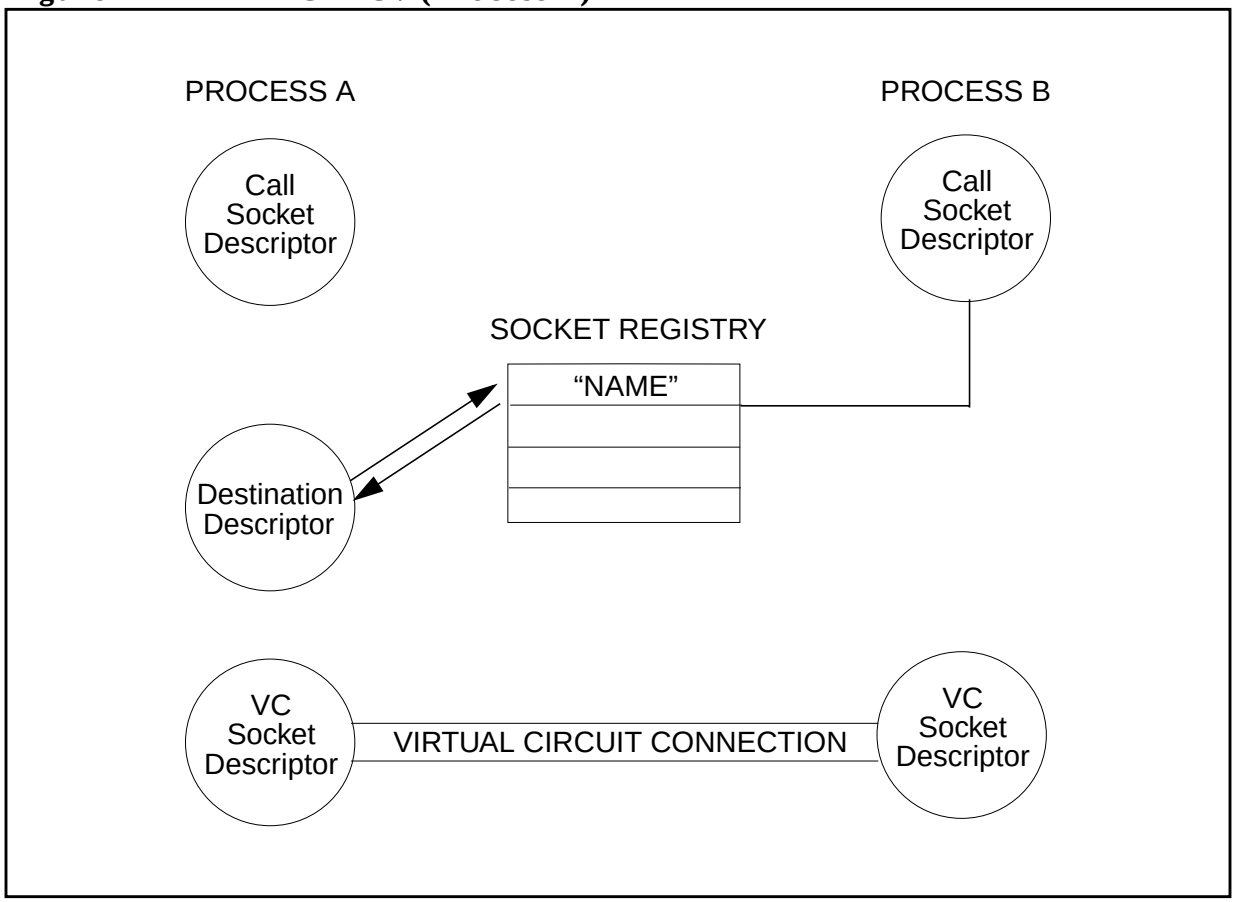
Figure 1-6 IPCRECVCN (Process B)



6. Checking the Status of a Connection

Process A calls IPCRECV using the VC socket descriptor returned by its IPCCONNECT call. IPCRECV returns the status of the connection (successful/unsuccessful) initiated by IPCCONNECT. If the status is successful, the connection has been established and Process A and Process B can "converse" over the new virtual circuit (see Figure 1-7). Compared to the telephone system, IPCRECV is similar to saying "hello" in response to the "hello" from the other end of the phone. IPCRECV can also be used to receive data. This function is described in the IPCRECV call discussion later in this section.

Figure 1-7 IPCRECV (Process A)

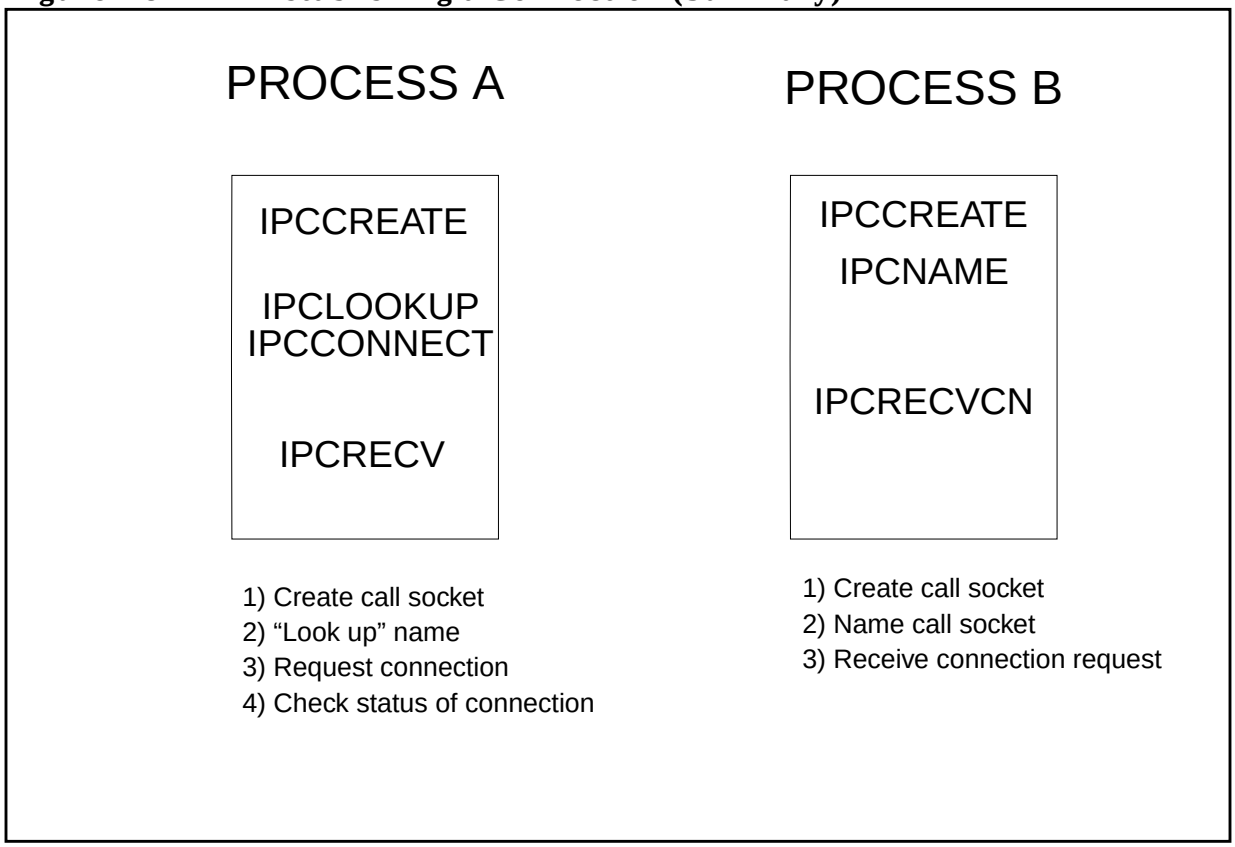


Connection Establishment Summary

The following discussions summarize the methods for establishing connections using NetIPC intrinsics.

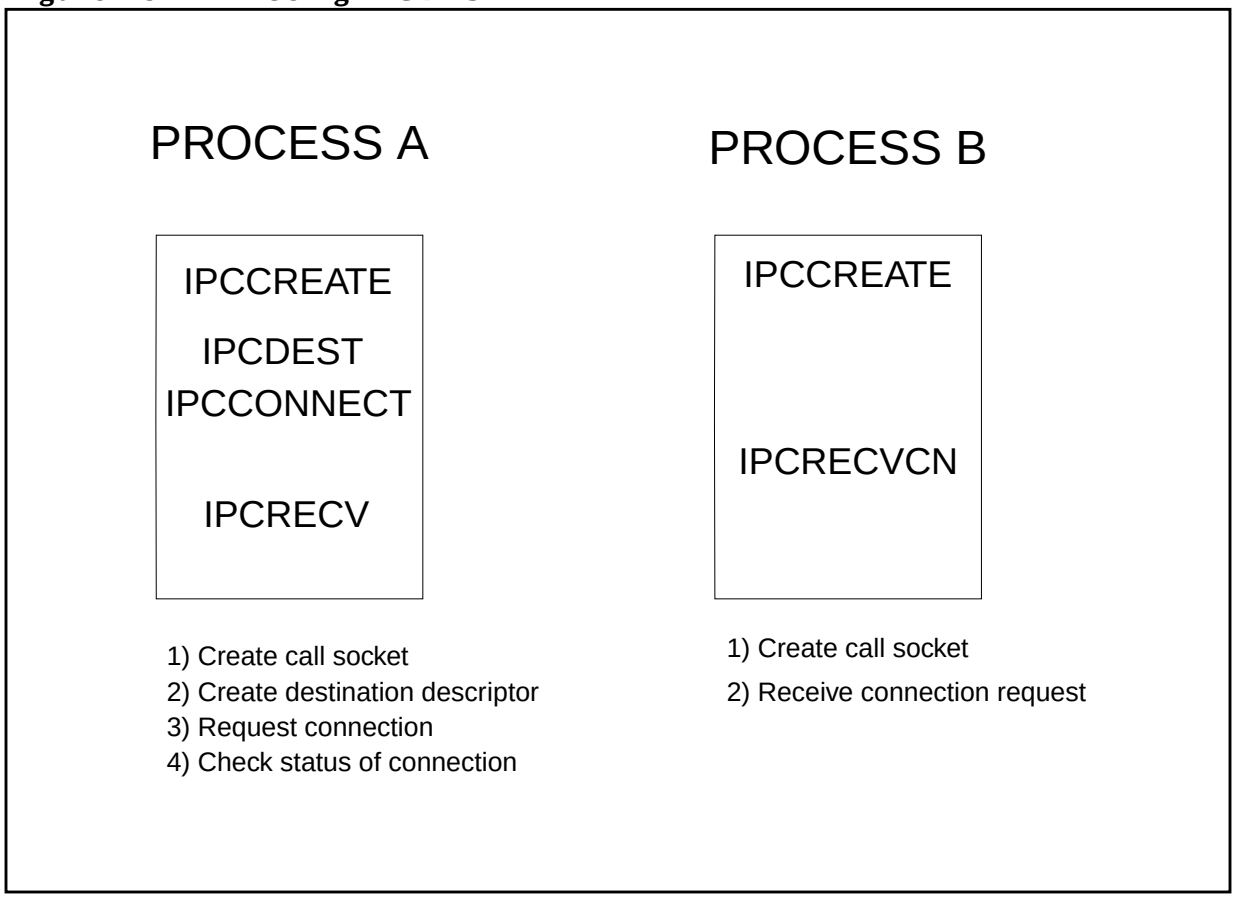
Connection Establishment Using IPCNAME <Figure 1-8 illustrates the sequence of NetIPC calls that is used to establish a virtual circuit connection. This figure summarizes the information presented in previous Figures.

Figure 1-8 Establishing a Connection (Summary)



Connection Establishment Using IPCDEST Figure 1-9 illustrates the sequence of NetIPC calls that is used to establish a virtual circuit connection when the protocol relative address of the remote node is known.

Figure 1-9 Using IPCDEST



Sending and Receiving Data Over a Connection

Once a virtual circuit connection is established, the two processes can exchange data using the NetIPC calls `IPCSEND` and `IPCRECV`. Either process can send or receive data. `IPCSEND` is used to send data on an established connection; it is analogous to “speaking” over a telephone connection. `IPCRECV` is used to receive data on an established connection; the use of `IPCRECV` is similar to “listening” at our telephone handset. (Note that `IPCRECV` has a dual function: to complete a virtual circuit connection as well as to receive data on a previously established connection.)

X.25 Access

Direct access to level 3 (X.25) provides message mode transfer. Stream mode is not supported for X.25. Each `IPCRECV` returns a complete message (provided the data length specified is of sufficient size). The X.25 protocol signals the end of message and NetIPC buffers the message until an `IPCRECV` (or required `IPCRECV`s), retrieve it.

TCP Access

For TCP access, all data transfers between user processes are in stream mode. In stream mode, data is transmitted as a stream of bytes with no end-of-message markers. This means that *the amount of data received in an individual* IPCRECV is not necessarily equivalent to the amount of data sent in an IPCSEND call. In fact, the data received may contain part of a message or even several messages sent by multiple IPCSEND calls.

You specify the maximum number of bytes you are willing to receive through a parameter of IPCRECV. When the call completes, that parameter contains the number of bytes *actually* received. This will never be more than the maximum amount you requested, but it may be less. The data you receive will always be in the correct order (in the order that the messages were sent), but there is no indication of where one message ends and the next one starts. It is up to the receiving process to check and interpret the data it actually receives. An application which does not need the information in the form of individual messages can simply process the data on the receiving side.

If an application is concerned about messages, the programmer must devise a scheme that allows the receiving side to determine what the messages are. If the messages are of a *known length*, the receiving process can execute a loop which calls IPCRECV with a maximum number of bytes equal to the length of the portion of the message not yet received.

Since IPCRECV returns to you the actual number of bytes received, you can continue to execute the loop until all the bytes of the message have been received. The following Pascal program fragment demonstrates this idea:

```
received_len := 0;
while (received_len < msg_length) and (errorcode = 0) do
  begin
    data_len := msg_length - received_len;
    ipcrecv( connection, tempbfr, data_len,,,errorcode );
    if errorcode = 0
        then strmove(data_len,tempbfr,1,databfr,
            received_len+1);
    received_len := received_len + data_len;
end;
```

In the above example, the Pascal function strmove takes each piece of the message received in tempbfr and concatenates it to the portion of the message already in databfr. Upon exiting the loop, the entire message has been stored in databfr.

If the length of the messages are not known, the sending side could send the length of the message as the first part of each message. In that case, the receiving side must execute two IPCRECV loops for each message: first to receive the length and then to receive the data. An example of this technique is shown at the end of this section.

Shutting Down Sockets and Connections

The NetIPC call IPCSHUTDOWN releases a descriptor and any resources associated with it. IPCSHUTDOWN can be called to release a call socket descriptor, a destination descriptor, or a VC socket descriptor. Because system resources are being used whenever descriptors exist, you should probably release them when they are no longer needed.

The call socket is needed as long as a process is expecting to receive a connection request on that socket. A process which receives a connection request can release the call socket any time after the IPCRECVCN connection request, as long as no other connection requests are expected for that call socket.

Similarly, a process which requests a connection can release its call socket any time after the call to IPCCONNECT, as long as it is not expecting to receive any more connection requests for that socket.

For TCP only, a process does not need to create a call socket (via IPCCREATE) at all; instead, it can use a temporary call socket by calling IPCCONNECT without specifying a call socket descriptor. (A temporary call socket is automatically destroyed when the IPCCONNECT call completes.) A process which requests a connection can also release the destination socket any time after the call to IPCCONNECT.

For example, in the section “Establishing a Connection”, Process A no longer needs the destination descriptor after calling IPCCONNECT and can use IPCSHUTDOWN to release the destination descriptor. In addition, if Process A does not expect to request additional connections, it can also call IPCSHUTDOWN a second time to release the call socket.

Process B, as described in “Establishing a Connection”, can call IPCSHUTDOWN to release its call socket any time after the call to IPCRECVCN (see “Receiving a Connection Request”). Process B should release its call socket only if it does not want to establish additional connections.

Before a process terminates, it should terminate its virtual circuit connections by releasing its VC sockets with IPCSHUTDOWN. If a process does not release its VC sockets before terminating, the system releases them when the process terminates. Because IPCSHUTDOWN takes effect very quickly, *all of the data that is in transit on the connection is lost when the connection is shut down*. As a result, if there is a possibility that data is in transit on the connection, the processes that share a connection must cooperate to ensure that no data is lost.

X.25 Access

X.25 direct access to level 3 does not support the *graceful release* bit. Using IPCSHUTDOWN on a VC socket descriptor causes a clear packet to be sent.

To ensure that no data packets are lost before the clear packet is sent, the D bit option can be set in the last IPCSEND. This assures end-to-end acknowledgment of this message before issuing the IPCSHUTDOWN to clear the virtual circuit.

Another method to ensure no lost packets is to send an unimportant message as the last message. The following example shows the calling sequence you would use. Note that the "dummy" message may or may not be received by the last IPCRECV. If the last message is not received, a SOCKERR 67 "CONNECTION FAILURE DETECTED" is returned.

```
IPCSEND          --->      IPCRECV
(important message)        (important message
                           received)

                 <----     IPCSEND
                           (dummy message sent)
IPCRECV
(receive dummy             IPCSHUTDOWN
 message)

IPCSHUTDOWN
```

TCP Access

To ensure that no data is lost, the IPCSHUTDOWN *graceful release* bit can be set, and the following sequence of steps can be followed:

- Process A calls IPCSHUTDOWN and sets bit 17, the *graceful release* flag. Process B receives a message (with an IPCRECV) informing it that Process A has called for *graceful release*. (This message is sent to B automatically when A sets the *graceful release* flag.) Process A enters a simplex-in state; meaning, it can receive, but not send, data. Process B will enter a simplex-out state, that is, it can send but cannot receive data. As a result, data that is in transit to Process A (which initiated the *graceful release* shutdown) will reach Process A without being lost.
- Next, one of two steps must occur to completely shut down the connection. Either (1) Process B initiates with or without its own *graceful release* or (2) Process A calls IPCSHUTDOWN *without* the *graceful release* option.
- This releases Process A's VC socket descriptor and shuts down the connection. In this case, Process B must also release its socket descriptor by calling IPCSHUTDOWN.

  If the *graceful release* option is not used (this may be necessary, for example, if the remote node does not support *graceful release*) the following steps should be followed when shutting down a connection.
- Process A sends a "last message" to Process B via an IPCSEND call. This message contains data that is recognized by Process B as a

termination request, and may also contain data to be processed by Process B. Process A then calls IPCRECV.

- Process B receives the message from Process A with a call to IPCRECV and sends a "confirmation message" to Process A via IPCSEND. This message contains data that indicates to Process A that it is okay to terminate the connection, and may also contain data to be processed by Process A. Process B then calls IPCRECV.
- Process A receives a "confirmation message" from Process B via the call to IPCRECV and calls IPCSHUTDOWN to release its VC socket descriptor and shut down the connection.
- The IPCRECV call of Process B completes with a *result* parameter value of 64 ("REMOTE ABORTED THE CONNECTION"). It then calls IPCSHUTDOWN to release its VC socket.

Additional NetIPC Functions

Once a virtual circuit is established between processes, descriptors can be given away, names can be erased, and other functions can be performed. The following NetIPC calls are provided in addition to those described in the previous paragraphs to enable you to perform these additional functions. A brief introduction to each call and its use follows. A complete description of NetIPC calls is provided in Chapter 3 , "NetIPC Intrinsics."

- **IPCCONTROL**. Performs special operations on sockets such as enabling synchronous mode, and changing asynchronous timeout values.
- **IPCDEST**. Returns a destination descriptor which can be used to send messages to another process. This is an alternative to naming the descriptor with IPCNAME and acquiring it with IPCLOOKUP.
- **IPCGET**. The companion call to IPCGIVE. Receives a call socket or virtual connection given away by a process that has called IPCGIVE. This call is similar to IPCLOOKUP because it enables your process to acquire a descriptor that can be used in subsequent NetIPC calls.
- **IPCGIVE**. The companion call to IPCGET. Releases ownership of a descriptor to NetIPC so that it can be acquired by another process via a call to IPCGET.
- **IPCNAMERASE**. Does the reverse of IPCNAME: it removes a name associated with a call socket from the socket registry. Only the owner of a call socket descriptor can remove its name.

Direct Access to Level 3 (X.25)

Features

The following features of direct access to level 3 (X.25) with NetIPC are described in this section:

- Supports switched virtual circuits (SVCs) and permanent virtual circuits (PVCs).
- Provides access to the call user data (CUD) field in call packets.
- Provides access to X.25 addresses in call packets.
- Creation of a catch-all socket which can be used to accept data packets with no CUD or unrecognized CUD.
- Provides ability to send up to 128 bytes of call user data using the fast select facility.
- Provides ability to append, generate and examine the facility field in call packets.
- Provides access to X.25 protocol features.
- Allows direct specification of the target X.25 address or PVC number.

Limitations

Limitations using direct access to level 3 (X.25) are:

- Intranet use only (level 4 provides internet and intranet connections)
- One virtual connection socket accesses one X.25 virtual circuit for data transfers over X.25. Multiplexing of connections over a virtual circuit is not supported.
- `IPCNAME`, `IPCNAMERASE` and `IPCLOOKUP` are not supported.

Switched Virtual Circuits (SVCs)

Switched virtual circuits are defined as a logical association that only exists as long as the connection does. Both processes create their own local call sockets using `IPCCREATE` that can be associated with protocol relative addresses. To establish a connection with a specific server process, a request process must include a server protocol relative address in the `IPCDEST` intrinsic. Alternatively, an *`opt`* parameter in `IPCCREATE` can be used to create a catch-all socket where any incoming

request for a connection can be accepted (whether or not the server protocol relative address exists). A catch-all socket receives incoming call requests that do not match any other given protocol relative address. One catch-all socket can be defined for each X.25 network.

As an example, two programs communicating over an SVC can be designated as the requester and server. Both programs need to be running in order for communication to occur. Figure 1-10 shows the order of NetIPC calls used for a requestor program and the X.25 packets generated as a result of the calls. Figure 1-11 describes the order of NetIPC calls used for a server program.

NOTE

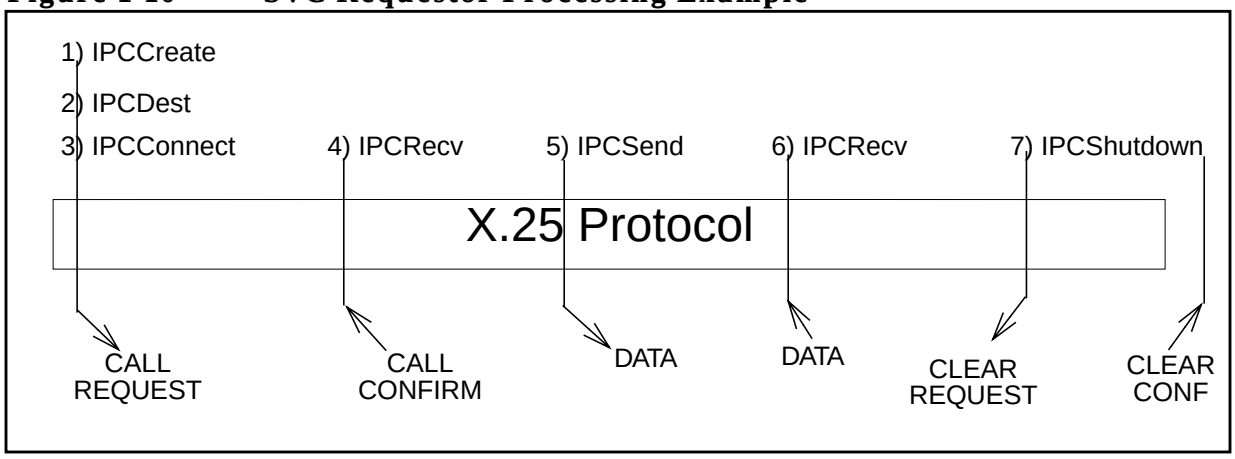
Note that Figure 1-10 and Figure 1-11 do not show synchronization of data transfer between the two programs, and do not include error checking, or the intrinsic calls required for adding options and special user capabilities. See example 3 in Chapter 4 , “NetIPC Examples,” of this manual for programmatic examples of a server and requestor using access to the X.25 protocol.

SVC Requestor Example

Figure 1-10 shows the order of NetIPC calls used for a requestor program and the X.25 packets generated as a result of the calls. The calls outlined in Figure 1-10 perform the following functions:

1. Create a call socket with `IPCCREATE`. The call socket descriptor (*`calldesc`*) is returned.
2. Create a destination descriptor socket (*`destdesc`*) with `IPCDEST`. You must specify a remote protocol relative address (*`protoaddr`*) to be associated with the destination descriptor.
3. Establish the virtual circuit socket with `IPCCONNECT`, supplying the *`calldesc`* and *`destdesc`* created by the previous two calls.
4. Receive a response to the connection request with `IPCRECV`, setting the data length parameter (*`dlen`*) equal to zero.
5. Send a message over the connection with `IPCSEND`.
6. Receive a message over the connection with `IPCRECV`.
7. Shutdown the connection with `IPCSHUTDOWN`. Cause and diagnostic values and/or clear user data can be entered that will be included in an X.25 clear packet sent as a result of this call.

Figure 1-10 SVC Requestor Processing Example

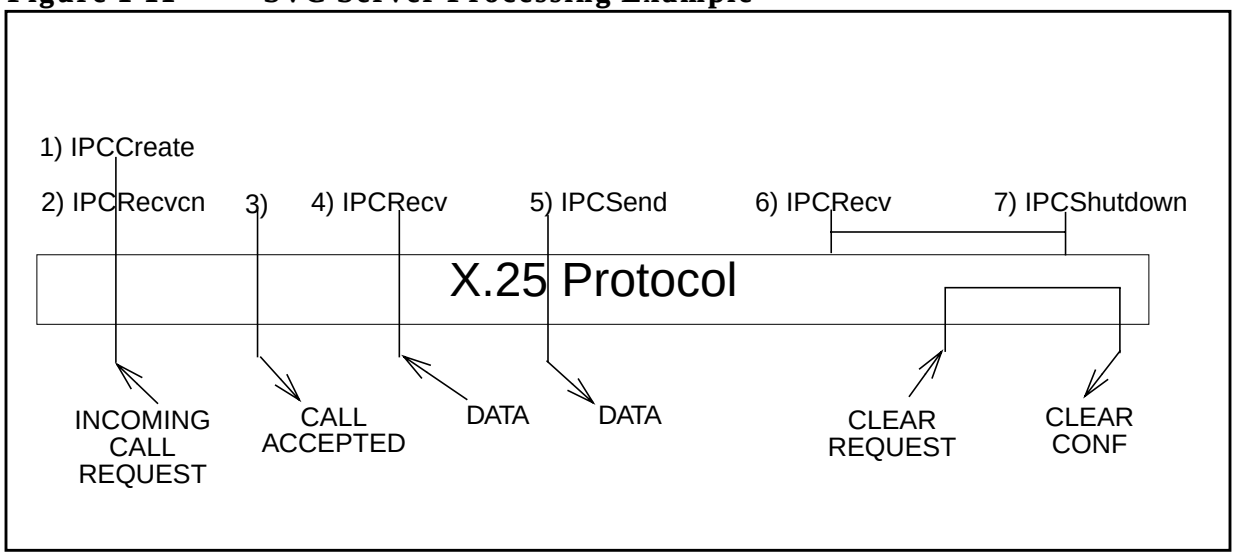


SVC Server Example

Figure 1-11 shows the order of NetIPC calls used for a server program and the X.25 packets generated as a result of the calls. The calls outlined in Figure 1-11 perform the following functions:

1. Create a call socket with `IPCCREATE`. The call socket descriptor (*`calldesc`*) is returned. The socket could be created as a catch-all and/or bound to a particular protocol relative address.
2. Call `IPCRECVCN` and wait for an incoming call request packet. `IPCRECVCN` will return a VC descriptor (*`vcdesc`*) when it is established that the incoming protocol relative address defined in (1) matches the incoming protocol relative address, or a catch-all socket was created in (1).
3. As `IPCRECVCN` completes and returns a *`vcdesc`*, X.25 sends the requestor process a call accepted packet.
4. Receive a message over the connection with `IPCRECV`.
5. Send a message over the connection with `IPCSEND`.
6. Since the server (`IPCRECV`) in this example waits to receive a message, you may decide to set a timer to handle the inactivity.
7. (Optional step.) Shutdown the connection with `IPCSHUTDOWN` after data has not been received for a period of time. (For example, after a timeout has occurred.) Note that the X.25 protocol implicitly handles the incoming clear request by sending a clear confirmation packet.

Figure 1-11 SVC Server Processing Example



Permanent Virtual Circuits (PVCs)

Permanent virtual circuits are defined as two DTEs with a logical association permanently held by the network. Since the connection is permanent, both processes must initiate the connection using the IPCCREATE intrinsic. Both processes must specify the destination of a connection request with the IPCDEST intrinsic which requires a node name corresponding to a configured PVC number.

The possible ordering of intrinsic calls to communicate over a PVC could be as follows:

1. Create a call socket with IPCCREATE. The call socket descriptor (*calldesc*) is returned.
2. Create a destination descriptor socket (*destdesc*) with IPCDEST.
3. Establish the virtual circuit socket with IPCCONNECT, supplying the *calldesc* and *destdesc* created by the previous two calls.
4. Send a reset packet (to the DCE) by setting the reset request in IPCCONTROL.
5. Send an interrupt packet to the remote process by setting the interrupt request in IPCCONTROL.
6. Send data over the connection with IPCSEND.
7. Receive data over the connection with IPCRECV.
8. Send a reset packet by setting the reset request in IPCCONTROL when all data has been sent/received.
9. Shutdown the connection with IPCSHUTDOWN. Note that a PVC is a permanent connection, and the shutdown process releases the resources associated with the connection.

Note that these steps do not show how to synchronize data transfer between the two programs, and do not include error checking, or the intrinsic calls required for adding options and special user capabilities.

Access to the Call User Data (CUD) Field

The NetIPC intrinsics IPCCONNECT, IPCRECVCN, IPCCONTROL, IPCRECV (on connection establishment), and IPCSHUTDOWN (with fast select) provide access to the call user data (CUD) field in call packets as follows:

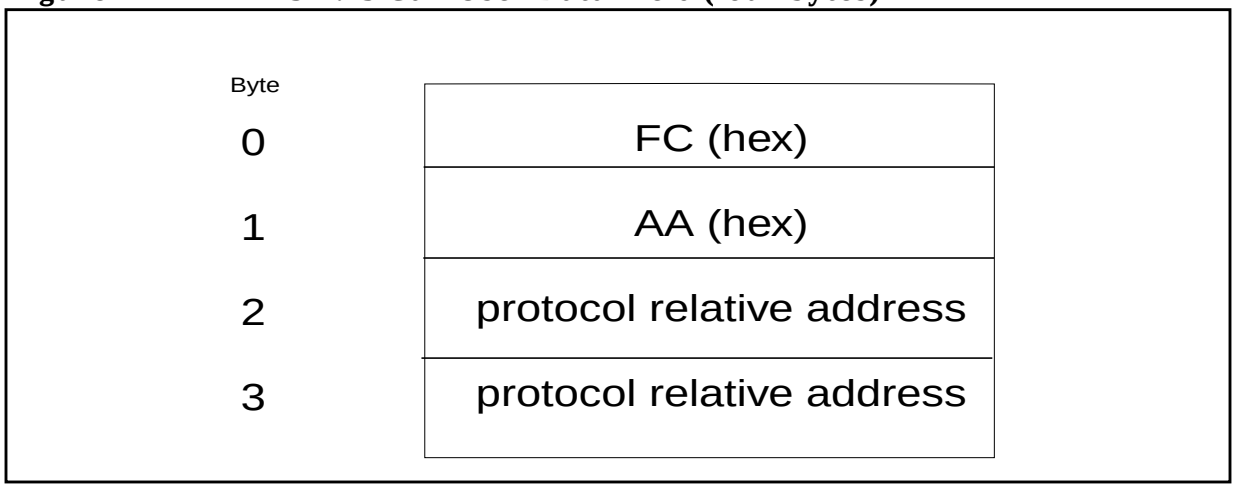
- **Specifying a protocol relative address in the CUD**.

  This field may be present in X.25 call request and incoming call packets which you can access with IPCCONNECT and IPCRECVCN. The call user data field can only be accessed over an SVC. The maximum length of the call user data (CUD) field is normally 16 bytes. The CUD can be up to 128 bytes if the fast select facility is available. For NS X.25, the first four bytes of the CUD are reserved for protocol relative addressing. Figure 1-12 shows the contents of the first four bytes of the HP 3000 X.25 CUD. The first two bytes, as shown in Figure 1-12, indicate that the source of the call request packet is an HP 3000 node using direct access to level 3. Optionally, the last two bytes contain the protocol relative address that the call request expects to find (if any).

  To access the entire CUD (16 bytes without fast select or 128 bytes with fast select), the *opt* parameter *protocol flags* bit 17 can be set in IPCCONNECT. This option is useful for communication with non-HP nodes.

  See the discussion of the Fast Select Facility for examples using NetIPC intrinsics to send and/or receive call user data using fast select.

Figure 1-12 NS X.25 Call User Data Field (four bytes)



- **Connecting to a catch-all socket**.

  Using IPCCREATE, you can identify a socket as a catch-all socket. All incoming calls with a protocol relative address specified in the CUD that does not match any given protocol relative address are routed to the catch-all socket. Only one catch-all socket may be defined for each X.25 network on each node.

  For an incoming call with a protocol relative address specified, NetIPC checks if the address matches one created. If it matches, the call is accepted. If it does not match, NetIPC checks for the existence of a catch-all socket. If no catch-all socket has been created, the call is rejected and a clear packet is sent by X.25. If a catch-all socket has been created, the call is accepted.

  If no protocol address is specified in the incoming call, NetIPC checks for the existence of a catch-all socket. If no catch-all socket has been defined, the call is rejected. If there is a catch-all socket, the call is accepted.

- **Defer connection requests**.

  Using IPCRECVCN with the deferred flag set allows you to examine the call user data, facilities, and calling node address before accepting or rejecting the connection.

  The IPCCONTROL intrinsic provides you with the capability to accept or reject a connection request that is in the deferred state.

Fast Select Facility

The fast select facility, a datagram service, is available over an X.25 network. Fast select is configured as a parameter in the X.25 User Facility Set Parameters screen during NMMGR configuration of X.25 on the HP 3000. (See the *X.25 XL System Access Configuration Guide* for more information.) When you use the fast select facility, up to 128 bytes of call user data may be sent in a X.25 call packet using NetIPC intrinsics.

Fast Select Options

Fast select has two options:

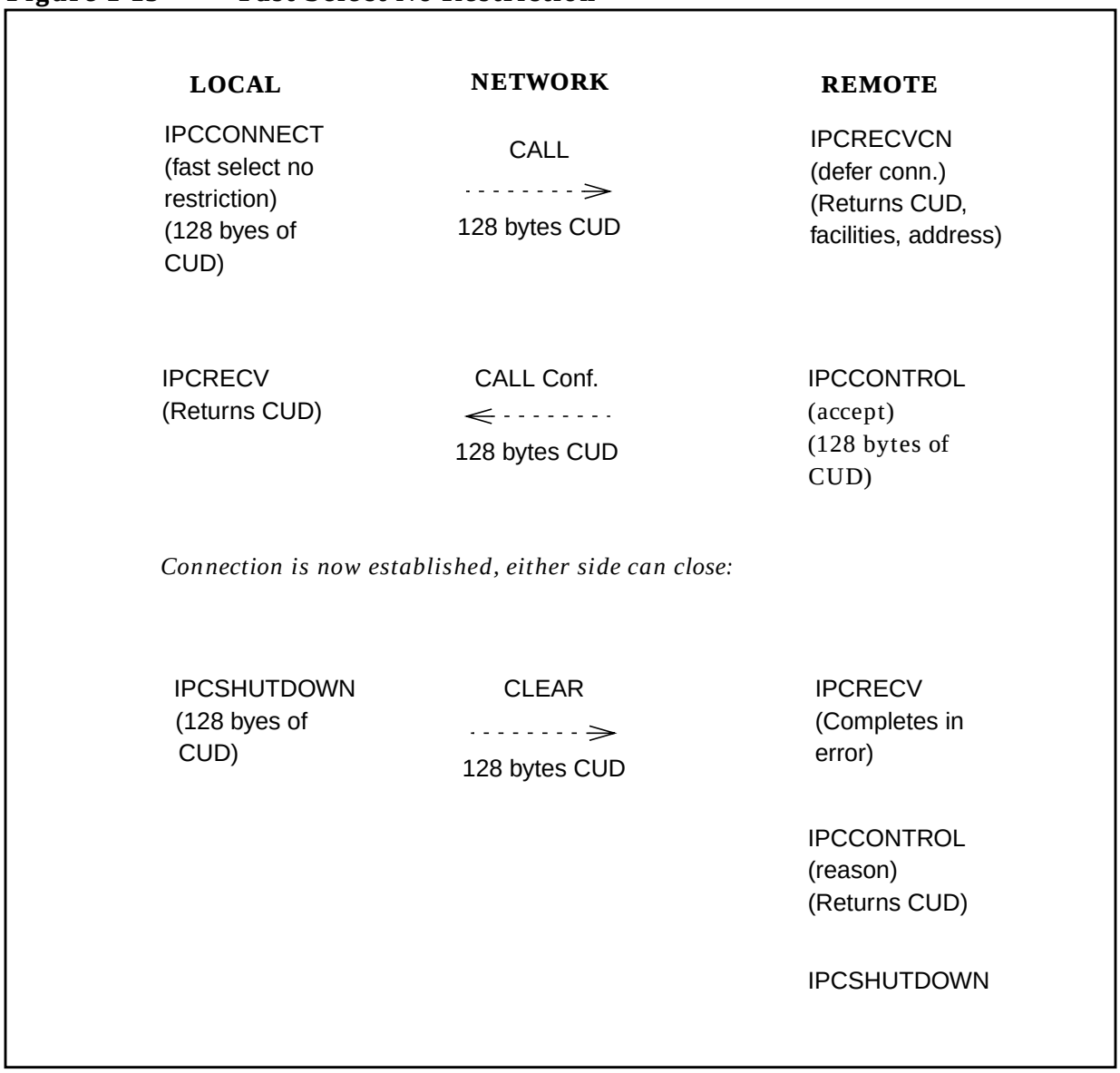
- **No restriction on response**: allows the receiver of the connection request the choice of either accepting or rejecting the connection.
- **Restriction on response**: the receiver must always reject the connection.

Using Fast Select

The receiver of the connection can use IPCRECVCN to examine the call user data, facilities or calling node's address before deciding to accept or reject the connection.

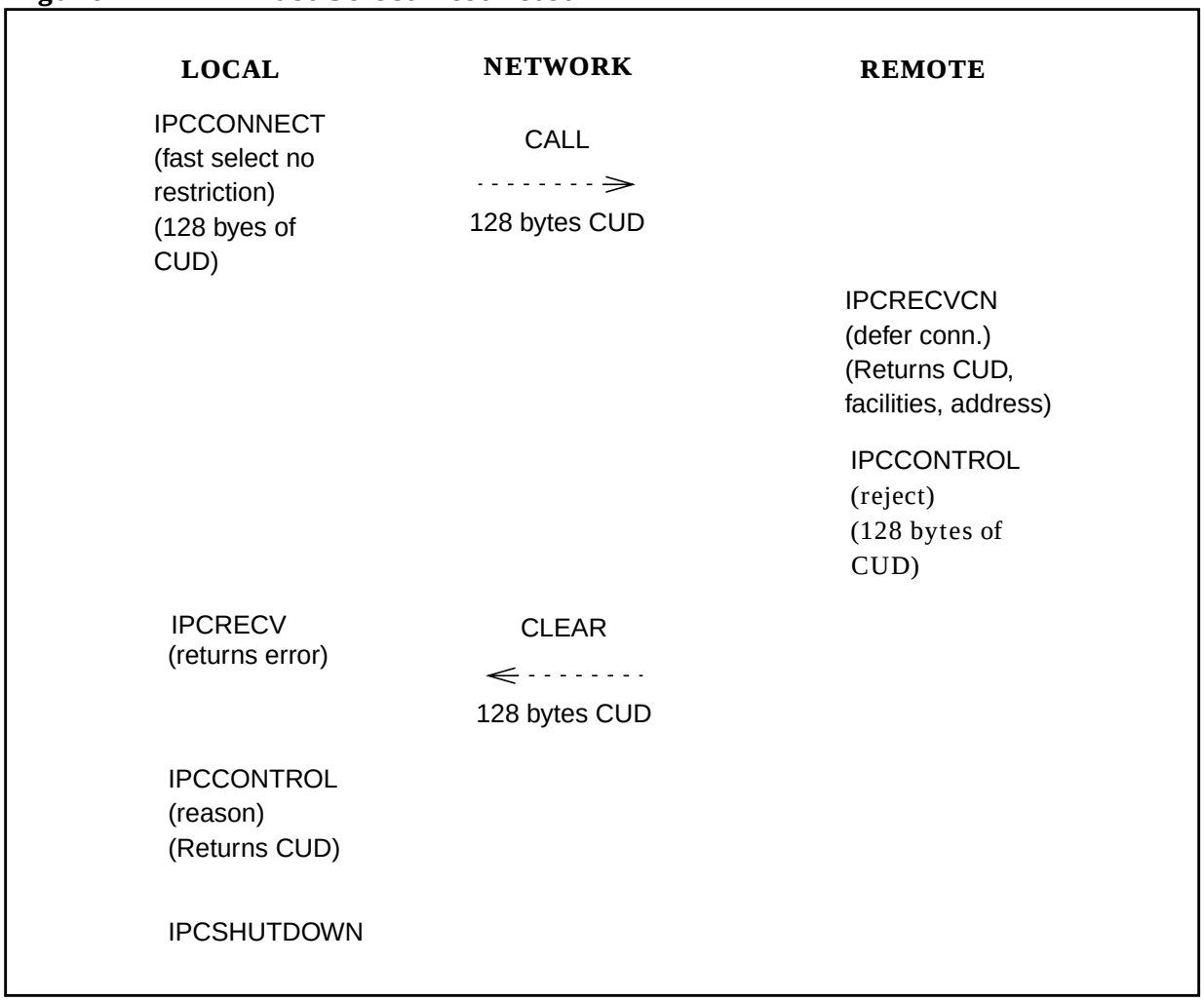
If the connection is accepted, up to 128 bytes of call user data may be placed in the call confirmation packet using IPCRECVCN. Once accepted, the virtual circuit is open and can be used as a virtual circuit that does not have fast select. The side that closes the connection can send 128 bytes of clear user data in the clear packet using the IPCSHUTDOWN call. Figure 1-13 is an example of the sequence of calls used with fast select, no restriction.

Figure 1-13 Fast Select No Restriction



If the connection is rejected, up to 128 bytes of clear user data may be put in the clear packet using IPCSHUTDOWN. Figure 1-14 is an example using fast select with restriction.

Figure 1-14 Fast Select Restricted



Facility Field

The X.25 facility field is built from the facility set configured with NMMGR. With direct access to X.25 level 3, the facility field can be appended with facilities specified in the *opt* parameter of the IPCCONNECT intrinsic. The values for the facilities used must follow the X.25 recommendation.

For example, a user wants to use a facility set with packet size and window size negotiation and wants to append the CCITT-specified DTE facilities to that facility set. In NMMGR, the user has verified the facility set contains packet size and window size negotiation. The facility field generated from the facility set would contain the following (in octal):

```
Facility length                          : %6
Packet size neg code field               : %102
Packet size in                           : %10
Packet size out                          : %10
Window size net code field               : %103
Window size in                           : %7
Window size out                          : %7
```

To add the DTE facilities, the user must specify all the bytes required by the X.25 recommendation in the IPCCONNECT request. In this example, to add the DTE facilities (calling address extension, called address extension and QOS end-to-end delay) the following values must be entered in the option field buffer (shown in octal):

```
CCITT-specified DTE fac marker           : %17
Calling add extension code               : %313
Calling add extension length             : %3
Calling add extension                    : %5
                                         : %5
Called add extension code                : %312
Called add extension length              : %3
Called add extension                     : %6
                                         : %6
QOS minimum throughput                   : %12
QOS throughput in/out                    : %314
```

NetIPC appends this to the facility-set generated field, and updates the “facility length”. In this example, the facility length would be %21.

When a connection is received, the user can examine the entire facility field (see IPCRECV).

Access to X.25 Protocol Features

The NetIPC intrinsics provide access to the following X.25 protocol features:

- **Send and receive interrupt and reset packets.**

  You can request the X.25 protocol to send an interrupt or reset packet with IPCCONTROL. When used in this way, the IPCCONTROL intrinsic will not return until the appropriate confirmation packet is received by X.25.

- **Set no activity timeout.**

  You can set a no activity timeout value with the IPCCONTROL intrinsic. This option clears the connection after the specified time if no data packets are exchanged on the virtual circuit.

- **Qualifying X.25 data packets.**

  The Q bit in the general format identifier field in an X.25 data packet can be set using the IPCSEND intrinsic. The status of the Q bit in incoming data packets is returned in the IPCRECV intrinsic. The Q bit status can be used to indicate whether the data is a user message (Q bit=0) or a device control message (Q bit=1) from or to a remote PAD.

- **Set end-to-end acknowledgment.**

  The D bit in the general format identifier field in an X.25 data packet can be set using the IPCSEND intrinsic. The status of the D bit in incoming data packets is returned in the IPCRECV intrinsic.

  Setting the D bit locally specifies end-to-end acknowledgment of data packets. IPCSEND does not complete until it receives acknowledgment that the entire message has been received. For HP 3000 to HP 3000 communication, IPCRECV initiates the acknowledgment when the remote HP 3000 process has received the entire message.

- **Set cause and diagnostic codes.**

  Using IPCSHUTDOWN, you can enter a reason code that will be included in X.25 clear packets as cause and diagnostic values. This option is only used with SVCs. Reasons for events or errors are returned by IPCCONTROL. See Appendix B , “Cause and Diagnostic Codes,” for a list of diagnostic codes used with X.25 protocol access. Note that when the DTE sends the clear packet, the cause code is always set to zero.

NetIPC Between MPE-XL and MPE-V Systems

NetIPC applications can be written to communicate between MPE-V and MPE XL-based HP 3000s with the following considerations:

- In addition to the X.25 features supported on MPE-V, NetIPC on MPE XL systems supports the following:
  - — Fast select facility
  - — Ability to append, generate and examine the facility field in call packets
  - — Directly specifying the remote host X.25 address or PVC number.
- There are additional differences between the MPE-V and MPE XL X.25 implementations that can affect NetIPC programming:
  - — On MPE XL, `IPCCONTROL` *`request`* 12, reason for error or event only returns 10 (clear), 11 (reset), or 12 (interrupt). On MPE-V, in addition to returning 10, 11, 12, `IPCCONTROL` returns 14 (network shutdown), 15 (restart sent by local network), 16 (level 2 failure), 17(restart sent by local protocol module), and 18 (restart packet received).
  - — On MPE XL, `IPCSHUTDOWN` completes immediately, while on MPE-V it does not complete until a clear confirmation arrives.
  - — On MPE XL, PAD calls are sent to the catch-all socket while on MPE-V PAD calls are sent to socket #2563.
  - — On MPE XL a reset is not sent to initialize or clear a PVC, while on MPE-V a reset is sent.
  - — MPE-V has a timeout for interrupt collisions, MPE XL does not.
  - — On MPE XL, the *`IPCCREATE`* parameter *`network name`* must be padded with blanks while on MPE-V, the network name is padded with nulls.

2 Cross-System NetIPC

A cross-system application refers to NetIPC communication between processes running on computers of different types. Cross-system NetIPC is supported using access to the Transmission Control Protocol (TCP) only. This chapter explains what NetIPC calls using TCP access need to be considered for a cross-system application between an HP 3000 and HP 1000 and between an HP 3000 and HP 9000 (Series 300 or 800). Cross-system NetIPC is also supported between HP 3000s and personal computers (PCs) in an HP Office Share Network.

NetIPC communication between MPE V based and MPE XL based HP 3000s is not considered cross-system. Chapter 2 , "Cross-System NetIPC," in this manual contains a section on MPE XL and MPE-V NetIPC communication. See the *NetIPC 3000/V Programmer's Reference Manual* for more information about NetIPC on MPE V based HP 3000s.

In a cross-system communication, you can have NetIPC programs running on both computer systems. NetIPC enables each program to send data to, or receive data from, the program on the remote system.

This chapter does not explain details about the NetIPC calls available on HP 1000 or HP 9000 computers, or personal computers. For this information, refer to the following manuals

- *NS/1000 User/Programmer Reference Manual.*
- *HP 9000 NetIPC Programmer's Guide (for the Series 300 and 800).*
- *PC NetIPC/RPM Programmer's Reference Guide.*

Software Required

For cross-system NetIPC to function properly, the software revision codes must be as follows:

- NS/1000 software revision code 5.0 or greater for the HP 1000.
- NS 3000/XL Release 1.2 or later for the HP 3000.
- LAN/9000 Series 800 Release 2.1 or later for the HP 9000 Series 800.
- NS-ARPA Services Release 6.2 or later for the HP 9000 Series 300.
- Revision B.00.01 for the personal computer.

To use cross-system NetIPC, you must first have a good understanding of the NetIPC intrinsics. Review Chapter 3 , “NetIPC Intrinsics,” which provides detailed information about the calls before and while you read this chapter.

Calls Affecting the Local Process

There are two categories of calls when considering cross-system NetIPC communication — local and remote. Calls made for the local process do not directly affect the remote process. The local NetIPC calls are used to set up or prepare the local node for interprocess communication with the remote node. That is, the resulting impact on the local calls is only to the local node. There is no information that needs to be passed to the remote node. This is true whether or not the remote node is another HP 3000 computer system. Table 2-1 lists the NetIPC calls affecting the local process.

Table 2-1 NetIPC Calls Affecting the Local Process

HP 3000	HP 1000	HP 9000	PC
ADDOPT	Addopt	addopt ()	AddOpt
Not implemented	Adrof	*Not implemented*	*Not implemented*
Not implemented	*Not implemented*	*Not implemented*	ConvertNetworkLong
Not implemented	*Not implemented*	*Not implemented*	ConvertNetworkShort
INITOPT	InitOpt	initopt()	InitOpt
IPCCHECK	*Not implemented*	*Not implemented*	*Not implemented*
IPCCONTROL	IPCControl	ipccontrol()	IPCControl
IPCCREATE	IPCCreate	ipccreate()	IPCCreate
IPCERRMSG	*Not implemented*	*Not implemented*	*Not implemented*
IPCGET	IPCGet	*Not implemented*	*Not implemented*
IPCGIVE	IPCGive	*Not implemented*	*Not implemented*
IPCNAME	IPCName	ipcname()	*Not implemented*
IPCNAMERASE	IPCNamerase	ipcnamerase()	*Not implemented*
Not implemented	IPCSelect	ipcselect()	*Not implemented*
Not implemented	*Not implemented*	*Not implemented*	IPCWait
OPTOVERHEAD	*Not implemented*	optoverhead()	OptOverhead
READOPT	ReadOpt	readopt()	ReadOpt

The intrinsics listed in Table 2-1 affect only local processes and therefore have no adverse affects if used in a program communicating with an unlike system (e.g., an HP 3000 program communicating with an HP 1000 program). However, keep in mind that the calls (even those of the same name) differ between system types. The following are some local call differences of which you should be aware:

- **Maximum number of sockets.** The maximum number of socket descriptors owned by an HP 3000 process at any given time is 64; on the HP 1000 the maximum is 32; on HP 9000 systems, the maximum is 60 (including file descriptors). On the PC, the maximum number of socket descriptors is 21. This number includes both call socket and virtual circuit socket descriptors.
- *IPCCONTROL* **parameters.** The `IPCCONTROL` intrinsic supports different sets of request codes on different system types. Refer to the NetIPC documentation for a particular system (this manual only documents the HP 3000) for a full description of the request codes available on that system.
- **Manipulation of descriptors**. On the HP 3000, the `IPCGIVE`, `IPCGET`, `IPCNAME`, and `IPCNAMERASE` calls can be used to manipulate call socket descriptors. On the HP 9000, you can manipulate call socket and destination descriptors with the `ipcname()` and `ipcnamerase()` intrinsics. On the HP 1000, you can only manipulate call socket descriptors with the `IPCName` and `IPCNamerase` intrinsics. In addition, on the HP 1000, you can manipulate call socket, vc socket, and path report descriptors with the `IPCGive` and `IPCGet` intrinsics.
- **Asynchronous I/O.** The HP 3000 utilizes the MPE XL intrinsics `IOWAIT` and `IODONTWAIT` to perform asynchronous I/O. On the HP 9000 and HP 1000, The NetIPC intrinsics `ipcselect()` and `IPCSelect` are used to perform asynchronous I/O. On the PC use the NetIPC intrinsic `IPCWait`.
- **Call sockets.** On the PC, call sockets are called source sockets and call socket descriptors are called source socket descriptors. Both sets of terms are used in the same way.

NOTE

There are many additional differences between local NetIPC calls for the HP 3000 and those used for other HP systems. Because these differences only affect the local node, they should not affect the cross-system communication capabilities of your program. Refer to the corresponding system's NetIPC documentation for more information.

Calls Affecting the Remote Process

Table 2-2 lists the NetIPC calls affecting communication with the remote process.

Table 2-2 NetIPC Calls Affecting the Remote Process

HP 3000	HP 1000	HP 9000	PC
IPCCONNECT	IPCConnect	ipcconnect()	IPCConnect
IPCDEST	IPCDest	ipcdest()	IPCDest
IPCLOOKUP	IPCLookUp	ipclookup()	*Not implemented*
IPCRECV	IPCRecv	ipcrecv()	IPCRecv
IPCRECVCN	IPCRecvCn	ipcrecvcn()	IPCRecvCn
IPCSEND	IPCSend	ipcsend()	IPCSend
IPCSHUTDOWN	IPCShutDown	ipcshutdown()	IPCShutDown

HP 3000 to HP 1000 NetIPC

The NetIPC calls affecting cross-system communication with the remote process have the following differences: checksumming, send and receive sizes, range of permitted TCP protocol addresses for users, and socket sharing. Table 2-3 summarizes the cross-system considerations.

Table 2-3 Cross-System Calls (HP 3000 — HP 1000)

NetIPC Call	Cross-System Considerations
IPCCONNECT	**Checksumming** — TCP checksumming will be enabled for both sides of the connection if it is enabled by either side for HP 3000 to HP 1000 cross-system communication. On both the HP 3000 and HP 1000 checksumming can be enabled by setting bit 21 in the *flags* parameter. **Send and receive sizes** — The HP 3000 send and receive size range is 1 to 30,000 bytes. The HP 1000 send and receive size range is 1 to 8,000 bytes. For example, if the HP 3000 node sends 16,000 bytes, the HP 1000 node can call IPCRECV twice, receiving the first 8,000 bytes the first time and the second 8,000 bytes the second time. Note that the default send and receive sizes are different on different HP systems. On the HP 3000, the default send and receive size is less than or equal to 1,024 bytes. On the HP 1000 the default send and receive size is 100 bytes.
IPCCREATE IPCDEST	**TCP protocol address** — The recommended range of TCP addresses for cross-system user applications is from 30767 to 32767 decimal (%74057 to%77777).

Table 2-3 Cross-System Calls (HP 3000 — HP 1000)

NetIPC Call	Cross-System Considerations
IPCLOOKUP	No differences that affect-cross-system operations.
IPCRECV	**Receive size** (*dlen* parameter) — Range for the HP 3000 is 1 to 30,000 bytes. Range for the HP 1000 is 1 to 8,000 bytes. Although the ranges are different, cross-system communication is not affected. If you specify a send or receive size, be sure it is within the correct range for the respective system. **Data wait flag** — The HP 1000 IPCRecv call supports a "DATA_WAIT" flag. This flag, when set, specifies that the call will not complete until the amount of data specified by the *dlen* parameter has been received. This flag is not available on the HP 3000, meaning that the call may complete before all the data is received. However, the HP 3000 IPCRECV supports other flags such as the "more data" and "destroy data" flags. Refer to the description of IPCRECV in Chapter 3 , "NetIPC Intrinsics," for more information.
IPCRECVCN	**Checksumming** — TCP checksumming will be enabled for both sides of the connection if it is enabled by either side for HP 3000 to HP 1000 connections. On both the HP 3000 and HP 1000 checksumming can be enabled by setting bit 21 in the *flags* parameter. Send and receive sizes — The HP 3000 send and receive size range is 1 to 30,000 bytes. The HP 1000 send and receive size range is 1 to 8,000 bytes. For example, if the HP 3000 node sends 16,000 bytes, the HP 1000 node can call IPCRECV twice, receiving 8,000 bytes the first time and the second 8,000 bytes the second time. Note that the default send and receive sizes are different on different HP systems. On the HP 3000, the default send and receive size is less than or equal to 1,024 bytes. On the HP 1000 the default send and receive size is 100 bytes.
IPCSEND	**Send size** — The HP 3000 send size range is 1 to 30,000 bytes. The HP 1000 send size is 1 to 8,000 bytes. Although the ranges are different, cross-system communication is not affected. If you specify a send or receive size, be sure it is within the correct range for the respective system. Note that the urgent data bit is not supported on the HP 1000; however, if this bit is set by the HP 3000 program, it will be ignored by the receiving process on the HP 1000.
IPCSHUTDOWN	**Socket shut down** — The HP 3000 provides a graceful release flag (flag 17) that is not available on the HP 1000. Do not set the graceful release flag on the HP 3000. Otherwise, the HP 1000 will not perform a normal shutdown.

HP 3000 to HP 9000 NetIPC

The NetIPC calls affecting cross-system communication with the remote process have the following differences. Checksumming, send and receive sizes, range of permitted TCP protocol addresses for users, and socket sharing. Table 2-4 lists the NetIPC calls affecting the remote process and summarizes the cross-system considerations.

Table 2-4 Cross-System Calls (HP 3000 — HP 9000

NetIPC Call	Cross-System Consideration
IPCCONNECT	**Checksumming** — When the ipcconnect( ) call is executed on the HP 9000 node, checksumming is always enabled. Therefore checksumming is always enabled for the HP 3000-to-HP 9000 connection. **Send and receive sizes** — The HP 3000 send and receive size range is 1 to 30,000 bytes. The HP 9000 send and receive size range is 1 to 32,767 bytes. Although the ranges are different, cross-system communication is not affected. If you specify a send or receive size, be sure it is within correct range for the respective system. Note that the default send and receive sizes are different on different HP systems. On the HP 3000, the default send and receive size is less than or equal to 1,024 bytes. On the HP 9000, the default send and receive size is 100 bytes.
IPCCREATE IPCDEST	**TCP protocol address** — The recommended range of TCP addresses for cross-system user applications is from 30767 to 32767 decimal (%74057 to %77777). Addresses outside of this range require privileged mode access.
IPCLOOKUP	No differences that affect cross-system operations.
IPCRECV	**Receive size** (*dlen* parameter) — Range for the HP 3000 is 1 to 30,000 bytes. Range for the HP 9000 is 1 to 32,767 bytes. Although the ranges are different, cross-system communication is not affected. If you specify a send or receive size, be sure it is within the correct range for the respective system. **Data wait flag** — The HP 9000 IPCRECV call supports a “DATA_WAIT” flag. This flag, when set, specifies that the call will not complete until the amount of data specified by the *dlen* parameter has been received. This flag is not available on the HP 3000, meaning that the call may complete before all the data is received. However, the HP 3000 IPCRECV supports other flags such as the “more data” and “destroy data” flags. Refer to the description of IPCRECV in Chapter 3 , “NetIPC Intrinsics,” for more information.

Table 2-4 Cross-System Calls (HP 3000 — HP 9000

NetIPC Call	Cross-System Consideration
IPCRECVCN	**Checksumming** — When the `ipcrecvcn()` call is executed on the HP 9000 node, checksumming is always enabled.
	Send and receive sizes — The HP 3000 send and receive size range is 1 to 30,000 bytes. The HP 9000 send and receive size range is 1 to 32,767 bytes. Although the ranges are different, cross-system communication is not affected. If you specify a send or receive size, be sure it is within the correct range for the respective system. Note that the default send and receive sizes are different on different HP systems. On the HP 3000, the default send and receive size is less than or equal to 1,024 bytes. On the HP 9000, the default send and receive size is 100 bytes.
IPCSEND	**Send size** — The HP 3000 send size range is 1 to 30,000 bytes. The HP 9000 send size is 32,767 bytes, although the ranges are different, cross-system communication is not affected. If you specify a send or receive size, be sure it is within the correct range for the respective system. Note that the urgent data bit is not supported on the HP 9000; however, if this bit is set by the HP 3000 program, it will be ignored by the receiving process on the HP 9000. For differences in send and receive size see the discussion for IPCRECVCN.
IPCSHUTDOWN	Socket shut down — The HP 3000 provides a graceful release flag that is not available on the HP 9000. If the graceful release flag (*flag 17*) is set on the HP 3000, the HP 9000 will respond as though it were a normal shutdown. The HP 3000 does not support shared sockets; the HP 9000 does. Shared sockets are destroyed only when the descriptor being released is the sole descriptor for the socket. Therefore, the HP 9000 process may take longer to close the connection than expected.

HP 3000 to PC NetIPC

The NetIPC calls affecting cross-system communication with the remote process have the following differences: checksumming, send and receive sizes, range of permitted TCP protocol addresses for users, and socket sharing. Table 2-5 lists the NetIPC calls affecting the remote process and summarizes the cross-system considerations.

Table 2-5 Cross-System Calls (HP 3000 — PC)

NetIPC Call	Cross-System Considerations
IPCCONNECT	**Checksumming** — With PC NetIPC, the TCP checksum option cannot be turned on. But if the HP 3000 requires it, the TCP checksum is in effect on both sides of the connection. **Send and receive sizes** — The HP 3000 send and receive size range is 1 to 30,000 bytes. The PC send and receive size range is 1 to 65,535 bytes. Although the ranges are different, cross-system communication is not affected. If you specify a send or receive size, be sure it is within the correct range for the respective system. For example, if the PC node sends 60,000 bytes, the HP 3000 node can call IPCRECV twice, receiving the first 30,000 bytes the first time and the second 30,000 bytes the second time. Note that the default send and receive sizes are different on different HP systems. On the HP 3000, the default send and receive size is less than or equal to 1,024 bytes.
IPCCREATE IPCDEST	**TCP protocol address** — The recommended range of TCP addresses for cross-system user applications is from 30767 to 32767 decimal (%74057 to %77777).
IPCRECV	**Receive size** (*dlen* parameter) — Range for the HP 3000 is 1 to 30,000 bytes. The PC send and receive size is 1 to 65,535 bytes. Although the ranges are different, cross-system communication is not affected. If you specify a send or receive size, be sure it is within the correct range for the respective system. On the PC, you can specify the maximum receive size of the data buffer through the *got array* in the IPCCONNECT call. This determines what the maximum value for *dlen* can be for any IPCRECV call. PC NetIPC has no option array defined in IPCCONNECT. This does not affect cross-system communication. The maximum receive size of the data in the buffer on the HP 3000 will determine the receive size buffer on the PC.

Table 2-5 Cross-System Calls (HP 3000 — PC)

NetIPC Call	Cross-System Considerations
IPCRECVCN	**Checksumming** — With PC NetIPC, the TCP checksum option cannot be turned on. But if the HP 3000 requires it, the TCP checksum is in effect on both sides of the connection. **Send and receive sizes** — The HP 3000 send and receive size range is 1 to 30,000 bytes. The PC send and receive size range is 1 to 65,535 bytes. Although the ranges are different, cross-system communication is not affected. If you specify a send or receive size, be sure it is within the correct range for the respective system. For example, if the PC node sends 60,000 bytes, the HP 3000 node can call IPCRECV twice, receiving 30,000 bytes the first time and the second 30,000 bytes the second time. Note that the default send and receive sizes are different on different HP systems. On the HP 3000, the default send and receive size is less than or equal to 1,024 bytes.
IPCSEND	**Send size** — The PC send and receive size range is 1 to 65,635 bytes. Although the ranges are different, cross-system communication is not affected. If you specify a send or receive size, be sure it is within the correct range for the respective system. On the PC, you can specify the maximum receive size of the data buffer through the *got array* in the IPCCONNECT call. This determines what the maximum value the *dlen* parameter can be for any IPCRECV call. PC NetIPC has no option array defined for IPCCONNECT. This does not affect cross-system communication. The maximum receive size of the data in the buffer on the HP 3000 will determine the receive size buffer on the PC.
IPCSHUTDOWN	**Socket shut down** — The HP 3000 provides a graceful release flag that is not available on the PC. If the graceful release flag (*flags 17*) is set on the HP 3000, the PC will respond as though it were a normal shutdown.

NetIPC Error Codes

NetIPC calls with the same names on different systems may return different error codes. Refer to the corresponding NetIPC documentation of your system for a complete list of the NetIPC error codes that are applicable to your implementation.

Program Startup

NetIPC itself does not include a call to schedule a peer process. In programs communicating between multiple HP 3000 systems, you can use the **Remote Process Management (RPM)** call, RPMCREATE, to programmatically schedule program execution. However, RPM between HP 3000 and HP 1000 systems, and between HP 3000 and HP 9000 systems, is not currently supported by Hewlett-Packard. Instead, you must manually start up each NetIPC program on its respective system.

HP 3000 Program Startup

To manually start up an HP 3000 NetIPC program, log on to the HP 3000 and run the NetIPC program (with the RUN command).

You can schedule the program to start at a particular time by writing a job file to execute the program, and then including time and date parameters in the :STREAM command that executes the job file.

HP 1000 Program Startup

To manually start up an HP 1000 NetIPC program, simply logon to the HP 1000 system and run the NetIPC program with the RTE XQ (run program without wait) command.

To have the NetIPC program execute at system start up, put the RTE XQ command in the WELCOME file.

HP 9000 Program Startup

Remote HP 9000 processes can be manually started or can be scheduled by daemons that are started at system start up. In HP-UX a daemon is a process that runs continuously and usually performs system administrative tasks. Although a daemon runs continuously, it performs actions either at a specified time, or upon a specified event.

To manually start up a NetIPC program, simply logon to the HP 9000 system and run the NetIPC program. HP recommends that you write a NetIPC daemon to schedule your NetIPC programs. You can start the daemon at start up by invoking it from the /etc/netlinkrc file. Refer to the *NS/ARPA Services/9000 Series 800 Node Manager's Guide* for more information about this file and system start up.

PC NetIPC Program Startup

To manually start up a PC NetIPC program, enter the NetIPC program name at the MS-DOS prompt.

To execute from within MS-Windows, copy the NetIPC program files to your Windows directory and double click the mouse on the executable file.

3 NetIPC Intrinsics

The information contained in this chapter is organized as follows:.

- **Programming Considerations:** describes programming considerations for using NetIPC on an MPE XL computer system.
- **Common Parameters:** provides details about the structure of the *flags*, *opt*, *data*, and *result* parameters which are common to many of the NetIPC calls.
- **NetIPC intrinsics:** a table summarizing the functions of each NetIPC intrinsic is included followed by reference pages in alphabetical order for each of the NetIPC calls.

Programming Considerations

Compatibility vs. Native Mode

Compatibility mode allows you to run application programs compiled on an MPE V computer system on an MPE XL computer system without change. Native mode refers to application programs compiled and executed on an MPE XL computer system.

NetIPC applications written for MPE V based HP 3000s can be migrated to MPE XL HP 3000s (series 900s) and run in compatibility mode as follows. On the MPE-V system, use the `MPE STORE` command to save your program's object code. On the MPE XL system, use the `MPE RESTORE` command on your object code.

To take advantage of the optimizing compilers and improved performance on the XL, you must recompile your application program on the MPE XL system that will execute in native mode (NM). Some applications contain code that must be altered before migrating to native mode.

Application migration considerations are documented in the *Application Migration Guide*.

Example 2 in Chapter 4 , "NetIPC Examples," shows the differences in declarations required for compiling a NetIPC program in compatibility mode and in native mode.

Option Variable

Many of the NetIPC intrinsics are option variable meaning they can be called with a variable number of parameters. Required parameters are listed in the discussion of each intrinsic. If you omit an optional parameter, the comma delimiter (,) is required to preserve parameter position.

For example, a call using `IPCCONNECT` could be entered as follows:

```
IPCCONNECT(CALLDESC, DEST, , , VCDESC, RESULT)
```

In this example, note that following the parameter `DEST`, commas delimit the omitted optional parameters *`flags`* and *`opt`*.

Syntax

The syntax description provided for each NetIPC intrinsic is the syntax required for Pascal programs. Differences in parameter declarations for other languages (if any) are documented in Appendix E , "C Program Language Considerations."

Capabilities

Some NetIPC intrinsics require special capabilities if you use the functions described below.

User-specified Protocol Addressing

NetIPC intrinsics IPCCONNECT, IPCCREATE, and IPCDEST allow you to specify protocol relative addresses. Addresses in the range 30767 to 32767 decimal (%74057 to %77777) can be used without special capabilities. In privileged programs you can specify protocol relative addresses between 1 and 30766 decimal (%1 and %74056).

NOTE The protocol relative address range 1 to 30766 decimal, (%1 to %74056) is administered by HP. Contact your HP representative before using an address within this reserved range.

X.25 Catch-all Socket

Using access to X.25 (level 3), network administrator (NA) capability is required to create a catch-all socket for an X.25 network. NA capability is required to run a program that creates a catch-all socket.

Common Parameters

The *flags*, *opt*, *data*, and *result* parameters are common to many NetIPC intrinsics. Remote Process Management intrinsics also use these parameters, with the exception of the *data* parameter. The following discussion of these parameters may help to clarify the more condensed information given under each intrinsic.

Flags Parameter

The *flags* parameter is a bit representation, 32 bits long, of various options. Normally an option is invoked if the appropriate bit is on (that is, set equal to 1). Borrowing Pascal-type syntax, we shall use *flags*[0] to refer to the high order bit in the parameter, *flags*[31] to refer to the low order bit, and a similar designation to refer to each of the bits in between. Bits which are not defined for a given intrinsic must be off (zero).

Opt Parameter

The *opt* parameter, which denotes various options, contains an integer code for each option along with associated information. It is not necessary to know the internal structure of this parameter in order to use it. Several *opt* parameter manipulation intrinsics have been provided to enable you to add option information without concerning yourself with the parameter's structure. However, a knowledge of the structure of the *opt* parameter can help you to determine an appropriate size for the array.

The *opt* parameter must be defined as a byte array or as a record structured in the manner described below. If your program is written in a language which supports dynamically allocated arrays, the OPTOVERHEAD intrinsic may be used to determine the size of the array.

The *opt* parameter consists of these fields as shown in Figure 3-1.

- **length**, in bytes, of option entries and data (2-byte integer);
- **number of entries** (2-byte integer);
- **option entries** (8 bytes per option entry). Each 8-byte option entry, in turn, consists of the following fields:
  - — **option cod**e (2-byte integer);
  - — **offset** (2-byte integer) byte offset relative to the base address of the *opt* parameter indicating the location of the data for this option entry;
  - — **data length**, in bytes(2-byte integer);

— **Reserved** (2 bytes).

- data associated with the option entries (variable length).

If the parameter is declared as a simple byte array, it must be large enough to contain 4 bytes for the first two fixed-length fields, 8 bytes for each option entry, plus the actual data. That is:

```
4  +  8 * numentries  +  datalength
```

NOTE Use of certain *opt* parameter options may result in the loss of portability between different Hewlett-Packard systems.

Figure 3-1 OPT Parameter Structure

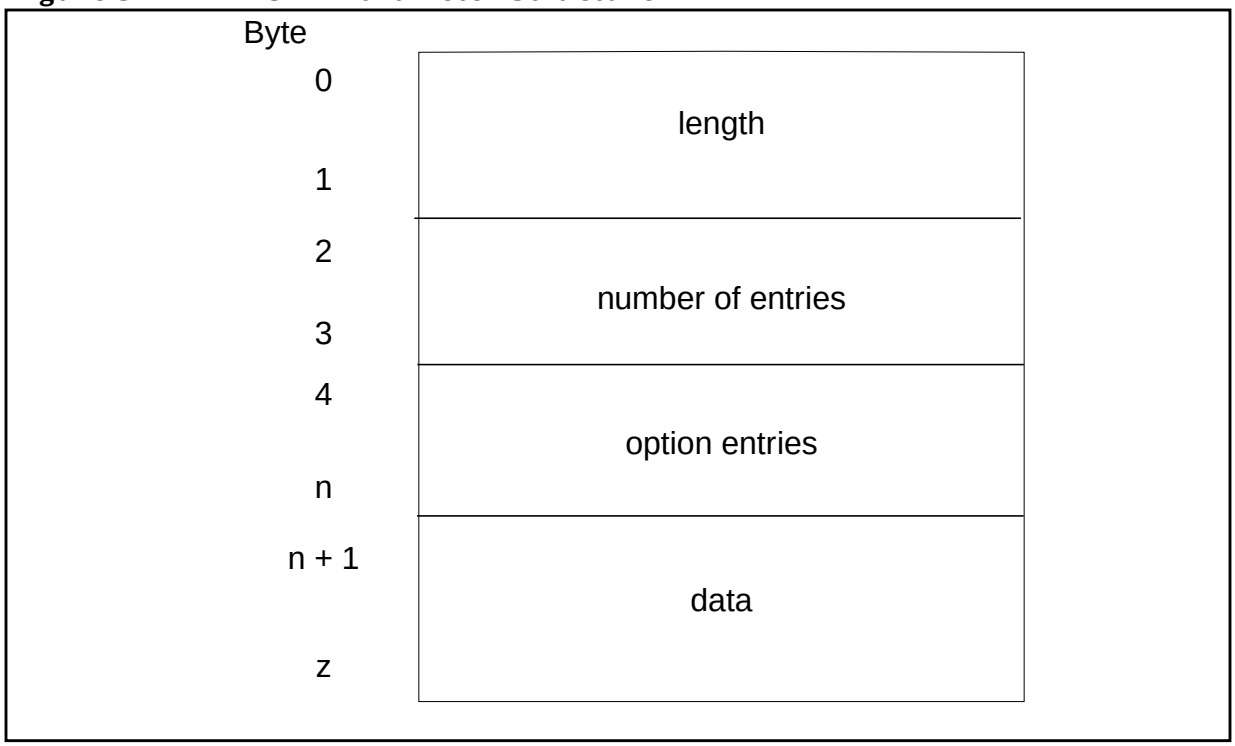
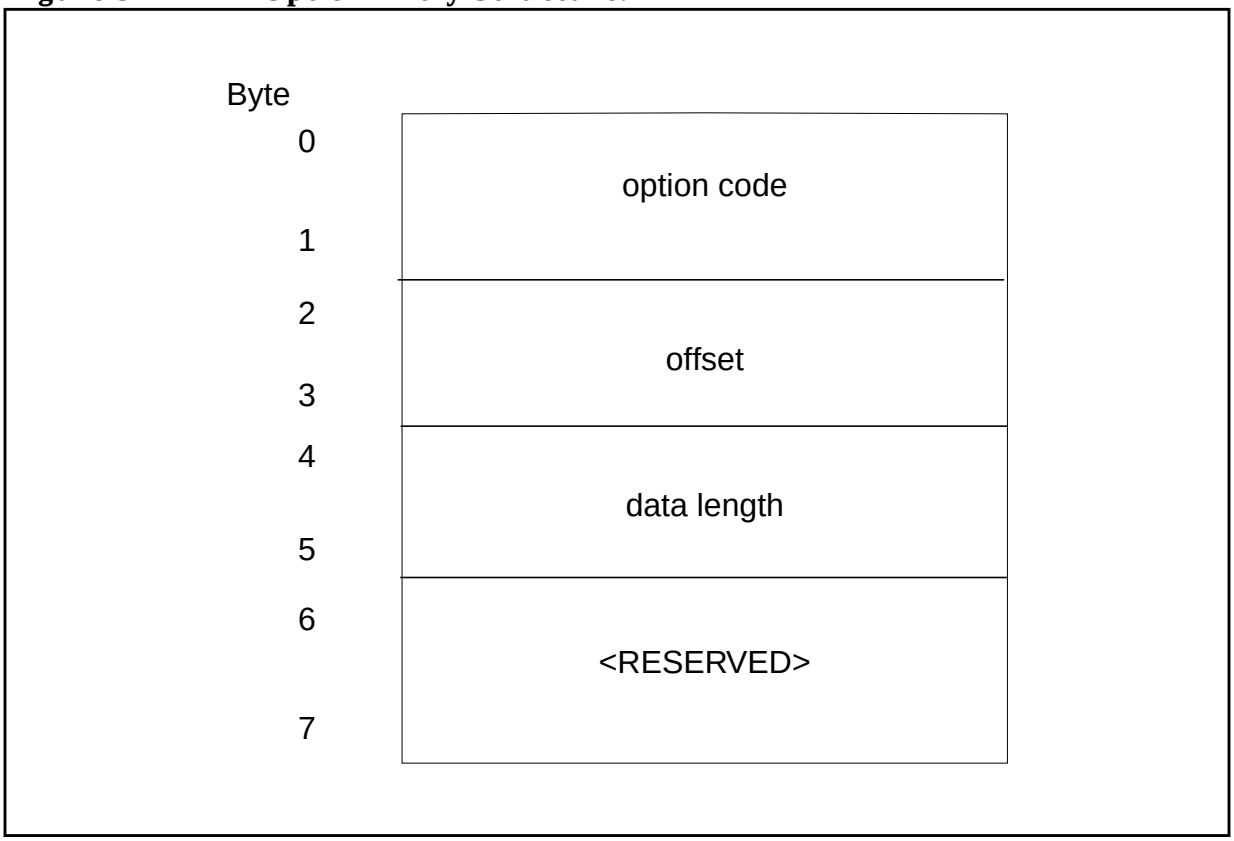


Figure 3-2 shows the structure of an option entry.

Figure 3-2 Option Entry Structure.



Data Parameter

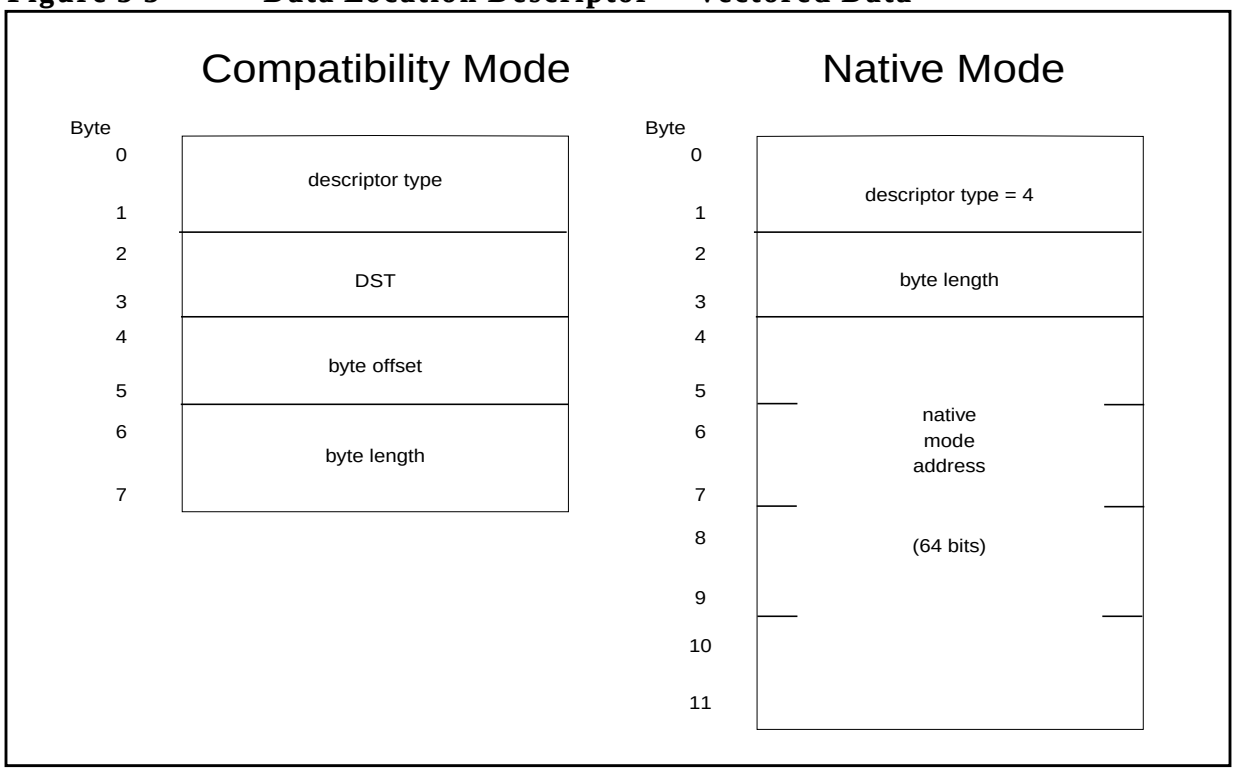
The *data* parameter is defined as a byte array. It can be in one of two formats: either holding the actual data or in a vectored format. In the case of vectored data, the *data* parameter does not contain actual data but rather the addresses from or to which the data will be gathered or distributed.

The addresses of the data are represented by data location descriptors. For all intrinsics supporting vectored data, a maximum of two data location descriptors is permitted.

For vectored data, the parameter must be a record structured as described below. The format of a compatibility mode data location descriptor is different than a native mode data location descriptor as shown in Figure 3-3.

Combining a compatibility mode vector with a native mode vector in the same NetIPC call is not supported.

Figure 3-3 Data Location Descriptor — Vectored Data



Compatibility Mode

In compatibility mode, the data location descriptor is eight bytes long and consists of four 2-byte fields. If the vector points to the stack, that is, type = 0, the value you input into the Data Segment Table (DST) is zero (DST = 0).

A compatibility mode data location descriptor consists of the following fields:

- **descriptor type** (represented by a 2-byte integer); The compatibility mode descriptor type field can have one of the following values:

  0 — (Stack) the offset is a DB-relative byte address on the calling process's data stack (the DST is ignored);

  1 — (DST Index) DST is the logical index number returned by the MPE XL HPGETDSEG intrinsic;

  2 — (DST Number) DST is an actual data segment number. All data segment references, that is, calls in split-stack mode, require privileged mode.

- **DST** (data segment table) number or index;
- **byte offset** indicating the start of the data to be transmitted. (This would be the byte count from DB on the calling process's stack or segment relative from the extra data segment (XDS)).

- **byte length** of the vectored data.

Native Mode

In native mode, the data location descriptor is 12-bytes long. It contains two 2-byte fields and a 64-bit native mode address. The DST field is unnecessary because DSTs do not exist in native mode. The three fields in the native mode vectored data descriptor are:

- **descriptor type** (2-byte integer); in this field, only Type = 4 for NM addressing is valid;
- **byte length** of the data;
- **native mode address** (64-bits long); this is a long pointer (in Pascal terminology) to the data.

Example:

```
nm_address:= globalanyptr(addr(data));
```

DLEN Parameter

In both compatibility and native mode, the *dlen* parameter indicates the full length of the *data* parameter. If the data are vectored, *dlen* must give the total length of each data location descriptor, not the length of the actual data buffer. For example, the length of a single or double vectored descriptor would be 8 or 16 bytes in compatibility mode; and 12 or 24 bytes for native mode.

Result Parameter

If a NetIPC (or Remote Process Management) intrinsic call is successful, the *result* parameter returns a value of zero. Otherwise the value returned represents a NetIPC error code (SOCKERR). NetIPC error messages are listed in Appendix A , “IPC Interpreter (IPCINT),”of this manual. You can also obtain the corresponding error message by calling IPCERRMSG.

NOTE When nowait I/O is used, the *result* parameter is not updated upon completion of an intrinsic. Therefore, the value of *result* only indicates whether the call was successfully initiated. To determine if the call completed successfully, you can use the IPCCHECK intrinsic immediately afterward.

Condition Codes

On an HP 3000, NetIPC intrinsics cause MPE condition codes to be set. CCE indicates successful completion, CCL indicates failure, and CCG is either not used or represents a warning.

Summary of NetIPC Intrinsics

Table 3-1 lists and summarizes the function of each of the NetIPC intrinsics.

Table 3-1 NetIPC Intrinsics

Intrinsic	Function
ADDOPT	Adds an option entry to the *opt* parameter.
INITOPT	Initializes the *opt* parameter so that entries may be added.
IPCCHECK	Returns the number of the last recorded NetIPC error for a call or VC socket.
IPCCONNECT	Requests a connection (a virtual circuit) to another process; returns a VC socket descriptor for a VC socket belonging to the calling process.
IPCCONTROL	Performs special operations such as enabling nowait I/O, and enabling user-level tracing.
IPCCREATE	Creates a call socket for the calling process.
IPCDEST	Returns a destination descriptor which the calling process can use to establish a connection to another process.
IPCERRMSG	Returns the IPC error message corresponding to a given error code.
IPCGET	Enables the calling process to obtain a VC socket or call socket that has been given away by another process.
IPCGIVE	Gives away a VC socket or call socket, thereby allowing another process to obtain it.
IPCLOOKUP	Returns a destination descriptor associated with a given socket name.
IPCNAME	Specifies a name for a call socket, thereby enabling other processes to obtain access to that socket.
IPCNAMERASE	Deletes a call socket name from the call socket registry.
IPCRECV	Receives the reply to a connection request, thereby establishing the connection, or receives data on an already established connection.
IPCRECVCN	Receives a connection request from another process; returns a VC socket descriptor for a VC socket belonging to the calling process.
IPCSEND	Sends data on a connection.
IPCSHUTDOWN	Releases a socket descriptor and any resources associated with it.
OPTOVERHEAD	Returns the number of bytes needed for the *opt* (option) parameter, a parameter common to many IPC intrinsics
READOPT	Allows the user to read an entry from the *opt* array.

NetIPC Reference Pages

The following pages provide syntax and usage information for each of the NetIPC intrinsics. The reference pages are organized alphabetically by NetIPC intrinsic name.

ADDOPT

Adds an option entry to the *opt* parameter.

Syntax

```
ADDOPT    (opt,entrynum,optioncode,datalength,data[,result])
```

Parameters

opt (input/output)

Record or byte array, by reference. The *opt* parameter to which you want to add an entry. Refer to "Common Parameters" for more information on the structure of this parameter.

entrynum (input)

16-bit integer, by value. Indicates which entry is to be initialized. The first entry is entry zero.

optioncode (input)

16-bit integer, by value. The option code of the entry, identifies the option.

datalength (input)

16-bit integer, by value. The length (in bytes) of the data associated with the option.

data (input)

Byte array, by reference. The data associated with the option.

result (output)

16-bit integer, by reference. The error code returned; zero if no error.

Description

The ADDOPT intrinsic specifies the values of an *opt* parameter's option entry fields and adds any associated data. The intrinsic also updates the size of the *opt* parameter.

The parameter must be initialized by INITOPT before options are added by ADDOPT. Consider this program fragment:

```
data_offset:=10;
                {10 bytes from beginning of data array}
INITOPT (opt,1, result);
                {one option entry}
ADDOPT (opt, 0, 8, 2, data_offset,result);
                {first entry is entry zero, option code 8; entry's data
                area contains a 2-byte integer specifying an offset from
                data parameter address}
IPCSEND (cd, data, dlen,flags,opt, result);
                {sends data located at offset from data address specified
                in opt}
```

INITOPT and ADDOPT allow you to initialize the *opt* parameter for use in another intrinsic. These auxiliary intrinsics make the structure of the *opt* parameter largely transparent.

Condition codes returned by ADDOPT are:

- CCE — Succeeded.
- CCL — Failed because of a user error.
- CCG — Not returned by this intrinsic.

This intrinsic can be called in split stack mode.

INITOPT

Initializes the *opt* parameter so that entries may be added.

Syntax

```
INITOPT  (opt,eventualentries,[result])
```

Parameters

opt (output)

Record or byte array, by reference. The *opt* parameter which is to be initialized. Refer to "Common Parameters" for more information on the structure of this parameter.

eventualentries (input)

16-bit integer, by value. The number of option entries that are to be placed in the *opt* parameter.

result (output)

16-bit integer, by reference. The error code returned; zero if no error.

Description

The INITOPT intrinsic initializes the length and number-of-entries fields (that is, the first 4 bytes) of the *opt* parameter. This must be done before options are added to the parameter by means of the ADDOPT intrinsic.

Condition codes returned by this intrinsic are:

- CCE — Succeeded.
- CCL — Failed because of a user error.
- CCG — Not returned by this intrinsic.

This intrinsic can be called in split stack mode.

IPCCHECK

Returns the number of the last applicable error.

Syntax

```
IPCCHECK  (descriptor [,ipcerr][,pmerr][,result]
```

Parameters

descriptor (input)

32-bit integer, by value. The call socket or VC socket descriptor for which the error is to be reported. A zero value indicates the last call socket or VC socket descriptor referenced.

ipcerr (output)

32-bit integer, by reference. The error code of the last recorded NetIPC error.

pmerr (output)

32-bit integer, by reference. The error code of the last recorded protocol module (i.e., the Transmission Control Protocol (TCP) or X.25 protocol).

result (output)

32-bit integer, by reference. The error code returned for this intrinsic call (not the previously recorded error). A zero value indicates no error.

Description

The IPCCHECK intrinsic returns the last recorded NetIPC and/or protocol module error for a given call socket or VC socket (that is, the VC socket descriptor at the calling process's end). If the descriptor value is zero, the most recent error applicable to the last call or VC socket referenced is returned. The descriptor is the only required parameter (option variable).

Condition codes returned by this intrinsic are:

- CCE — The intrinsic call was successful.
- CCL — Unsuccessful.
- CCG — Unsuccessful. The intrinsic could not return the error code because the calling process does not have access to the NetIPC data structure which retains error codes.

This intrinsic can be called in split stack mode.

IPCCONNECT

Requests a connection to another process.

Syntax

```
IPCCONNECT  ([calldesc],destdesc[,flags][,opt],vcdesc[,result]
```

Parameters

calldesc (input)

32-bit integer, by value. A call socket descriptor for a call socket belonging to this process. Required for X.25 level 3 access. For TCP access, if -1, or if omitted, a call socket is created temporarily to establish the connection.

destdesc (input)

32-bit integer, by value. Destination descriptor. Describes the location of the named call socket. (this is the call socket to which the connection request will be sent). A destination descriptor can be obtained by calling IPCDEST. For TCP access you can also obtain a destination descriptor by calling IPCLOOKUP.

flags (input)

32 bits, by reference. A bit representation of various options. No flags are defined for access to the X.25 protocol. The following flag bits are defined:

- **flags [0]** (input). (TCP only.) Makes the connection a "protected" one. A protected connection is one which only privileged users may establish or use.
- **flags [21] (input)**. (TCP only.) Enables checksumming on the Transmission Control Protocol (TCP) connection for error checking. Checksumming may also be set by the corresponding IPCRECVCN call. If either side specifies "checksum enabled" then the connection will be checksummed.

  TCP checksum may be enabled globally, over all connections, when configuring the Network Transport. Checksumming enabled by either IPCRECVCN or the network transport (remote or local) configuration overrides a 0 setting (checksum disabled) for this flag. Checksum error checking is

handled at the link level and is not normally required at the user level.

Checksumming is only used for connections between nodes and is not used for connections within the same node. Enabling checksumming may reduce network performance. Recommended value: 0.

opt (input)

Record or byte array, by reference. A list of options, with associated information. Possible options are:

- **call user data** (code=2, length=n, n bytes) (input). For access to the X.25 protocol only. This option contains data to be inserted as the call user data (CUD) field in an X.25 packet. The maximum length for the CUD is 16 bytes. With the fast select facility, the maximum length for the CUD is 128 bytes. HP has reserved the first four bytes of the CUD for protocol addressing. The user can supply data up to 12 bytes (or 124 bytes with fast select). By setting the no address flag (protocol flags option), the user can access all 16 bytes (or 128 bytes with fast select) of the CUD. See Chapter 1 , “NetIPC Fundamentals.” Access to the Call User Data (CUD) Field for more information.
- **maximum send size** (code=3, length=2; 2-byte integer). (TCP only.) This option, which must be in the range from 1 to 30,000, specifies the length of the longest message the user expects to send on this connection. The information is passed to the protocol module for internal use only. This does not mean that the user cannot send a message larger than the value that is specified in this option code. If this option is not used, the protocol module will be able to send messages at least 1024 bytes long. If the value specified is smaller than a previously specified maximum send size, then the new value is ignored.
- **maximum receive size** (code=4, length=2; 2-byte integer). (TCP only.) This option, which must be in the range from 1 to 30,000, specifies the length of the longest message the user expects to receive on this connection. The information is passed to the protocol module for calculating its transmission-window size. This does not mean that the user cannot receive messages larger than the length specified with this option code. If this option is not used, the protocol module can handle messages of at least 1024 bytes long. If the value specified is smaller than a

previously specified maximum receive size, then the new value is ignored.

- **address option** (code=128, length=2; 2-byte integer) (input). (TCP only.) This option specifies the source port address of the connection request. Addresses in the range (octal) %74057 to %77777 can be used without special capabilities.
- **facilities set name** (code=142, length=8, packed array of 8 characters) (input). For access to the X.25 protocol only. This option field is used to associate a facilities set with the virtual circuit to be created over an SVC. This option does not apply to a PVC. This is an optional parameter and defaults to the facilities set name entered while configuring the X.25 network with NMMGR on the HP 3000.
- **protocol flags** (code=144, length=4, 4-byte buffer) (input). This option contains 32 bits of protocol-specific flags. The following flags are currently defined:
  - **no address** (bit 17, input). (X.25 only.) This flag provides the user with access to the entire X.25 call user data field (16 bytes or 128 bytes with fast select). This option can be useful for communication with non-HP nodes.
- **facility field** (code=145, length=n, n bytes) (input). This option field defines the part of the facility field built using the facility set. This data must follow the X.25 recommendation and is appended to the facilities from the facility set without any change. See the discussion entitled “Facility Field” in Chapter 1 , “NetIPC Fundamentals,” for more information.

vcdesc (output)

32-bit integer, by reference. The returned VC socket descriptor, a number identifying a VC socket belonging to this process through which data can be sent or received. This descriptor can be used in other intrinsics.

result (output)

32-bit integer, by reference. The error code returned; zero if no error.

Description

The IPCCONNECT intrinsic is used to establish a VC socket (for a virtual circuit) to another process. The calling process must first create a call socket for itself and obtain the destination descriptor of a call socket belonging to the other process.

A successful result means that the connection request has been initiated. The process which requested the connection (via IPCCONNECT) must then call IPCRECV with the VC socket descriptor value in order to complete the connection. IPCCONNECT is a non-blocking call: the calling process is not blocked pending completion of its request.

Only the destination descriptor and VC socket descriptor parameters are required (that is, the intrinsic is option variable).

Condition codes returned by this intrinsic are as follows:

- CCE — Succeeded.
- CCL — Failed.
- CCG — Not returned by this intrinsic.

Protocol-Specific Considerations

The following Table 3-2 outlines parameters that are specific to the particular protocol you are accessing.

Table 3-2 IPCCONNECT Protocol Specific Parameters

Parameters	TCP	X.25
flags		
0	Protected connection	n/a
21	Enable checksum	n/a
opt		
2	n/a	Call user data (CUD)
3	Maximum send size	n/a
4	Maximum receive size	n/a
128	TCP source port address	n/a
142	n/a	Facilities set name
144	None defined	Bit 17: access to CUD
145	n/a	Facility field

X.25 Considerations

IPCCONNECT used over a switched virtual circuit causes the X.25 protocol to send a call request packet to the node and process described by the destination socket. Over a permanent virtual circuit (PVC), a reset packet is sent.

The *opt* parameter CUD field is sent as the CUD field in the call request packet. Based on the setting of the *opt protocol flags* "no address" flag, the user has access to either 12 or 16 bytes in the CUD field. With fast select, the user has access to either 124 or 128 bytes.

For communication between HP nodes, the first four bytes of the CUD field are interpreted as an address for incoming call packets (the third and fourth bytes contain the protocol relative address). The X.25 protocol uses this data to find the proper source socket to route the incoming call. This corresponds to the relative address parameter passed when the source socket was created.

Common errors returned by IPCCONNECT in *result* are:

```
SOCKERR   0  Request completed successfully.
SOCKERR  46  Unable to interpret received path
             report.
SOCKERR  55  Exceeded protocol module's limit.
SOCKERR 116  Destination unreachable.
SOCKERR 143  Invalid facilities set.
SOCKERR 155  Invalid X.25 flags.
SOCKERR 157  No virtual circuit configured.
SOCKERR 160  Incompatible with protocol state.
SOCKERR 162  X.25 permanent virtual circuit does
             not exist.
SOCKERR 163  Permanent virtual circuit already
             established.
SOCKERR 170  Error with use of the fast select facility.
SOCKERR 171  Invalid facility field opt record entry.
```

A complete table of SOCKERRs is included in Appendix C , "Error Messages."

TCP Access

If a call socket descriptor is not supplied, or if the specified value is -1, a "ghost" socket is created for the purpose of setting up the connection. This temporary socket is destroyed before the IPCCONNECT call is complete.

Cross-System Considerations for TCP

The following are cross-system programming considerations for this intrinsic:

HP 3000 to HP 1000:

Checksumming — TCP checksumming will be enabled for both sides of the connection if it is enabled by either side for HP 3000 to HP 1000 cross-system communication. On both the HP 3000 and HP 1000 checksumming can be enabled by setting bit 21 in the *flags* parameter.

Send and receive sizes — The HP 3000 send and receive size range is 1 to 30,000 bytes. The HP 1000 send and receive size range is 1 to 8,000 bytes. Although the ranges are different, specify a send size within the correct range. For example, if the HP 3000 node sends 16,000 bytes, the HP 1000 node can call IPCRECV twice, receiving the first 8,000 bytes the first time and the second 8,000 bytes the second time.

Note that the default send and receive sizes are different on different HP systems. On the HP 3000, the default send and receive size is less than or equal to 1,024 bytes. On the HP 1000 the default send and receive size is 100 bytes.

HP 3000 to HP 9000:

Checksumming — When the ipcconnect() call is executed on the HP 9000 node, checksumming is always enabled. Therefore checksumming is always enabled for the HP 3000-to-HP 9000 connection.

Send and receive sizes — The HP 3000 send and receive size range is 1 to 30,000 bytes. The HP 9000 send and receive size range is 1 to 32,767 bytes. Although the ranges are different, cross-system communication is not affected. If you specify a send or receive size, be sure it is within the correct range for the respective system.

Note that the default send and receive sizes are different on different HP systems. On the HP 3000, the default send and receive is less than or equal to 1,024 bytes. On the HP 9000, the default send and receive size is 100 bytes.

HP 3000 to PC NetIPC:

Checksumming — With PC NetIPC, the TCP checksum option cannot be turned on. But if the HP 3000 requires it, the TCP checksum is in effect on both sides of the connection.

Send and receive sizes — The HP 3000 send and receive size range is 1 to 30,000 bytes. The PC send and receive size range is 1 to 65,535 bytes. Although the ranges are different, cross-system communication is not affected. For example, if the PC node sends 60,000 bytes, the HP 3000 node can call IPCRECV twice, receiving the first 30,000 bytes the first time and the second 30,000 bytes the second time.

Note that the default send and receive sizes are different on different HP systems. On the HP 3000, the default send and receive size is less than or equal to 1,024 bytes.

IPCCONTROL

Performs special operations.

Syntax

```
IPCCONTROL      (descriptor,request[,wrtdata][,wlen][,readdata][,rlen]
    [,flags][,result)]
```

Parameters

descriptor (input)

32-bit integer, by value. Either a call socket descriptor or a VC socket descriptor.

request (input)

32-bit integer, by value. The value supplied indicates what control operation is to be performed.

- **1 = Enable nowait** (asynchronous) I/O for the specified call socket or VC socket descriptor.
- **2 = Disable nowait** (asynchronous) I/O for the specified call socket or VC socket descriptor; perform waited (blocking) calls only.
- **3 = Change the default timeout** (initially 60 seconds) for waited and nowait I/O (receive operations only). The *wrtdata* (two bytes) and *wlen* parameters are required with this request. The timeout is specified in tenths of a second.
- **9 = Accept a connection request** that is in the deferred state. This request is valid only over connection sockets in the connection pending state. To reject a connection request, see *request* code 15. The call must be accepted before attempting to send or receive data over the connection. No *readdata* is associated with this request.

  For this request, the *wrtdata* parameter can (optionally) contain information in the format of the *opt* parameter used in the other intrinsics. The valid codes are:

  — **code 2** — (X.25 only.) This option field contains data to be sent as user data in the call accepted or clear indication packet. The maximum length for X.25 call user data (CUD) is 16 bytes. If the fast

select facility has been used, the maximum length is 128 bytes.

— **code 145** — (X.25 only.) This option field defines the part of the facility field to be appended to the facility field built using the received facilities. The maximum length of this field is 109 bytes. This buffer has to follow the X.25 recommendation and is appended to the facility field built based on the information contained in the call request.

- **10 = Send a reset packet** (X.25 only.) This request is valid only over connection sockets. The *wrtdata* parameter (2 bytes) can contain the cause (byte 1) and diagnostic (byte 2) fields to be included in the reset packet sent by the X.25 protocol. The cause field may be overwritten by the PDN. If configured as a DTE, the cause will always be 0, irrespective of the value entered. Suggested value for the cause field is 0 (zero), DTE originated. No *readdata* is associated with this request.
- **11 = Send an interrupt packet** (X.25 only.) This request is valid only over connection sockets. The *wrtdata* parameter can contain from 1 to 32 bytes of user data that will be inserted in the interrupt packet sent by the X.25 protocol.
- **12 = Reason for error or event** (X.25 only.) This request returns the reason for the NetIPC error or event on an X.25 connection in the *readdata* parameter. The first byte of *readdata* contains the type of packet that caused the error or the unsolicited even (Clear, Reset, Interrupt). The second byte contains the length of the clear user data field or the length of the interrupt data field. The third and fourth bytes contain the cause and diagnostic fields.

  Beginning with byte 5 there can be either clear user data or interrupt data. There can be up to 128 bytes of clear user data if the type of packet is clear and if the fast select facility was used at the beginning of the communication. There can be up to 32 bytes of interrupt data if the type of packet is interrupt.

  This request is valid only over an X.25 connection socket after a communications line error has occurred. Possible cause and diagnostic codes generated by X.25 are listed in Appendix B , “Cause and Diagnostic Codes.”

The type of packets returned are:

— 10 = Clear packet received
— 11 = Reset packet received
— 12 = Interrupt packet received

If no event is reported, *readdata* contains zeros. If the error was caused by a clear packet, the connection is lost, and the user must use IPCSHUTDOWN to clear the connection. There is no *wrtdata* associated with this request.

- **13 = Set no activity timeout** (X.25 only.) This request is only valid on connection sockets. The *wrtdata* parameter contains the timeout value in minutes (16-bit positive integer). If not specified, the default value of zero will be passed to *wrtdata* disabling the timer. After a timeout, IPCSHUTDOWN must be used to remove the connection socket. There is no *readdata* associated with this request.
- **14 = Return local node name.** If this request is used, the fully-qualified local node name is returned in *readdata*.
- **15 = Reject a connection request that is in the deferred state.** The VC socket is automatically deleted after this request.

  For X.25, this request causes the protocol to send a clear packet with the cause field set to zero (DTE originated) and the diagnostic field set to 64. This request is valid only over connection sockets in the connection pending state.

  For this request, the *wrtdata* parameter can (optionally) contain information in the format of the *opt* parameter used in the other intrinsics. The valid codes are:

  — **code 2** — (X.25 only.) This option field contains data to be sent as user data in the call accepted or clear indication packet. The maximum length for X.25 call user data (CUD) is 16 bytes. If the fast select facility has been used, the maximum length is 128 bytes.
  — **code 145** — (X.25 only.) This option field defines the part of the facility field to be appended to the facility field built using the received facilities. The maximum length of this field is 109 bytes. This buffer has to follow the X.25 recommendation and is appended to the facility

field built based on the information contained in the call request.

- **256 = Enable nowait receives;** disable nowait sends.
- **257 = Enable nowait sends; disable nowait receives.**
- **258 = Abort outstanding nowait receives**.
- **259 = Enable user-level NetIPC tracing.** This request causes NetIPC intrinsic calls (both initiation and completion of I/O requests) to be traced.
  - — **code 131** — Indicates that the data portion of the *wrtdata* parameter contains the trace file name. If omitted, the trace file is named SOCK####, where #### are four randomly chosen digits, and placed in the caller's group and account.
  - — **code 132** — Indicates that the data portion of the *wrtdata* parameter contains a 2-byte value representing the number of records allotted to the trace file. If omitted, or if this value is zero, the default is 1024 records.
  - — **code 133** — Indicates that the data portion of the *wrtdata* parameter contains a 2-byte value representing the maximum number of bytes of user data which you wish to trace. If omitted, or if the value is -1, the default is 2000 bytes (a zero value means zero bytes). The largest amount of user data which may be traced is 8,192 bytes.
- **260 = Disable user-level NetIPC tracing.**
- **261 = Enable immediate acknowledgment.** (TCP only.) Instructs the TCP protocol module to acknowledge received frames immediately. *Note that use of option 261 can degrade performance of the user's processes*.
- **262 = Change the timeout for waited and no-wait sends.** The default is timeout disabled.
- **514 = Available to processes running in privileged mode only.** Over an open connection, if the previous call was an IPCCONNECT, the two byte local TCP address is returned in the *readdata* buffer. Over an open connection, if the previous call was an IPCRECVCN, two bytes of the remote TCP address and four bytes of the remote IP address are returned in the *readdata* buffer. The *rlen* parameter returns the length of *readdata*.

wrtdata (input)

Record or byte array, by reference. If the request is to change the default timeout, (*request* code 3 or 262) the value in the first two bytes of the *wrtdata* buffer will become the new timeout, in tenths of a second. A zero value indicates an indefinite timeout: a call to IOWAIT returns only after the next I/O request completes.

If the request is to enable tracing, (*request* code 259) or for X.25 requests 9 or 15, this parameter may (optionally) contain information in the same format as the *opt* parameter in other intrinsics.

wlen (input)

32-bit integer, by value. Length in bytes of the *wrtdata* parameter.

readdata (output)

Record or byte array, by reference. If the request enables tracing, the trace file's name is returned in this parameter. If the request asks for the socket's address, that address is returned here. If the request is for the local node name, the fully qualified node name is returned in *readdata*.

rlen (input/output)

32-bit integer, by reference. The maximum number of bytes that you expect to receive in the *readdata* parameter. If *readdata* returns the trace file name, *rlen* returns the length of this name, in bytes. If *readdata* returns the socket's address, *rlen* will return the byte length of the address.

flags (input)

32 bits, by reference. A bit representation of various options. The following flag is defined:

- **flags [31]** (input) — (TCP only.) If NetIPC tracing is enabled (*request* = 259), this flag indicates that Transport Layer protocol activity (headers and internal messages) should also be traced.

result (output)

32-bit integer, by reference. The error code returned; zero if no error.

Description

The IPCCONTROL intrinsic is used to perform various special operations on sockets. This intrinsic is option variable. All requests require the *descriptor* and *request* parameters except *request* 14 which requires *request* and *readdata* only. The timeout requests (code 3 or 262) also require the *wrtdata* parameter. For tracing, socket address requests, and the local node name, information may be returned in the *readdata* buffer.

Request code 3 is used to set a receive timeout value as specified in *wrtdata* (two bytes). Zero (0) may be used to indicate no timeout. The timeout value is measured in tenths of a second. The default value is 60 seconds with the timeout enabled.

Request code 3 and the no activity timeout for X.25 (code 13) are independent timers. Setting one has no influence on the other. Both timers can be operational at the same time.

Request code 262 is used to set a send timeout value as specified in *wrtdata* (two bytes). Zero (0) may be used to indicate no timeout. If timeouts are enabled, the timer will expire the number of timeout seconds (as specified in *wrtdata*) after completion of the last send. The default value is timeout NOT enabled. There is only one send timer per connection. It will be running any time there is an outstanding send. That is, if nowait I/O is used, it will run until IOWAIT completes for all sends.

For a waited send, the timer will run until the intrinsic completes. If multiple nowait sends are issued, the timer will be restarted for each send initiated and for each IOWAIT completed with sends still outstanding. If a send timer expires before a send completes, the connection must be shutdown.

Request codes 9 and 15 allow the user to accept or reject a connection that is in the deferred-connection state (see IPCRECVCN). If the connection request is accepted, the connection can receive and send data upon the completion of IPCCONTROL. If the connection is rejected, all resources allocated for the connection are returned and the requestor is notified of the rejection.

When requesting the descriptor's address (request code 514), *readdata* has the meanings shown in Table 3-3.

Table 3-3 readdata Meanings

Descriptor Type	Address Meaning
`call socket`	port address of socket (for TCP, length=2 bytes)
`connection from IPCCONNECT`	local port address of connection socket (for TCP, length=2 bytes)
`connection from IPCRECVCN`	remote port address of connection socket in bytes 0 and 1, remote internet address of node in bytes 2 through 5. (6 bytes total length)

Condition codes returned by this intrinsic are:

- CCE — Succeeded.
- CCL — Failed
- CCG — Not returned by this intrinsic.

This intrinsic may not be called in split stack mode.

Protocol-Specific Considerations

The following Table 3-4 outlines parameters that are specific to the particular protocol you are accessing.

Table 3-4 IPCCONTROL Protocol Specific Parameters

Parameters	TCP	X.25
request		
10	n/a	Send reset
11	n/a	Send interrupt
12	n/a	Reason for error or event
13	n/a	Set inactivity timeout
261	Enable immediate ack	n/a
wrtdata		
2	n/a	Call user data
145	n/a	Facility field
flags		
31	Trace transport layer protocol activity	n/a

X.25 Considerations Common errors returned by IPCCONTROL in *result* are:

```
SOCKERR   0  Request completed successfully.
SOCKERR  59  Socket timeout.
SOCKERR  65  Connection aborted by local protocol
             module.
SOCKERR  67  Connection failure detected.
SOCKERR 107  Transport is going down.
SOCKERR 160  Incompatible with protocol state.
SOCKERR 170  Error with use of fast select facility.
SOCKERR 171  Invalid facility field opt record
             entry.
```

A complete table of SOCKERRs is included in Appendix C , “Error Messages.”

IPCCREATE

Creates a call socket for the calling process.

Syntax

```
IPCCREATE  (socketkind[,protocol][,flags][,opt],calldesc[,result])
```

Parameters

socketkind (input)

32-bit integer, by value. Indicates the type of socket to be created. The only type that a user process may create is: 3 = call socket. It is used for sending and receiving connection requests.

protocol (input)

32-bit integer, by value. Indicates the protocol module which the calling process wishes to access. If the value is zero or if this parameter is not specified, the TCP module is chosen by default. The protocols currently available to user processes are:

- 0 = Default protocol. The current default is TCP. The recommended value for programs using IPCNAME and IPCLOOKUP is 0 rather than 4 for TCP.
- 2 = X.25 protocol
- 4 = TCP (Transmission Control Protocol)

flags (input)

32 bits, by reference. A bit representation of various options. The following option is defined:

- **flags [0]** (input). TCP only. Makes the newly created socket a “protected” socket. A protected socket is one which only a privileged user may create or use.

opt (input)

Record or byte array, by reference. A list of options, with associated information. Refer to “Common Parameters” for more information on the structure of this parameter. The following options are available:

- **maximum connection requests queued** (option code=6, length=2, 2-byte integer) (input). Used to specify the maximum number of unreceived

connections that can be queued to a call socket. The default value is 7.

- **address option** (option code=128, length=*n*; n-byte array), (input). Allows users to specify the socket's protocol relative address rather than having NetIPC allocate an address. The format of this address is defined by the protocol, for TCP and X.25 protocol access, the address is a 2-byte array. For X.25, you must either specify a protocol relative address, or identify the socket as catch-all. (See the *opt* protocol flags "catch-all socket flag" (bit 2) description).

  Address values in the range 30767 to 32767 decimal (% 74057 to % 77777) can be used without special capabilities. In privileged programs, values in the range 1 to 30766 decimal (%1 to % 74056) can be used. See the paragraph "User-specified Protocol Addressing" at the beginning of this chapter for more information.

- **network name** (code=140, length=8, packed array of characters) (input). The X.25 network name is the network interface (NI) name defined when the network is configured with NMMGR. This option is required for X.25 protocol access. This field is left-justified.

- **protocol flags** (code 144, length=4, 4-byte buffer).

  — **catch-all socket flag** (bit 2) (input). X.25 protocol access only. This flag identifies the socket as a catch-all socket. Network administrator (NA) capability is required to set this flag. User capability is required to run a program that creates a catch-all socket. The address option (protocol relative address) does not apply to a catch-all socket.

calldesc (output)

32-bit integer, by reference. Call socket descriptor. The socket descriptor which identifies the created socket.

result (output)

32-bit integer, by reference. The returned error code; zero if no error.

Description

The IPCCREATE intrinsic creates a call socket, returning a call socket descriptor. A call socket descriptor is an identifying number which may be used in other NetIPC intrinsic calls. A process may own a maximum of 64 (call and VC) sockets. If a socket has been given away (via the IPCGIVE intrinsic), it is included in this total until another process takes it (via IPCGET).

Only the *socketkind* and *calldesc* parameters are required.

Condition codes returned by this intrinsic are:

- CCE — Succeeded.
- CCL — Failed.
- CCG — Not returned by this intrinsic.

IPCCREATE runs in waited mode. It does not return until the request is completed.

Protocol-Specific Considerations

The following Table 3-5 outlines parameters that are specific to the particular protocol you are accessing.

Table 3-5 IPCCREATE Protocol Specific Parameters

Parameters	TCP	X.25
flags 0	Protected socket	n/a
opt 140 144	n/a none defined	NI name required Bit 2: catch-all socket flag

X.25 Considerations

For direct access to X.25, the *protocol* parameter must be 2 (X.25). The *opt* parameter network name must include the X.25 network NI name.

The *opt* parameter address option (code 128) is used to contain the protocol relative address of the source socket.

X.25 compares the protocol relative address contained in an incoming call (in the CUD field) to the protocol relative addresses assigned to all X.25 call sockets at the source sockets' destination. If the protocol relative address of the source socket matches the incoming call's address (CUD) the call is routed to that socket.

If no match is found, the incoming call is routed to the catch-all socket if one has been defined. If the CUD address does not match any of the call sockets and no catch-all socket has been defined, the incoming call is cleared. the cause field of the clear packet is set to 0 and the diagnostic is 64.

The catch-all socket can be defined by setting the *opt protocol flags* catch-all socket flag (bit 2). Only one catch-all socket can be defined per directly-connected network.

The catch-all socket and address option (protocol relative address) only apply to switched virtual circuits (SVCs).

Common errors returned by IPCCREATE in *result* are:

```
SOCKERR   0  Successful completion.
SOCKERR   4  Transport has not been initialized.
SOCKERR   9  Protocol is not active.
SOCKERR  55  Exceeded protocol module's limit.
SOCKERR 106  Address currently in use by another
             socket.
SOCKERR 107  Transport is going down.
SOCKERR 153  Socket is already in use.
```

A complete table of SOCKERRs is included in Appendix C , "Error Messages."

TCP

for TCP access, only the *socketkind* and *calldesc* parameters are required.

Cross-System Considerations for TCP

The following are HP 3000 to HP 1000, HP 3000 to HP 9000, and HP 3000 to PC programming considerations for this intrinsic:

TCP protocol address — Although the ranges of protocol addresses for each computer system are different, the recommended range for cross-system user applications is from 30767 to 32767 decimal (%74057 to %77777).

IPCDEST

Creates a destination descriptor.

Syntax

```
IPCDEST  (socketkind[,location][,locationlen],protocol,
protoaddr,protolen[,flags][,opt],destdesc[,result])
```

Parameters

socketkind (input)

32-bit integer, by value. Defines the type of socket. The only type user processes can create is: 3 = call socket.

location (input)

Character array, by reference. The name of the node (either *node* or *node.domain.organization*) on which the destination socket is to be created. If this parameter is omitted, the local node is assumed.

locationlen (input)

32-bit integer, by value. The length in bytes of the destination node name. Zero indicates that no location was given (that is, the node is local). Maximum (for a fully qualified name) is 50.

protocol (input)

32-bit integer, by value. Defines the Transport Layer protocol to be used. The protocols currently available to user processes are:

- 2 = X.25 protocol
- 4 = TCP

protoaddr (input)

Byte array, by reference. Protocol relative address (remote address) with which the socket will be associated. The format of this address, defined by the protocol, is a 2-byte array (16 bits). Nonprivileged programs must use addresses in the range 30767 to 32767 decimal (%74057 to %77777). For X.25 access to level 3, this address is included in the CUD field of an X.25 call packet. (See the discussion of IPCCONNECT for the parameters providing access to the CUD).

protolen (input)

32-bit integer, by value. The length in bytes of the protocol address.

flags

32 bits, by reference. A bit representation of various options. No flags are currently defined.

opt (input)

Record or byte array, by reference. A list of options, with associated information.

- destination network address (code = 16, length=n, n byte buffer) (input). (X.25 only.)This option allows a user to bypass the use of the network directory and associated IP address by specifying the destination node network address.

  The first two bytes of the option data field contain the "protocol" value of the module. For X.25, the protocol value is 2. The rest of the option data field is relative to that protocol. For X.25, two subformats are defined: one for the PVC numbers and one for SVC addresses. The third and fourth bytes of the data field contain a format identifier, where 1 indicates a PVC number and 0 indicates a SVC address. The PVC number is a 4 byte field. For a PVC, the total length of the option field is 8 bytes.

  The SVC field is composed of up to 16 nibbles where the first nibble is the nibble length of the remaining address. An odd number of nibbles can be passed since the length indicates the significance of the remaining field. This implies that the option data length for an SVC is between 5 and 12 bytes inclusive.

destdesc (output)

32-bit integer, by reference. Destination descriptor. Describes the location of the named call socket. May be used in subsequent NetIPC calls to `IPCCONNECT`.

result (output)

32-bit integer, by reference. The error code returned; zero if no error.

Description

The IPCDEST intrinsic creates a destination descriptor that contains routing information for sending data to another process.

This intrinsic is option variable. The required parameters are: *socketkind*, *protocol*, *protoaddr*, *protolen*, and *destdesc*.

Condition codes returned by this intrinsic are:

- CCE — Succeeded.
- CCL — Failed.
- CCG — Not returned by this intrinsic.

This intrinsic cannot be called in split stack mode.

Protocol-Specific Considerations

The following Table 3-6 outlines parameters that are specific to the particular protocol you are accessing.

Table 3-6 IPCDEST Protocol Specific Parameters

Parameters	TCP	X.25
opt 16	n/a	destination network address

X.25 Considerations

IPCDEST is used to create a destination descriptor for X.25 direct access. The *protoaddr* parameter is only used with switched virtual circuits (SVCs).

Using the destination network address (*opt* 16) to directly specify an X.25 address of an SVC or a PVC number allows the user to bypass the use of the network directory and the associated IP address.

Cross-System Considerations For TCP

The following are HP 3000 to HP 1000, HP 3000 to HP 9000, and HP 3000 to PC programming considerations for this intrinsic.

TCP protocol address — Although the ranges of protocol addresses for each computer system are different, the recommended range of TCP addresses for user applications is from 30767 to 32767 decimal (%74057 to %77777).

IPCERRMSG

Returns the NetIPC error message corresponding to a given error code.

Syntax

```
IPCERRMSG   (ipcerr,msg,len,result)
```

Parameters

ipcerr (input)

32-bit integer, by value. A valid NetIPC error code.

msg (output)

Character array, by reference. The NetIPC error message corresponding to the given error code.

len (output)

32-bit integer, by reference. The length (in bytes) of the error message. The maximum is 80 bytes.

result (output)

32-bit integer, by reference. The error code returned for this intrinsic call; zero if no error.

Description

The IPCERRMSG intrinsic returns the NetIPC error message corresponding to a given error code. It also gives the length of the message. All parameters are required.

Condition codes returned by this intrinsic are:

- CCE — Succeeded.
- CCL — Failed because of a user error.
- CCG — Failed because of an internal error (for example, unable to open the message catalog, a GENMESSAGE failure, etc.).

This intrinsic cannot be called in split stack mode.

IPCGET

Obtains a VC socket or call socket descriptor that has been given away by another process.

Syntax

```
IPCGET   (givename,nlen,flags,descriptor,result)
```

Parameters

givename (input)

Character array, by reference. The temporary name assigned to the socket when it was given away. It is up to 16 characters long.

nlen (input)

32-bit integer, by value. The length in bytes of the specified name.

flags

32 bits, by reference. A bit representation of various options. No flags are currently defined for this intrinsic.

descriptor (output)

32-bit integer, by reference. The VC socket or call socket descriptor that was given away via the IPCGIVE command.

result (output)

32-bit integer, by reference. The error code returned; zero if no error.

Description

The IPCGET intrinsic allows a process to obtain ownership of a VC socket or call socket descriptor that has been relinquished by another process through the IPCGIVE intrinsic. A temporary name identifies the socket for the process which wishes to acquire it. All the parameters are required.

Condition codes returned by this intrinsic are:

- CCE — Succeeded.
- CCL — Failed.
- CCG — Not returned by this intrinsic.

This intrinsic cannot be called in split stack mode.

IPCGIVE

Gives away a VC socket or call socket descriptor, making it available for use by other processes.

Syntax

```
IPCGIVE  (descriptor,givename,nlen,flags,result)
```

Parameters

descriptor (input)

32-bit integer, by value. The VC socket or call socket descriptor to be given away.

givename (input/output)

Character array, by reference. A name which will be temporarily assigned to the specified socket. The process which obtains the socket must request it by this name. If the *nlen* (name length) parameter is zero, an 8-character name is randomly assigned and returned in the *givename* parameter. If the name is supplied by the user, it must be no longer than 16 characters.

nlen (input)

32-bit integer, by value. Length in bytes of the specified name. If the value is zero, the NetIPC facility will assign the name.

flags

32 bits, by reference. A bit representation of various options. No flags are currently defined for this intrinsic.

result (output)

32-bit integer, by reference. The error code returned; zero if no error.

Description

A process can invoke IPCGIVE to give away a VC socket or call socket descriptor that it owns. Another process at the same node must then “get” the descriptor in order to use it. For example, Process A at node X can give away a VC socket descriptor. Process B, also at node X, may get the descriptor and send data over the connection that Process A has previously established with process C at node Z. Because Process B “got” the endpoint of a previously established connection, it does not

need to create its own call socket and engage in the NetIPC connection dialogue in order to communicate with Process C.

All the parameters are required.

When a socket is given away, it is assigned a new, temporary name. This name is either specified by the user or assigned by the NetIPC facility. It continues to exist only until the socket is obtained by another process or destroyed. The other process uses this name in a call to `IPCGET`, not `IPCLOOKUP`. However, the syntax of the name is the same as it is for other intrinsics permitting socket name parameters. Therefore it is possible to use a socket's "well-known" name — a name bound to the socket and known to other processes — in the `IPCGIVE` and `IPCGET` intrinsics.

Once a process has given away a socket, it no longer has access to the VC socket (or call socket) descriptor specified. If a process expires after giving away a socket, and no other process has obtained it, the VC socket or call socket will be destroyed.

Also, after a socket has been given away, it is the responsibility of the new owning process to tell other processes that the socket has been acquired. Other processes will then know who is receiving the data they send.

Condition codes returned by this intrinsic are:

- CCE — Succeeded.
- CCL — Failed.
- CCG — Not returned by this intrinsic

This intrinsic cannot be called in split stack mode.

IPCLOOKUP

Obtains a destination descriptor for a named call socket. Use with TCP access only.

Syntax

```
IPCLOOKUP      (socketname,nlen[,location][,loclen][,flags],destdesc
        [,protocol][,socketkind][,result])
```

Parameters

socketname (input)

Character array, by reference. The name of the socket.

nlen (input)

32-bit integer, by value. The length in bytes of the specified socket name. Maximum is 16.

location (input)

Character array, by reference. An environment ID or node name indicating where the socket registry search is to take place. The domain and organization names which fully qualify the node/environment designation are optional. If no location is specified, the local socket registry is searched. This parameter can be a maximum of 50 characters long.

loclen (input)

32-bit integer, by value. The length in bytes of the location parameter. A zero value indicates that the socket registry search is to take place on the local node.

flags (input)

32 bits, by reference. A bit representation of various options. The only flag defined is: flags [0]. It causes the destination descriptor to be “protected.” A protected destination descriptor is one which only privileged users may create or use.

destdesc (output)

32-bit integer, by reference. The returned destination descriptor, which the calling process may use to access the named socket as a destination. This descriptor is required by the IPCCONNECT intrinsic.

protocol (output)

32-bit integer, by reference. A number identifying the protocol module with which the socket is associated: The only protocol available to user processes is: 4 = TCP.

socketkind (output)

32-bit integer, by reference. A number which identifies the socket's type: 3 = call.

result (output)

32-bit integer, by reference. The error code returned; zero if no error.

Description

The IPCLOOKUP intrinsic is used to gain access to a named socket. When supplied with the socket's name, it returns a destination descriptor that the calling process can use in order to connect to and send messages to that socket. It is important to synchronize the naming and lookup of sockets so that the naming occurs before the lookup. If these two events are occurring concurrently, you can repeat the IPCLOOKUP call, checking the result parameter after each call, until the call is successful. If the result value is 37 ("NAME NOT FOUND"), the socket has not yet been given the name. The following Pascal program fragment illustrates this idea:

```
socketname := 'RAINBOW';
location := 'SOMEWHERE';
result := 0;
repeat
 IPCLOOKUP (socketname, 7, location, 9, ,
  destdesc, , , result);
until result<>37;
if result<>0 then ERRORPROCEDURE;
```

The only required parameters in the IPCLOOKUP intrinsic are *socketname*, *nlen*, and *destdesc*. This intrinsic is option variable. Condition codes returned by this intrinsic are:

- CCE — Succeeded.
- CCL — Failed.
- CCG — Not returned by this intrinsic.

This intrinsic cannot be called in split stack mode.

IPCNAME

Associates a name with a call socket descriptor.

Syntax

```
IPCNAME  (calldesc,socketname,nlen,result)
```

Parameters

calldesc (input)

32-bit integer, by value. The call socket descriptor to be named.

socketname (input/output)

Character array, by reference. The name (maximum of 16 characters) to be assigned to the socket. If the *nlen* (name length) parameter is zero, an 8-character name is randomly assigned and returned in the *givename* parameter. If the name is supplied by the user, it must be no longer than 16 characters.

nlen (input)

32-bit integer, by value. The length in bytes of the specified socket name. Maximum is 16.

result (output)

32-bit integer, by reference. The error code returned; zero if no error.

Description

The IPCNAME intrinsic allows a user to bind a name to a call socket. Using the IPCLOOKUP intrinsic, another process can obtain access to the socket by means of its name. A single call socket on an HP 3000 can have a maximum of 4 names. VC sockets cannot be named. If the specified name length is zero, an 8-character name will be randomly generated and returned in the socketname parameter. When the socket is destroyed, the name will be removed from the socket registry.

All parameters are required. Condition codes returned by this intrinsic are:

- CCE — Succeeded.
- CCL — Failed.
- CCG — Not returned by this intrinsic.

This intrinsic cannot be called in split stack mode.

IPCNAMERASE

Deletes a name associated with a call socket descriptor.

Syntax

```
IPCNAMERASE  (socketname,nlen,result)
```

Parameters

socketname (input)

Character array, by reference. The socket name, bound to a socket, which is to be removed.

nlen (input)

32-bit integer, by value. The length in bytes of the specified socket name. Maximum is 16.

result (output)

32-bit integer, by reference. The error code returned; zero if no error.

Description

If a socket has been named with the IPCNAME intrinsic, the owner of the socket may remove the name by means of the IPCNAMERASE intrinsic. The owner is the process which created the socket or, if the socket has been given away, the process which has acquired it.

All the parameters are required. Condition codes returned by this intrinsic are:

- CCE — Succeeded.
- CCL — Failed.
- CCG — Not returned by this intrinsic.

This intrinsic cannot be called in split stack mode.

IPCRECV

Receives a response to a connection request, thereby completing a connection, or receives data on an existing connection.

Syntax

```
IPCRECV  (vcdesc[,data][,dlen][,flags][,opt][result]
```

Parameters

vcdesc (input

32-bit integer, by value. The VC socket descriptor, a number identifying the VC socket belonging to this process through which the data will be received.

data (output)

Record or byte array, by reference. A buffer to hold the received data or a list of data descriptors (maximum of two) indicating where the data are to be distributed. For programming in "C" language, see Appendix E , "C Program Language Considerations."

dlen (input/output)

32-bit integer, by reference. Gives the maximum number of bytes you are willing to receive. For a response to a connection request, this value must be 0 (or the parameter may be omitted). For actual data on an established connection, the value must be between 1 and 30,000. The returned value indicates how many bytes were actually received.

In one IPCRECV call, you can receive all the accumulated data from one or more IPCSEND calls, but you may also receive only part of the data "sent" in an IPCSEND. In order to obtain the rest of the data, you can issue another IPCRECV call.

flags (input/output)

32 bits, by reference. A bit representation of various options. The following options are defined:

- **flags [16]** — no output (input). If nowait I/O is used and this bit is set, the *flags* parameter will not be updated upon completion of this IPCRECV. This allows a calling procedure to have a local *flags* parameter and still complete before the IPCRECV

completes. This flag has no effect if waited I/O is used.

- **flags [25]** — discarded (output). (X.25 only.) This flag indicates that the call user data, or the facility field were present, but that some or all had to be discarded. This can occur if no call user data receive option was specified or if either field is too short to hold all of the data.
- f**lags [26]** — more data (output). This flag indicates that there may be more data to be received after completion of this IPCRECV.

  For TCP, this bit will always be set when normal, non-urgent data has been received because TCP sends data in stream mode, with no end-of-data indication. However, if urgent data has been received, and no more is pending, this bit will be set to 0.

  For X.25, the "more data" flag indicates that the data returned is not the complete message. This will only occur if the user request was for a smaller message than was sent. The amount of data specified in *dlen* has been moved into data. The following part of the message will be returned in the next call to IPCRECV, unless the *destroy data flag* (29) was set.
- **flags [29]** — destroy data (input). If set, this flag causes delivered data that exceeds the amount allowed by the specified *dlen* or byte count (for vectored data) to be discarded. Use this flag to remove data that may have arrived at your node (and queued in the NetIPC buffer) that you do not want the process to receive.

  Note that in TCP stream mode, there is no mechanism to verify that data left over has been discarded.
- **flags [30]** — preview (input). This flag allows the calling process to preview the data - that is, to read the data without removing them from the queue of data to the receiving socket.
- **flags [31]** — vectored (input). This flag indicates that the received data are to be distributed to the addresses (vectors) given in the data parameter.

opt (input)

Record or byte array, by reference. A list of options, with associated information. The following options are defined:

- **call user data receive** (code 5, len=n; n bytes) (output). This option provides a buffer for the return of the call user data (CUD) field if you are using the fast select facility. If call user data is present, but this option is not supplied, then the discarded flag (*flags* bit 25) will be set. If the buffer supplied is not long enough to contain all the data, the data is truncated and the discarded flag is set. To ensure receiving all the CUD, this buffer should be at least 128 bytes long.
- **data offset** (code=8, length=2; 2-byte integer) (input/output). This option is valid for non-vectored compatibility mode data only. This option specifies an offset in bytes from the data parameter's address. The received data are to be written into memory beginning at this location.
- **protocol flags** (code=144, length=4; 4-byte buffer) (output). This option contains 32 bits of protocol-specific flags. The following flags are currently defined:
  - — **end-to-end acknowledgment** (bit 18, output). (X.25 only.) This flag indicates that the D bit is set in the X.25 packet associated with this call.
  - — **qualifier bit** (bit 19, output). (X.25 only.) This flag indicates that the Q bit is set in the X.25 packet associated with this call.
  - — **urgent data** (bit 27, output). (TCP only.) This flag indicates that urgent data has been received on an established connection.
- **Facility field** (code=145, length=n, n bytes) (output). (X.25 only). This option provides a buffer for the return of the facility field. If the buffer is not long enough to contain all of the data, then the data is truncated and the discarded flag (bit 25) is set. This buffer should be at least 109 bytes long to ensure receipt of the facility field (for more information, see Chapter 1 , "NetIPC Fundamentals.")

NOTE If using nowait I/O and *opt* array options that generate output, the array must remain intact until after IOWAIT completes. Otherwise, the array area will be overwritten or (if the area has been deleted from the stack) an error will occur.

result (output)

32-bit integer, by reference. The error code returned; zero if no error.

NOTE When nowait I/O is used, the *result parameter is not updated upon completion of an intrinsic. Therefore, the value of* result will indicate only whether the call was successfully initiated. To determine if the call completed successfully, you can use the IPCCHECK intrinsic after IOWAIT completes.

Description

The IPCRECV intrinsic serves two purposes: (1) to receive a response to a connection request, thereby completing a connection, and (2) to receive user data on an established connection.

NOTE In the first case the VC socket descriptor is the only required parameter; in the second, the VC socket descriptor, data, and data length parameters are required. The brackets in the syntax diagram represent the first case.

In receiving a response to a connection request, the IPCRECV intrinsic returns nothing in the data buffer. A result value of zero indicates a successful connection establishment. For X.25 only, the facility field and call user data field are returned in the option field. The *result* parameter will indicate an error if the destination rejected the request. In that case you must still call IPCSHUTDOWN with the returned VC socket descriptor value to shut down the connection.

Successful completion of an IPCRECV request on an established connection (*result* code zero) means that some amount of data was received: the amount requested or the amount transmitted, whichever is smaller. It does not mean that you received all the data you asked for.

Completion of IPCRECV (in wait mode) with a non-zero *result* can mean that a fatal error occurred. If nowait I/O is being used, IPCCHECK must be called to detect a fatal error reported for the specified VC descriptor after IOWAIT has been called and the receive completes.

Unless the intrinsic is called in nowait mode, the process is blocked until some data arrive or a timeout occurs. In nowait mode, the addresses of the *data*, *flags* and output *opt* options parameters are retained by NetIPC until needed. The input value of *flags* is retained

and updated (with the “more data” flag off or on) when IOWAIT completes. The *data* parameter will then contain the data received. Only one nowait receive may be outstanding on a single connection.

The returned *dlen* parameter (or the IOWAIT *tcount* parameter in the case of a nowait request) shows how many bytes of data were actually received in the *data* parameter. This amount may be different from what you requested. If you did not receive all the data you want, you can obtain the additional data in a subsequent IPCRECV call. For more information, see the discussion of “Sending and Receiving Data Over a Connection” in Chapter 1 , “NetIPC Fundamentals,” and the programmatic examples in Chapter 4 , “NetIPC Examples.”

Condition codes returned by this intrinsic are:

- CCE — Succeeded.
- CCL — Failed.
- CCG — Not returned by this intrinsic.

This intrinsic can be called in split stack mode.

Protocol-Specific Considerations

The following Table 3-7 outlines parameters that are specific to the particular protocol you are accessing.

Table 3-7 IPCRECV Protocol Specific Parameters

Parameters		TCP	X.25
flags			
	16	No output flag	n/a
	25	n/a	Discarded
	26	More data	More data
opt			
	5	n/a	CUD field
	144	n/a	Bit 18: state of D bit in X.25 packets
			Bit 19: state of Q bit in X.25 packets
			Bit 27: urgent data
	145	n/a	Facility field

X.25 Considerations

In receiving a response to a connection request, the option field returns the facility field and the call user data (CUD) field. With fast select, up to 128 bytes of call user data may be returned in the call user data receive buffer (*opt* 5).

A single IPCRECV call returns data for one message only. If the "more data" flag is set, the complete message has not been received. The remaining part of the message can be received by subsequent calls to IPCRECV, unless the destroy flag (29) is set.

If the destroy flag is set, the remaining part of the message is destroyed. The end of message is indicated by the "more data" flag not being set.

If an interrupt packet is received in the middle of a data packet stream, IPCRECV returns no data. The *result* parameter indicates that an interrupt event has occurred. The interrupt user data field can be retrieved by calling IPCCONTROL, request 12 (reason for error or event). The next call to IPCRECV returns the whole data message.

If a reset packet is received in the middle of a data packet stream, all previously received packets are discarded. IPCRECV returns no data. The *result* parameter indicates a reset has occurred. Use the IPCCONTROL request 12 (reason for error or event) to retrieve the cause and diagnostic fields for the reset.

Common errors returned by IPCRECV in *result* are:

```
SOCKERR   0  Request complete successfully.
SOCKERR  59  Socket timeout.
SOCKERR  65  Connection aborted by local protocol module.
SOCKERR  67  Connection failure detected.
SOCKERR 107  Transport is going down.
SOCKERR 117  Attempt to establish connection failed.
SOCKERR 146  Reset event occurred on X.25 connection.
SOCKERR 156  Interrupt event occurred on X.25
             connection.
SOCKERR 158  Connection request rejected by remote.
```

A complete table of SOCKERRs is include in Appendix C , "Error Messages."

TCP

The urgent data bit indicates that urgent data has been received. Table 3-8 demonstrates the meaning of urgent data and more data. Use these bits in combination to determine the status of data received.

Table 3-8 TCP Urgent and More Data Bit Combinations

Urgent	More Data	Meaning
0	0	Should never happen. (The receipt of normal data in stream mode causes more data to be set.)
0	1	Normal receive, no urgent data.
1	0	Urgent data received, no more urgent data.
1	1	Urgent data received and more is pending.

Cross-System Considerations for TCP

The following are cross-system programming considerations for this intrinsic:

HP 3000 to HP 1000: Receive size (*dlen* parameter) — Range for the HP 3000 is 1 to 30,000 bytes. Range for the HP 1000 is 1 to 8,000 bytes. Although the ranges are different, cross-system communication is not affected. If you specify a send or receive size, be sure it is within the correct range for the respective system.

Data wait flag — The HP 1000 IPCRecv call supports a “DATA_WAIT” flag. This flag, when set, specifies that the call will not complete until the amount of data specified by the *dlen* parameter has been received. This flag is not available on the HP 3000, meaning that the call may complete before all the data is received. However, the HP 3000 IPCRECV supports other flags such as the “more data” and “destroy data” flags.

HP 3000 to HP 9000: Receive size (*dlen* parameter) — Range for the HP 3000 is 1 to 30,000 bytes. Range for the HP 9000 is 1 to 32,767 bytes. Although the ranges are different, cross-system communication is not affected. If you specify a send or receive size, be sure it is within the correct range for the respective system.

Data wait flag — The HP 9000 IPCRECV call supports a “DATA_WAIT” flag. This flag, when set, specifies that the call will not complete until the amount of data specified by the *dlen* parameter has been received. This flag is not available on the HP 3000, meaning that the call may complete before all the data is received. However, the HP 3000 IPCRECV supports other flags such as the “more data” and “destroy data” flags.

HP 3000 to PC NetIPC: Receive size (*dlen* parameter) — Range for the HP 3000 is 1 to 30,000 bytes. The PC send and receive size range is 1 to 65,535 bytes. Although the ranges are different, cross-system communication is not affected. If you specify a send or receive size, be sure it is within the correct range for the respective system.

On the PC, you can specify the maximum receive size of the data buffer through the *got array* in the IPCCONNECT call. This determines what the maximum value the *dlen* parameter can be for any IPCRECV call. PC NetIPC has no option array defined in IPCCONNECT. This does not affect cross-system communication. The maximum receive size of the data in the buffer on the HP 3000 will determine the receive size buffer on the PC.

IPCRECVCN

Receives a connection request on a call socket.

Syntax

```
IPCRECVCN  (calldesc,vcdesc[,flags][,opt][,result])
```

Parameters

calldesc (input)

32-bit integer, by value. Call socket descriptor. The socket descriptor for a call socket belonging to this process.

vcdesc (output)

32-bit integer, by reference. The returned VC socket descriptor, a number identifying a VC socket belonging to this process through which data can be sent or received. This descriptor can be used in other intrinsics.

flags (input)

32 bits, by reference. A bit representation of various options. The following flags are defined:

- **flags [0]** — protected (input). (TCP only.) Ensures that the connection will be "protected" (privileged users only).
- **flags [18]** — defer (input). Causes the reply to the connection request to be deferred. The intrinsic will complete when a connection request is received, but the virtual circuit will not be established. The IPCCONTROL intrinsic can be used later to accept or reject the connection.
- **flags [21]** — checksum (input). (TCP only.) Enables checksum on the Transmission Control Protocol (TCP) connection for error checking. Checksum may also be set by the corresponding IPCCONNECT call. If either side specifies "checksum enabled" then the connection will be checksummed. TCP checksum may be enabled globally, over all connections, when configuring the Network Transport. See the *NS 3000/XL NMMGR Screens Reference Manual* for details on Network Transport configuration.

Checksum enabled by either IPCCONNECT or TCP (remote or local) configuration overrides a 0 setting (checksum disabled) for this flag. Checksum error checking is handled at the link level and is not normally required at the user level.

In fact, checksumming is only used for connections between nodes and is not used for connections within the same node. Enabling checksum may reduce network performance. Recommended value: 0.

- **flags [25]** — discarded (output). For X.25 protocol access. Indicates that the call user data (CUD) or the facility field was present, but that the data had to be discarded or truncated. If the call user data option (code=5) is not specified the call user data is discarded. If the CUD or the facility field buffer is not long enough to contain the data, this flag is set and the data is truncated.

opt (input)

Record or byte array, by reference. A list of options, with associated information. The following options are defined:

- **maximum send size** (code=3, length=2; 2-byte integer) (input). (TCP only). This option, which must be in the range from 1 to 30,000, specifies the length of the longest message that the user expects to send on this connection. The information is passed to the protocol module. If this option is not specified, then the protocol module will be able to handle messages at least 1,024 bytes long. If the specified value is smaller than a previously specified maximum send size, then the new value is ignored.
- **maximum receive size** (code=4, length=2; 2-byte integer) (input). (TCP only.) This option, which must be in the range from 1 to 30,000, specifies the length of the longest message that the user expects to receive on this connection. The information is passed to the protocol module. If this option is not specified, then the protocol module will be able to handle messages at least 1,024 bytes long. If the specified value is smaller than a previously specified maximum receive size, then the new value is ignored.
- **call user data** (code=5, length=n, n bytes) (output), (X.25 only.) This option provides a buffer for the

return of the call user data (CUD) field from an X.25 packet. If call user data is present, but this option is not supplied, the discarded *flag*[25] is set. If the buffer is not long enough to contain the data, the data is truncated and the discarded flag is set.

- **calling node address** (code=141, length=8; 8-byte array) (output). An output parameter that is used to contain the address of the requestor.

  For TCP, the first two bytes of the array contain the remote socket's port address and the next four types contain the remote node's internet protocol address. The remaining bytes are unused.

  For X.25 protocol access, the X.25 address of the calling node is returned in this field. The format of the record is equivalent to 16 nibbles (or BCD digits) in which the first nibble is the address length (ranging from 1 to 15), and the following 15 nibbles contains the calling address. The calling node address is not available if the call originated from a PAD.

  You can use READOPT to obtain the output of this parameter.

- **protocol flags** (code=144, length4; 4-byte buffer) (output). This option contains 32 bits of protocol-specific flags. The following flags are currently defined:

  — **Fast select** (bit 7, output). (X.25 only.) If this flag is set, the fast select facility has been used in the connection request.

  — **Fast select restricted** (bit 8, output). (X.25 only.) This flag only has meaning if the previous flag was set. If this flag is set, the fast select facility has been used in the connection request with restriction on response. This means that if the deferred flag was set, then the connection can be rejected (only) by using IPCCONTROL (and up to 128 bytes of CUD can be returned). If this flag is not set, but the fast select (bit 7) was set, the connection can be accepted or rejected by using IPCCONTROL, and up to 128 bytes of CUD can be returned.

  — **request from PAD** (bit 14, output). (X.25 only.) This flag indicates that the connection request is coming from a PAD as opposed to a connection coming from a host.

— **calling node address available** (bit 16, output). (X.25 only.) This flag indicates that the calling node X.25 address was present.

- **Facility field** (code=145, length=n, n bytes) (output). (X.25 only.) This option provides a buffer for the return of the facility field. If the facility field is present, but this buffer is not supplied, then the discarded flag (bit 25) is set.

  If the buffer is not long enough to contain all of the data, then the data is truncated and the discarded flag (bit 25) is set. This buffer should be at least 109 bytes long to ensure receipt of the facility field. The received buffer contains the facilities exactly as they were received from the network.

result (output)

32-bit integer, by reference. The error code returned; zero if no error.

Description

The IPCRECVCN intrinsic allows a process to receive a connection request and establish a connection (virtual circuit). The connection is identified by the returned VC socket descriptor. The calling process can then employ the IPCSEND and IPCRECV intrinsics to send and receive data on the connection. A maximum of one unreceived connection request may be queued to a call socket.

If the calling process sets the defer reply to connection request flag (*flags* [18]), this intrinsic will complete when a connection request is received, but the virtual circuit will not be established. The calling process must use IPCCONTROL to either accept or reject the request. This feature is useful if an application must defer replying to the connection request and then, depending upon the identity of the requestor, decide to reject or accept the request.

The calling process may also specify whether TCP checksumming is to be enabled. Checksumming is usually disabled unless it is included by the remote protocol module or if the TCP checksumming flag (*flags* [21]) is set. When checksumming is enabled, performance is usually degraded because of increased overhead.

If this intrinsic is called in nowait mode, the data structures for the connection are created when the call to IOWAIT completes. They are not created with the initial call to IPCRECVCN. Therefore the address of the VC socket descriptor parameter is retained by NetIPC, and the descriptor's value is returned to that location when IOWAIT completes. The VC socket descriptor parameter must be global to both the

IPCRECVCN and the IOWAIT intrinsic calls. NetIPC also retains the *flags* parameter.

The only required parameters are the *calldesc* and *vcdesc* parameters (option variable). Condition codes returned by this intrinsic are:

- CCE — Succeeded.
- CCL — Failed.
- CCG — Not returned by this intrinsic.

Condition codes returned by the call to IOWAIT are:

- CCE — Succeeded.
- CCL — Failure in NetIPC (for example, resource problems, VC socket descriptor bounds) or protocol module. In the event of a NetIPC failure the connection request will still be pending, allowing the user to correct the problem and issue another call to IPCRECVCN.
- CCG — Connection established but a noncritical error (for example, flags parameter out of bounds) occurred.

The IPCRECVCN intrinsic cannot be called in split stack mode.

Protocol-Specific Considerations

The following Table 3-9 outlines parameters that are specific to the particular protocol you are accessing.

Table 3-9 IPCRECVCN Protocol Specific Parameters

Parameters		TCP	X.25
flags			
	0	Protected connection	n/a
	21	Enable checksum	n/a
	25	n/a	Discarded
opt			
	3	Maximum send size	n/a
	4	Maximum receive size	n/a
	5	n/a	Received CUD
	141	Calling node's IP address	Calling node's X.25 address
	144	n/a	Bit 7:fast select Bit 8: fast select restricted Bit 14: PAD Bit 16: calling node address available flag
	145	n/a	Facility field

X.25 Considerations

IPCRECVCN is used with switched virtual circuits (SVCs) only.

The call user data field returned in the *opt* parameter (code=5) is used by X.25 as follows. The first four bytes of the call user data field is used to determine the destination call (source) socket. The incoming call is sent to the call socket whose relative protocol address matches the first four bytes of the call user data. See the discussion for IPCCREATE for more information on protocol relative addresses.

Call acceptance can be affected by the X.25 configuration of the security field in the SVC path table which can limit access to a node by specifying which remote X.25 addresses are allowed to communicate with the node. See the *X.25 XL System Access Configuration Guide* for more information about security features.

Common errors returned by IPCRECVCN in *result* are:

```
SOCKERR   0  Request completed successfully.
SOCKERR  59  Socket timeout.
SOCKERR 107  Transport is going down.
```

A complete table of SOCKERRs is included in Chapter C , "Error Messages."

TCP

The calling process may also specify whether checksumming is to be employed by the protocol modules (i.e., TCP) that support it. For TCP, checksumming is usually disabled unless it is included by the remote protocol module or if the TCP checksumming flag (*flags* [21]) is set. When checksumming is enabled, performance is usually degraded because of increased overhead.

Cross-System Considerations For TCP

The following are cross-system programming considerations for this intrinsic:

HP 3000 to HP 1000: Checksumming — TCP checksumming will be enabled for both sides of the connection if it is enabled by either side for HP 3000 to HP 1000 connections. On both the HP 3000 and HP 1000 checksumming can be enabled by setting bit 21 in the *flags* parameter.

Send and receive size — The HP 3000 send and receive size range is 1 to 30,000 bytes. The HP 1000 send and receive size range is 1 to 8,000 bytes. Although the ranges are different, you must specify a send size within the correct range for the respective receiving system; otherwise, an error will occur. For example, if the HP 3000 node sends 16,000 bytes, the HP 1000 node can call IPCRECV twice receiving 8,000 bytes the first time and the second 8,000 bytes the second time.

Note that the default send and receive sizes are different on different HP systems. On the HP 3000, the default send and receive size is less than or equal to 1,024 bytes. On the HP 1000 the default send and receive size is 100 bytes.

HP 3000 to HP 9000: Checksumming — When the ipcrecvcn() call is executed on the HP 9000 node, checksumming is always enabled.

Send and receive sizes — The HP 3000 send and receive size range is 1 to 30,000 bytes. The HP 9000 send and receive size range is 1 to 32,767 bytes. Although the ranges are different, cross-system communication is not affected. If you specify a send or receive size, be sure it is within the correct range for the respective system.

Note that the default send and receive sizes are different on different HP systems. On the HP 3000, the default send and receive size is less than or equal to 1,024 bytes. On the HP 9000, the default send and

receive size is 100 bytes.

HP 3000 to PC NetIPC Checksumming — With PC NetIPC, the TCP checksum option cannot be turned on. But if the HP 3000 requires it, the TCP checksum is in effect on both sides of the connection. On the HP 3000, enabling/disabling checksumming with NetIPC intrinsics allows you to override the checksumming decision made during network transport configuration for this particular process.

Send and receive size — The HP 3000 send and receive size range is 1 to 30,000 bytes. The PC send and receive size range is 1 to 65,535 bytes. Although the ranges are different, cross-system communication is not affected. If you specify a send or receive size, be sure it is within the correct range for the respective system. For example, if the PC node sends 60,000 bytes, the HP 3000 node can call IPCRECV twice, receiving 30,000 bytes the first time and the second 30,000 bytes the second time.

Note that the default send and receive sizes are different on different HP systems. On the HP 3000, the default send and receive size is less than or equal to 1,024 bytes.

IPCSEND

Sends data on a connection.

Syntax

```
IPCSEND  (vcdesc,data,dlen[,flags][,opt],result)
```

Parameters

vcdesc (input)

32-bit integer, by value. The VC socket descriptor, a number identifying the VC socket belonging to this process through which the data will be sent.

data (input)

Record or byte array, by reference. Contains the data to be sent or a list of data descriptors (maximum of two) indicating the locations from which the data will be gathered. If data descriptors are used, flags [31] must be set to indicate vectored sends. For programming in "C" language, see Appendix E , "C Program Language Considerations."

dlen (input)

32-bit integer, by value. The byte length of the data parameter: that is, the amount of actual data (maximum of 30,000) or the combined length of the data descriptors. The data descriptor length is 8 for compatibility mode and 12 for native mode. The combined length is 16 for compatibility mode and 24 for native mode.

flags (input)

32 bits, by reference. A bit representation of various options. The only flag defined is:

- **flags [31]** — vectored input. Indicates that the data to be sent are to be gathered from the addresses specified in the data parameter. (The parameter will not contain actual data.)

opt (input)

Record or byte array, by reference. A list of options, with associated information. Refer to "Common Parameters" for more information on the structure of this parameter. The following options are defined:

- **data offset** (code=8, length=2; 2-byte integer) (input). Compatibility mode (CM) only. An offset in bytes from the data parameter's address indicating the actual beginning of the data. HP recommends that you do not use data offset if data descriptors are used to point to another location from which data should be obtained.
- **protocol flags** (code=144, length=4; 4-byte buffer) (input). This option contains 32 bits of protocol-specific flags. The following flag is currently defined:
  - — **end-to-end acknowledgment** (bit 18, input). (X.25 only.) D bit will be set in the last X.25 data packet corresponding to this message. When this flag is set, IPCSEND waits to complete until acknowledgment from the remote that the complete message has been received. When the connection is between two HP 3000's running NS X.25, the acknowledgment is made when the remote IPC user has received the data.
  - — **qualifier bit** (bit 19, input). (X.25 only.) This flag indicates to X.25 to set the Q bit in the packets that contain this message.
  - — **urgent data** (bit 27, input). (TCP only). If set, this bit will cause the data sent to be marked urgent.

result (output)

32-bit integer, by reference. The error code returned; zero if no error.

NOTE When nowait I/O is used, the *result* parameter is not updated upon completion of IOWAIT. Therefore, the value of *result* will indicate only whether the call was successfully initiated. To determine whether the call completed successfully, you can use the IPCCHECK intrinsic after IOWAIT completes.

Description

The IPCSEND intrinsic is used to send data on a connection. The only required parameters are *vcdesc*, *data*, and *dlen* (option variable).

A set of addresses in the *data* parameter allows vectored data to be gathered from multiple locations.

The value specified for the data offset option (compatibility mode only) is relative to the data array. If data descriptors are used, specifying this option will cause a portion of the descriptor to be passed over (the offset

is NOT applied to the pointer in the descriptor). This may lead to unexpected results.

If this intrinsic is called in nowait mode, the address of the data is passed to the TCP protocol module. The contents of the data buffer will have been read when IOWAIT completes. As many as 7 nowait sends may be outstanding on a connection.

Condition codes returned by IPCSEND and IOWAIT are:

- CCE — Succeeded.
- CCL — Failed.
- CCG — Not returned.

This intrinsic can be called in split stack mode.

Protocol-Specific Considerations

The following Table 3-10 outlines parameters that are specific to the particular protocol you are accessing.

Table 3-10 IPCSEND Protocol Specific Parameters

Parameters	TCP	X.25
opt 144	Bit 27: urgent data	Bit 18: state of D bit in X.25 packets Bit 19: state of Q bit in X.25 packets

X.25 Considerations

Setting the Q bit flag causes X.25 to set the Q bit (qualifier bit) in X.25 data packets.

Setting the D bit flag causes X.25 to specify end-to-end acknowledgment of data packets. IPCSEND does not complete until it receives acknowledgment that the message has been received.

Common errors returned by IPCSHUTDOWN in *result* are:

```
SOCKERR   0  Request completed successfully.
SOCKERR  50  Invalid data length.
SOCKERR  65  Connection aborted by local protocol
             module.
SOCKERR  67  Connection failure detected.
SOCKERR 107  Transport is going down.
SOCKERR 159  Invalid X.25 D-bit setting.
SOCKERR 160  Incompatible with protocol state.
```

A complete table of SOCKERRs is included in Appendix C , "Error Messages."

TCP

The urgent data bit of the protocol flags option (*opt* parameter) is used to inform TCP that the data to be sent should be marked urgent. This will not cause the data to be delivered out of band, and the receiver of this data will not know of urgent data that is pending until a receive is posted.

Cross-System Considerations For TCP

The following are cross-system programming considerations for this intrinsic:

HP 3000 to HP 1000: Send size — The HP 3000 send size range is 1 to 30,000 bytes. The HP 1000 send size is 1 to 32,767 bytes. Although the ranges are different, cross-system communication is not affected. If you specify a send or receive size, be sure it is within the correct range for the respective system.

Note that the *urgent data* bit is not supported on the HP 1000; however, if this bit is set by the HP 3000 program, it will be ignored by the receiving process on the HP 1000.

HP 3000 to HP 9000: Send size — The HP 3000 send size range is 1 to 30,000 bytes. The HP 9000 send size is 1 to 32,767 bytes. Although the ranges are different, cross-system communication is not affected. If you specify a send or receive size, be sure it is within the correct range for the respective system.

Note that the *urgent data* bit is not supported on the HP 9000; however, if this bit is set by the HP 3000 program, it will be ignored by the receiving process on the HP 9000. For differences in send and receive sizes see the discussion for IPCRECVCN.

HP 3000 to PC NetIPC: Send size — The PC send and receive size range is 1 to 65,535 bytes. Although the ranges are different, cross-system communication is not affected. If you specify a send or receive size, be sure it is within the correct range for the respective system.

On the PC, you can specify the maximum receive size of the data buffer through the *got array* in the IPCCONNECT call. This determines what the maximum value for *dlen* can be for any IPCRECV call. PC NetIPC has no option array defined for IPCCONNECT. This does not affect cross-system communication. The maximum receive size of the data in the buffer on the HP 3000 will determine the receive size buffer on the PC.

IPCSHUTDOWN

Releases a descriptor and any resources associated with it.

Syntax

```
IPCSHUTDOWN  (descriptor[,flags][,opt][,result]
```

Parameters

descriptor (input)

32-bit integer, by value. The socket to be released. May be a call socket, destination, or VC socket descriptor.

flags

32 bits, by reference. A bit representation of various options. The following flag is defined:

flag [17] — (TCP only.) graceful release of connection.

opt

Record or byte array, by reference. A list of options, with associated information. The following option is defined:

- **clear user data** (code=2, length=n, n bytes) (input). (X.25 only.) If the fast select facility was used in the connection request, and the connection was accepted, you can include a clear user data field which can contain up to 128 bytes of data. For more information, see the discussion of the fast select facility in Chapter 1 , “NetIPC Fundamentals,” of this manual.
- **reason code** (code=143, length=2) (input). (X.25 only.) This option allows you to include cause and diagnostic values in the X.25 clear packets when a connection is closed down. The first byte contains the cause and the second byte contains the diagnostic code. A list of cause and diagnostic codes used with NS X.25 protocol access is contained in Appendix B , “Cause and Diagnostic Codes.” If DTE originated, the cause code will always be zero.

result (output)

32-bit integer, by reference. The error code returned; zero if no error.

Description

The IPCSHUTDOWN intrinsic permits the creating process to close a call socket or a VC socket descriptor or to release a connection. The descriptor is the only required parameter (option variable).

IPCSHUTDOWN can be called to release a call socket descriptor, a destination descriptor, or a VC socket descriptor. Since systems resources are used up as long as call sockets and destination sockets exist, you should release them whenever they are no longer needed.

The call socket is needed as long as a process expects to receive a connection request on that socket. A process which receives a connection request can release the call sockets any time after the connection request is received via IPCRECVCN, as long as no other connection requests are expected for that call socket. For more information on IPCSHUTDOWN, refer to "Shutting Down Sockets and Connections" at the beginning of this chapter. Condition codes returned by this intrinsic are:

- CCE — Succeeded.
- CCL — Failed.
- CCG — Not returned by this intrinsic.

This intrinsic cannot be called in split stack mode.

Protocol-Specific Considerations

The following Table 3-11 outlines parameters that are specific to the particular protocol you are accessing.

Table 3-11 IPCSEND Protocol Specific Parameters

Parameters	TCP	X.25
flags		
17	Graceful release of connection	n/a
opt		
2	n/a	Clear user data
143	n/a	Reason code

X.25 Considerations

Shutting down an X.25 connection causes a clear packet to be sent by X.25 over an SVC, unless the virtual circuit is already cleared. You can specify the cause and diagnostic fields in the *opt* parameter (code=143) that will be included in the clear packet over an SVC. Over a public

data network (PDN), the cause may not be transmitted to the remote node.

When used with direct access to level 3, the intrinsic IPCSHUTDOWN can only be called in waited mode. The intrinsic will not return until the request is completed.

X.25 direct access to level 3 does not support the *graceful release* bit. As a suggestion, to ensure that no data packets are lost before the clear packet is sent, use the D bit option in the last IPCSEND. This would assure end-to-end acknowledgment of this message before issuing the IPCSHUTDOWN to clear the virtual circuit. Another method is to send an unimportant message as the last message. (See example 2 in Chapter 4 , "NetIPC Examples," for an example of this method.)

Common errors returned by IPCSHUTDOWN in *result* are.

```
SOCKERR   0  Request completed successfully.
SOCKERR  54  Invalid call socket descriptor.
SOCKERR  66  Invalid connection descriptor.
SOCKERR  67  Connection failure detected.
SOCKERR 142  Invalid call user data opt record entry.
SOCKERR 170  Error with use of the fast select facility.
```

A complete table of SOCKERRs is included in Appendix C , "Error Messages."

TCP

If graceful release is specified and supported by the remote process, the requestor of a graceful release will go to a simplex-in state (that is, able only to receive, unable to send) and the remote process will go to a simplex-out state. The VC remains in this state until the remote process shuts down its socket, at which time all resources are released. See "Shutting Down a Connection" in Chapter 1 , "NetIPC Fundamentals," for the steps to take in implementing a graceful release shutdown.

If graceful release is selected, a SOCKERR 102 result will be returned if any of the following conditions exists:

- A connection request has been received, but the connection has not been accepted.
- The connection has already been gracefully released, and the process is therefore in a simplex-in state.
- A connection request has been issued, but the connection has not yet been established.
- The connection has been aborted.
- The protocol module (part of the NS transport) does not support graceful release.

- Data is being sent from the connection. This could occur, for example, if IPCSEND was called in nowait mode and has not yet completed.

Cross-System Considerations For TCP

The following are cross-system programming considerations for this intrinsic:

HP 3000 to HP 1000: Socket shut down — The HP 3000 provides a graceful release flag (flag 17) that is not available on the HP 1000. Do not set the graceful release flag on the HP 3000. Otherwise, the HP 1000 will not perform a normal shutdown. If the HP 3000 process sets the graceful release flag, the HP 1000 IPCRecv call will return a NetIPC error 68 (No more data). The HP 1000 process should handle error 68 as if it were an error 64 (Connection aborted by peer). After receiving an error 68, subsequent IPCRecv calls on the HP 1000 will return an error 109 (Remote connection has already graceful released the socket).

HP 3000 to HP 9000: Socket shut down — The HP 3000 provides a graceful release flag that is not available on the HP 9000. If the graceful release flag (flag 17) is set on the HP 3000, the HP 9000 will respond as though it were a normal shutdown. The HP 3000 does not support shared sockets; the HP 9000 does. Shared sockets are destroyed only when the descriptor being released is the sole descriptor for that socket. Therefore, the HP 9000 process may take longer to close the connection than expected.

HP 3000 to PC NetIPC: Socket shut down — The HP 3000 provides a graceful release flag that is not available on the PC. If the graceful release flag (flag 17) is set on the HP 3000, the PC will respond as though it were a normal shutdown.

OPTOVERHEAD

Returns the number of bytes needed for the *opt* parameter in a subsequent intrinsic call, not including the data portion of the parameter.

Syntax

```
optlength := OPTOVERHEAD(eventualentries[,result])
```

Parameters

optlength (returned function value)

16-bit integer. The number of bytes required for the *opt* parameter, not including the data portion of the parameter.

eventualentries (input)

16-bit integer, by value. The number of option entries that will be placed in the *opt* parameter.

result (output)

16-bit integer, by reference. The error code returned; zero if no error.

Description

This function returns the number of bytes needed for the *opt* parameter, excluding the data area. The first parameter is required.

Condition codes returned by this intrinsic are:

- CCE — Succeeded.
- CCL — Failed because of a user error.
- CCL — Failed because of a user error.

This intrinsic can be called in split stack mode.

READOPT

Obtains the option code and argument data associated with an *opt* parameter argument.

Syntax

```
READOPT (opt,entrynum,optioncode,datalength,data,result)
```

Parameters

opt (input)

Record or byte array, by reference. The *opt* parameter to be read. Refer to "NetIPC Intrinsics/Common Parameters" for information on the structure and use of this parameter.

entrynum (input)

16-bit integer, by value. The number of the option entry to be obtained. The first entry is number zero.

optioncode (output)

16-bit integer, by reference. The option code associated with the entry. These codes are described in each NetIPC call *opt* parameter description.

datalength (input/output)

16-bit integer, by reference. The length of the data buffer into which the entry should be read. If the data buffer is not large enough to accommodate the entry data, an error will be returned. On output, this parameter contains the length of the data actually read. (The length of the data associated with a particular option code is provided in each NetIPC call *opt* parameter description.)

data (output)

Byte array, by reference. An array which will contain the data read from the option entry. If the array is not large enough to hold the data read, only as much as requested will be returned and SOCKERR 137, more data available will be returned in the *result* parameter.

result (output)

16-bit integer, by reference. The error code returned; zero if no error.

Description

If the data field is not large enough, then as much data as the user asked for will be returned and SOCKERR 173 will be returned indicating more data is available. A second call to READOPT could get all the data.

Asynchronous I/O

In order to perform nowait (asynchronous) socket I/O on an HP 3000, a process must use the MPE XL IOWAIT and IODONTWAIT intrinsics. IOWAIT and IODONTWAIT behave in the same way except that, in the first case, the calling process must wait until the I/O operation is complete; in the second case, control is immediately returned to the calling process. One of these intrinsics must be called at some point after a nowait I/O request. The calling process is not blocked after the initial nowait I/O request.

IPCSEND, IPCRECV, and IPCRECVCN are normally blocking calls. The calling process must wait until the send/receive request is completed. A process can use IPCCONTROL to enable nowait I/O for a specified call socket or VC socket descriptor. (Nowait mode remains in effect until another IPCCONTROL call restores waited mode.) If a process issues a nowait send or receive request, the request will be initiated but its completion cannot be verified until IOWAIT or IODONTWAIT is called. (For a nowait IPCRECVCN call, the data structures for the connection are not created until IOWAIT is called.) IPCCONNECT is always an unblocked call: control returns immediately to the calling process, which must then call IPCRECV to complete the connection.

Within the IOWAIT/IODONTWAIT intrinsic, the *filenum* parameter should be given the appropriate call socket/VC socket descriptor value. A value of zero indicates all sockets or files for which asynchronous I/O requests have been issued. The function value returned by the intrinsic is the descriptor (or file number) for which the I/O has completed (zero if no completion).

The *cstation* (calling station) parameter returns a zero value for any nowait receive request. For a nowait send request, bit one of the parameter (the second highest bit) is set to on (all other bits off). Therefore you can check bit one of the *cstation* parameter to determine whether an input or an output operation completed.

The *tcount* parameter returns the amount of data received after a nowait IPCRECV call. The target parameter is not currently used by NetIPC.

The syntax for IOWAIT and IODONTWAIT is given here for convenience. For further information on these intrinsics, please see the *MPE XL Intrinsics Reference Manual*.

Steps for Programming with Asynchronous I/O

The following summarizes the steps to follow to have your program perform asynchronous I/O:

- Create the call or VC socket with IPCCREATE, IPCCONNECT, or IPCRECVCN.
- Enable nowait I/O with IPCCONTROL.
- Make a IPCRECVCN, IPCRECV, or IPCSEND NetIPC call on the socket. The call will be asynchronous.
- Check the *result* code returned by the call to see if an error occurred when the call was initiated.
- Call IOWAIT to cause the calling process to wait until the NetIPC call completes or IODONTWAIT to see if the request has completed.
- Once the asynchronous NetIPC call completes do the following:
  - — Check the condition code to see if an error occurred. If the condition code=CCE, no error occurred. If the condition code <> CCE, an error occurred.
  - — If an error occurred call IPCCHECK to determine the error code (returned in the *ipcerr* parameter).
  - — If IOWAIT or IDONTWAIT was called with *filenum*=0 or no *filenum* specified, check the *fnum* value returned to determine the socket for which I/O completed. (You can compare the *fnum* value with the *calldesc* value returned by IPCCREATE and the *vcdesc* value returned by IPCCONNECT and IPCRECVCN.)
  - — If both a send and receive request were pending, check the returned *cstation* value to determine if a send completed (bit 1 is on) or a receive completed (bit 1 is off).

NOTE A program does not need Privileged Mode capability in order to make nowait NetIPC I/O requests.

IO[DONT]WAIT

Initiates completion operations for a nowait I/O request.

Syntax

```
fnum := IO[DONT]WAIT ([filenum][,target][,tcount][,cstation])
```

Parameters

fnum (returned function value)

16-bit integer. The socket/VC socket descriptor for which an I/O request has completed. Zero indicates no completion. If there are any nowait file access requests outstanding, and if the *filenum* parameter is zero, the returned function value may be an actual file number.

filenum (input)

16-bit integer, by value. The call/VC socket descriptor indicating the socket or connection (that is, call or VC socket) for which the nowait I/O request was issued. If omitted, or if the value is zero, any nowait NetIPC or file request issued by the calling process may be completed by this intrinsic call.

target

Array of 16-bit values, by reference. Not used by NetIPC.

tcount (output)

16-bit integer, by reference. Returns the amount of data received after a nowait IPCRECV call. The actual data will be in the IPCRECV data parameter.

station (output)

16-bit integer (unsigned), by reference. Bit one is returned on if the completed request was a send, off if it was a receive. All other bits will be off.

Description

Either IOWAIT or IODONTWAIT is needed to complete a NetIPC nowait send or receive request. IOWAIT waits until a request can be completed; IODONTWAIT checks to see if a request can be completed and then immediately returns control to the calling process.

If a nowait IPCSEND, IPCRECVCN, or IPCRECV request is issued, the data and flags parameters (if specified) must exist when IOWAIT or IODONTWAIT is called. In other words, these parameters must be global to both the IPC intrinsic that initiates the request and the IO[DONT]WAIT that attempts to complete the I/O.

All parameters are optional (option variable). In general, the condition codes returned by IOWAIT/IODONTWAIT for socket I/O have the following meanings:

- CCE — Succeeded.
- CCL — Failed.
- CCG — The operation succeeded but a noncritical error occurred (for example, the flags parameter was out of bounds).

IOWAIT and IODONTWAIT can be called in split stack mode.

4 NetIPC Examples

This chapter contains sample programs using NetIPC for both TCP access and X.25 (level 3) access:

- Example 1 (programs 1A and 1B) present an example of how to set up and use a connection (virtual circuit) for TCP access. The two programs, running on different nodes, open communication via call sockets. They then establish a connection (using VC sockets) and send and receive data over this connection. Finally, they terminate their connection.
- Example 2 (programs 2A and 2B) illustrate the differences in using NetIPC for vectored I/O in compatibility mode and in native mode for TCP access.
- Example 3 (programs 3A and 3B) provides an example using NetIPC to communicate using direct access to level 3, X.25 in compatibility mode.
- Example 4 (programs 4A and 4B) provides an example using NetIPC to communicate using direct access to level 3, X.25 in native mode. These programs use the features provided with MPE XL X.25 such as adding to the facility field, and using fast select.

In these programs, it is assumed that processes are already operating at the remote and local nodes. For an example program that illustrates the use of Remote Process Management (RPM) intrinsics to start processes, refer to the *Using NS 3000/XL Network Services Manua*l.

NOTE You may need to disable timeouts on either the call descriptors or virtual circuit descriptors when writing applications intended for use on networks where you expect significant timing delays due to excessive line noise or traffic.

Example 1

In the first two programs (1A and 1B), the lengths of the data messages are not known. The sending side (Program 1A) includes the length of each message as the first two bytes of each message it sends. The receiving side (Program 1B) executes two IPCRECV loops for each message: first to receive the length and then to receive the data.

The first program (Program 1A):

- looks up the call socket named RALPH located on node JANE and gets back a destination descriptor;
- creates its own call socket;
- sends a connection request to RALPH;
- shuts down its call socket and its destination socket;
- completes the connection;
- executes a loop in which it:
  - — reads a line of data;
  - — stores the length (number of bytes) of the data in the first part of the message;
  - — stores the data itself in the second part of the message;
  - — sends the message on the connection, including the message length as the first two bytes of the message;
- sends a “last message” which will be recognized by the receiving program as a termination request;
- receives a “termination confirmation message” and shuts down the connection by releasing its VC socket.

The second program (Program 1B):

- creates a call socket and names it RALPH;
- waits to receive a connection request;
- shuts down its call socket;
- executes a loop in which it:
  - — calls a procedure that receives a message by executing two IPCRECV loops (the first loop determines the incoming message length and the second loop receives data until all the pieces of the message have been received);
  - — prints the message which was received;
- receives a “last message” termination request;

- sends a “termination confirmation message” in response to the termination request;
- receives a *result* parameter value of 64 (“REMOTE ABORTED CONNECTION”) in response to a receive request;
- releases its VC socket.

Program 1A

```
$standard_level 'HP3000', uslinit$
program connection_example1 (input,output);

const
     maxdata = 2000;
     maxmsg = maxdata + 2;
     maxname = 20;
     maxloc = 20;

type
     smallint = -32768..32767;
    datatype = record;
       len : smallint;
       msg : packed array[1..maxdata] of char;
    end;

timeval type =
    record case boolean of
       true  : (int   : smallint);
       false : (chars : packed array [1..30] of char);
    end;

    nametype = packed array[1..maxname] of char;
    loctype = packed array[1..maxloc] of char;

var  calldesc       :     integer;   {2-word integer}
       vcdesc       :     integer;
       protocol  :     integer;
       socket_kind  :     integer;

       dest         :     integer;
       result       :     integer;
       data         :     datatype;
       name         :     nametype;
       location  :     loctype;
       y_len        :     integer;
       y_data       :     char;
       num_msgs  :     integer;
       strdata      :     string[maxdata];
       i            :     integer;
       timeval  :     timeval type;

procedure terminate;      intrinsic;
{NetIPC intrinsic declarations}
procedure   ipccreate;    intrinsic;
procedure   ipclookup;    intrinsic;
```

```
procedure   ipcconnect;   intrinsic;
procedure   ipcrecv;      intrinsic;
procedure   ipcsend;      intrinsic;
procedure   ipcshutdown   intrinsic;
procedure   ipcerrmsg;    intrinsic;
procedure   ipccontrol;   intrinsic;

{error handling procedure}

procedure leave(result: integer);
     var  msg: string[80];
          i, len, newresult: integer;
begin
     ipcerrmsg(result, msg, len, newresult);
     if newresult = 0 then
       begin
          setstrlen(msg, len);
          writeln(msg);           {print error message}
       end
     else
     writeln('IpcErrMsg result is ', newresult:1);
terminate;
end;

{main of NetIPC Program 1}

begin
{ look up the call socket RALPH located on node JANE }
name:= 'RALPH';
location:= 'JANE';
ipclookup( name, 5, location, 4, , dest, protocol, socket_kind, result);
     if result <> 0 then leave(result); {failed}

{ create a call socket; then initiate and complete connection to destination
socket}

ipccreate(socket_kind, protocol, , , calldesc, result);
     if result <> 0 then leave(result); {failed}
ipcconnect(calldesc, dest, , , vcdesc, result);  {initiate connection}
     if result <> 0 then leave(result); {failed}
timeval.int:=0;
ipccontrol(vcdesc, 3, timeval.chars, 2, , , result);
     if result <> 0 then leave(result);
ipcshutdown(calldesc);
ipcshutdown(dest);
ipcrecv(vcdesc, , , , , result); {complete connection}
if result <> 0 then leave(result);     {failed}

{ prompt for messages and send them }
writeln('Enter "//" to terminate the program.');
setstrlen(strdata, 0);
while strdata <> '//' do
     begin
     prompt('Message? ');
```

```
     readln(strdata);             {read message}
     data.len := strlen(strdata); {store message length}
     strmove(data.len, strdata, 1, data.msg, 1);        {store message}
     ipcsend(vcdesc, data, data.len+2, , ,result)  {send message with length
                                                    as first 2 bytes}
   if result <>  0 then leave(result); {failed}
end;

{connection shutdown procedure}

data.len := 4;
data.msg := 'END?'                          { termination request}
ipcsend(vcdesc, data, 6, , , result);
      writeln('END sent');
if result <> 0 then leave(result);
      y_len := 1;
ipcrecv(vcdesc, y_data, y_len, , , result);   {receive 'Y' confirmation}
if (y_data = 'Y') then writeln('Y received');
if (y_data = 'Y') and (result = 0) then
           ipcshutdown(vcdesc)
else
      begin
        writeln('Warning: shutdown not confirmed or result  0');
        leave(result);
     end;
end.
```

Program 1B

```
$standard_level 'HP3000', uslinit$
program connection_example2 (output);

const
    maxdata = 2000;
    maxname = 20;
type
     smallint = -32768..32767;
     datatype = packed array [1..maxdata] of char;
timeval type =
     record case boolean of
     true  : (int   : smallint);
           false : (chars : packed array [1..30] of char);
     end;
           nametype = packed array [1..maxname] of char;
var        calldesc    :     integer;          {2-word integer}
           vcdesc      :     integer;
           dlen        :     integer;
           result      :     integer;
           data        :     datatype;
           name        :     nametype;
           len         :     smallint;
           datastr     :     string[maxdata];
           timeval     :     timeval type;
```

```
procedure terminate;    intrinsic;

{NetIPC intrinsic declarations}

procedure    ipccreate;        intrinsic;
procedure    ipcname;          intrinsic;
procedure    iprecvcn;         intrinsic;
procedure    ipcrecv;          intrinsic;
procedure    ipcsend;          intrinsic;
procedure    ipcshutdown;      intrinsic;
procedure    ipcerrmsg;        intrinsic;
procedure    ipccontrol;       intrinsic;

{error handling procedure}

procedure leave(result : integer);
     var msg: string[80];
          i, len, newresult: integer;
begin
     ipcerrmsg(result, msg, len, newresult);
     if newresult = 0 then
          begin    setstrlen(msg, len);
          writeln(msg);                        {print error message}
     end
     else
     writeln('IpcErrMsg result is ', newresult:1);
     terminate;
end;

{ The following procedure receives one message which was sent via an ipcsend
call. It assumes that the length (number of bytes) of the message was sent as
the first two bytes of data and that the length value does not include those
two bytes. }

procedure receive ( connection : integer;
                    var       rbfr :      datatype;
                    var       rlen :      smallint;
                    var       errorcode : integer    ) ;
const     head_len = 2;

type      length_buffer_type = packed array[1..2] of char;
          header_len_type = record
                 case integer of
                 0: ( word: smallint );
                 1: ( byte: length_buffer_type);
                 end;
var  i, j                   :integer;
           dlen             :integer;
           header_len  :header_len_type;
           tempbfr         :datatype;

begin      { procedure receive }
i:=0;
errorcode := 0;
while (i <> head_len) and (errorcode = 0) do { get length of message }
```

```
     begin
     dlen := head_len - i;
     ipcrecv( connection, tempbfr, dlen, , , errorcode );
     if errorcode = 0 <F100P12M>
        then strmove(dlen, tempbfr, 1, header_len.byte, i+1);
          i := i + dlen;
     end;
if errorcode = 0 then
     begin
     rlen := header_len.word;
     i := 0;
     while (i <> rlen) and (errorcode = 0) do { get the message }
         begin
         dlen := header_len.word - i;
         ipcrecv ( connection, tempbfr, dlen, , ,errorcode );
         if errorcode = 0
           then strmove(dlen, tempbfr, 1, rbfr, i+1);
           i := i + dlen;
         end;
     end
else
     rlen := 0;
end;

{ procedure receive }
{main of NetIPC Program 2}

begin

{create a call socket and name it}
ipccreate(3, 4, , , calldesc, result);
if result <> 0 then
     leave(result);                  {failed}
name := 'RALPH';
ipcname(calldesc, name, 5, result );
if result <> 0 then
     leave(result);                  {failed}
{wait for a connection request}
timeval.int:=0;
ipccontrol(calldesc, 3, timeval.chars, 2, , , result);

ipcrecvcn(calldesc, vcdesc, , , result);
if result <> 0 then
     leave(result);                  {failed}
ipcshutdown(calldesc);

{wait for a message on the connection and print message received}

repeat
     begin
     receive (vcdesc, data, len, result);
     if result <> 0 then leave(result);
     setstrlen(datastr, len);
```

```
     strmove(len, data, 1, datastr, 1);
     if datastr <> 'END?' then writeln (datastr);       {print data received}
     end
until datastr = 'END?';

{connection shutdown procedure}

if datastr = 'END?' then writeln('END received');
data := 'Y';
ipcsend( vcdesc, data, 1, , , result ); {confirmation message}
writeln('Y sent');
if result <> 0 then leave(result);
receive(vcdesc, data, len, result );
if result = 64 then
     ipcshutdown(vcdesc)
else
     leave(result );
end.
```

Example 2

This pair of programs show the differences in compiler options for writing NetIPC programs to run in compatibility mode and native mode. The programs are designated vector1 and vector2. You can compile them in either compatibility mode or native mode as described in the comments preceding the programs.

Program 2A (Vector1)

```
{*********************************************************************}
{ This program pair, vector1 and vector2, gives an example of how to   }
{ send and receive vectored data, both in Compatibility Mode and       }
{ Native Mode.  To compile in Native Mode, set the native_mode flag to }
{ true; compile with "pasxl svector1,,$null" and link with             }
{ "link $oldpass,nvector1".  Run nvector1 before running nvector2.     }
{ To compile in Compatibility mode, set the native_mode flag to false; }
{ compile with "pascal svector1,,$null" and link with                  }
{ "prep $oldpass,pvector1".  Run pvector1 before pvector2.  You can    }
{ run pvector1 with nvector2, or nvector1 with pvector2.               }
{*********************************************************************}
$set 'native_mode = true '$
$stats off$
$code_offsets on$
$tables on$
$lines 120$
$if 'native_mode'$
  $standard_level 'hp_modcal'$
   $type_coercion 'conversion'$
$else$
   $uslinit$
$endif$

program vector1 (input,output);

TYPE
   byte     =      0..255;            { this is one byte long         }
   shortint = -32768..32767;          { this is two bytes long        }

CONST
$if 'native_mode'$
   DESC_TYPE    =     4;              { descriptor type for 64b ptr   }
   DESC_LEN     =    12;              { length of NM vector descriptor}
$else$
   DESC_TYPE    =     0;              { descriptor type for CM stack  }
   DESC_LEN     =     8;              { length of CM vector descriptor}
$endif$

   F_VECTORED   =    31;              { vectored data                 }
```

```
TYPE
   flags_type       = set of 0..31;
   msg_type         = packed array [1..80] of char;

   netipc_data_desc = packed record
      { This structure contains a maximum of two data descriptors.     }
$if 'native_mode'$
      d_desc_type1      : shortint;       { type of data desc - use 4  }
      d_desc_len1       : shortint;       { length in bytes of area 1  }
      d_desc_dataptr1   : globalanyptr;   { pointer to area 1          }

      d_desc_type2      : shortint;       { type of d_d - use 4        }
      d_desc_len2       : shortint;       { length in bytes of area 2  }
      d_desc_dataptr2   : globalanyptr;   { pointer to area 2          }
$else$
      d_desc_type1      : shortint;       { type of data desc - use 0  }
      d_desc_dst1       : shortint;       { dst is 0 for stack         }
      d_desc_dataptr1   : shortint;       { pointer to area 1          }
      d_desc_len1       : shortint;       { length in bytes of area 1  }

      d_desc_type2      : shortint;       { type of d_d - use 0        }
      d_desc_dst2       : shortint;       { dst is 0 for stack         }
      d_desc_dataptr2   : shortint;       { pointer to area 2          }
      d_desc_len2       : shortint;       { length in bytes of area 2  }
$endif$
      end;

CONST
   SOCK_ADDR = 32000;                { socket's address               }

VAR
   sd_local:     integer;             { local socket descriptor        }
   cd_local:     integer;             { local connection descriptor    }
   dlen:         integer;             { data length                    }
   flags:        flags_type;          { flags parameter                }
   result:       integer;             { back from IPC call             }
   result16:     shortint;            { back from opt calls            }
   i:            integer;             { loop counter for messages      }
   messag1:      msg_type;            { for printed messages           }
   messag2:      msg_type;            { for printed messages           }
   expect :      msg_type;            { expected message               }
   vd:           netipc_data_desc;    { vectored data desc             }
   error:        boolean;             { set if an error occurred       }
   adrs: packed array [0..1] of byte; { socket's address               }
   opt:  packed array [0..31] of byte;{ options array                  }

{ IPC intrinsics used }
procedure addopt;      intrinsic;
procedure initopt;     intrinsic;
procedure ipccheck;    intrinsic;
procedure ipcconnect;  intrinsic;
procedure ipccontrol;  intrinsic;
procedure ipccreate;   intrinsic;
procedure ipcdest;     intrinsic;
```

```
procedure ipcerrmsg;   intrinsic;
procedure ipcget;      intrinsic;
procedure ipcgive;     intrinsic;
procedure ipcrecv;     intrinsic;
procedure ipcrecvcn;   intrinsic;
procedure ipcsend;     intrinsic;
procedure ipcshutdown; intrinsic;

begin
$if 'native_mode'$
writeln
('example program vector1 to show vectored data operation in Native Mode');
$else$
writeln
('example program vector1 to show vectored data operation in Compatibility
Mode'
 );
$endif$

{ specify the address of the local socket }
adrs[0] := SOCK_ADDR div 256;   { first 8 bits of 32000                }
adrs[1] := SOCK_ADDR mod 256;   { last 8 bits of 32000                 }

{ initialize opt array for one entry }
initopt ( opt, 1, result16 );
if result16 <> 0 then
   writeln ('initopt failed');

{ add the option for specification of the socket's address             }
addopt ( opt, 0, 128, 2, adrs, result16 );
if result16 <> 0 then
   writeln ('addopt failed');

{ Create the local socket by using the special option 128 which allows }
{ specification of the socket's address using the opt array.           }
ipccreate ( 3, 4, , opt, sd_local, result );
if result <> 0 then
   writeln ('ipccreate of local socket failed ',  result );
{ Local side receives the connection }
ipcrecvcn ( sd_local, cd_local, , , result );
if result <> 0 then
   writeln ('ipcrecvcn failed');

{ set up vectors, ready for sending and receiving data                 }
$if 'native_mode'$
vd.d_desc_dataptr1 := globalanyptr ( addr ( messag1 ) );
vd.d_desc_dataptr2 := globalanyptr ( addr ( messag2 ) );
$else$
vd.d_desc_dst1     := 0;    { this is ignored                          }
vd.d_desc_dataptr1 := baddress ( messag1 );
vd.d_desc_dst2     := 0;    { this is ignored                          }
vd.d_desc_dataptr2 := baddress ( messag2 );
 $endif$
```

```
vd.d_desc_type1    := DESC_TYPE;
vd.d_desc_type2    := DESC_TYPE;
flags              := [ F_VECTORED ];
{ Receive the message in a double vector                                    }
messag1 := '                                                  ';      { 46 }
messag2 := '                                                  ';      { 46 }
vd.d_desc_len1 := 27;              { max we are willing to receive          }
vd.d_desc_len2 := 80;              { max we are willing to receive          }
dlen := DESC_LEN * 2;              { 2 vectors                              }
ipcrecv ( cd_local, vd, dlen, flags, , result );
if result <> 0 then
   writeln ('ipcrecv data failed');
if dlen <> 40 then
   writeln ('dlen was not = 40');

{ Check that the correct data was received in the first vector       }
expect  := '40 four oh forty XL 40 four ';
error := false;
for i := 1 to 27 do
   if messag1[i] <> expect[i] then
      error := true;
if error then
   begin
   writeln ('did not receive expected first vector data, got:');
   writeln ( messag1 );
   end;

{ Check that the correct data was received in the second vector      }
expect  := ' tens fortify';
error := false;
for i := 1 to (dlen - 27) do
   if messag2[i] <> expect[i] then
      error := true;
if error then
   begin
   writeln ('did not receive expected second vector data, got:');
   writeln ( messag2 );
   end;
{ Now send a single vectored message to the local side }

messag1         := '.';            { This means GREETINGS          }
vd.d_desc_len1  := 1;              { byte size of message          }
dlen            := DESC_LEN;
ipcsend ( cd_local, vd, dlen, flags, , result );
if result <> 0 then
   writeln ('ipcsend failed');

{ do a regular receive of the double vectored send                    }
messag1 := '                                                  ';      { 46 }
dlen    := 46;                       { max amount of data to receive }
ipcrecv ( cd_local, messag1, dlen, , , result );
if result <> 0 then
   writeln ('ipcrecv data failed');
if dlen <> 21 then
```

```
   writeln ('dlen was not = 21');

{ Check that the correct data was received }
expect  := 'Abaracadabara  magic!';
error := false;
for i := 1 to dlen do
   if messag1[i] <> expect[i] then
      error := true;
if error then
   begin
   writeln ('did not receive expected data, got:');
   writeln ( messag1 );
   end;

{ Clean up and shutdown }

{ shutdown the local connection descriptor }
ipcshutdown ( cd_local, , , result );
if result <> 0 then
   writeln ('ipcshutdown cd_local failed');
{ shutdown the local socket descriptor }
ipcshutdown ( sd_local, , , result );
if result <> 0 then
   writeln ('ipcshutdown sd_local failed');

end.
```

Program 2B (Vector2)

```
{*********************************************************************}
{ This program pair, vector1 and vector2, gives an example of how to   }
{ send and receive vectored data, both in Compatibility Mode and       }
{ Native Mode.  To compile in Native Mode, set the native_mode flag to }
{ true; compile with "pasxl svector2,,$null" and link with             }
{ "link $oldpass,nvector2".  Run nvector1 before running nvector2.     }
{ To compile in Compatibility mode, set the native_mode flag to false; }
{ compile with "pascal svector2,,$null" and link with                  }
{ "prep $oldpass,pvector2".  Run pvector1 before pvector2.  You can    }
{ run pvector1 with nvector2 or nvector1 with pvector2.                }
{*********************************************************************}
$set 'native_mode = true '$
$stats off$
$code_offsets on$
$tables on$
$lines 120$
$if 'native_mode'$
   $standard_level 'hp_modcal'$
   $type_coercion 'conversion'$
$else$
   $uslinit$
$endif$

program vector2 (input,output);
```

```
TYPE
   byte     =      0..255;              { this is one byte long         }
   shortint = -32768..32767;            { this is two bytes long        }

CONST
$if 'native_mode'$
   DESC_TYPE    =     4;                { descriptor type for 64b ptr   }
   DESC_LEN     =    12;                { length of NM vector descriptor}
$else$
   DESC_TYPE    =     0;                { descriptor type for CM stack  }
   DESC_LEN     =     8;                { length of CM vector descriptor}
$endif$

   F_VECTORED   =    31;                { vectored data                 }

TYPE
   flags_type       = set of 0..31;
   location_type    = packed array [1..50] of char;
   msg_type         = packed array [1..80] of char;

   netipc_data_desc = packed record
      { This structure contains a maximum of two data descriptors.     }
$if 'native_mode'$
      d_desc_type1      : shortint;     { type of data desc - use 4    }
      d_desc_len1       : shortint;     { length in bytes of area 1    }
      d_desc_dataptr1   : globalanyptr; { pointer to area 1            }

      d_desc_type2      : shortint;     { type of d_d - use 4          }
      d_desc_len2       : shortint;     { length in bytes of area 2    }
      d_desc_dataptr2   : globalanyptr; { pointer to area 2            }

$else$

      d_desc_type1      : shortint;     { type of data desc - use 0    }
      d_desc_dst1       : shortint;     { dst is 0 for stack           }
      d_desc_dataptr1   : shortint;     { pointer to area 1            }
      d_desc_len1       : shortint;     { length in bytes of area 1    }

      d_desc_type2      : shortint;     { type of d_d - use 0          }
      d_desc_dst2       : shortint;     { dst is 0 for stack           }
      d_desc_dataptr2   : shortint;     { pointer to area 2            }
      d_desc_len2       : shortint;     { length in bytes of area 2    }
$endif$
      end;

CONST
   SOCK_ADDR = 32000;                   { socket's address              }

VAR
   sd_remote:    integer;               { remote socket descriptor      }
   cd_remote:    integer;               { remote connection descriptor  }
   dd:           integer;               { destination descriptor        }
   dlen:         integer;               { data length                   }
```

```
   flags:        flags_type;           { flags parameter              }
   result:       integer;              { back from IPC call           }
   result16:     shortint;             { back from opt calls          }
   i:            integer;              { loop counter for messages    }
   messag1:      msg_type;             { for printed messages         }
   messag2:      msg_type;             { for printed messages         }
   expect :      msg_type;             { expected message             }
   vd:           netipc_data_desc;     { vectored data desc           }
   error:        boolean;              { set if an error occurred     }
   location:     location_type;        { other programs location node }

   adrs: packed array [0..1] of byte;  { socket's address             }

   opt:  packed array [0..31] of byte; { options array                }

{ IPC intrinsics used }
procedure addopt;      intrinsic;
procedure initopt;     intrinsic;
procedure ipccheck;    intrinsic;
procedure ipcconnect;  intrinsic;
procedure ipccontrol;  intrinsic;
procedure ipccreate;   intrinsic;
procedure ipcdest;     intrinsic;
procedure ipcerrmsg;   intrinsic;
procedure ipcget;      intrinsic;
procedure ipcgive;     intrinsic;
procedure ipcrecv;     intrinsic;
procedure ipcrecvcn;   intrinsic;
procedure ipcsend;     intrinsic;
procedure ipcshutdown; intrinsic;


begin
$if 'native_mode'$
writeln
('example program vector2 to show vectored data operation in Native Mode');
$else$
writeln
('example program vector2 to show vectored data operation in
  Compatibility Mode');
$endif$

 { create the remote socket normally }
ipccreate ( 3, 4, , , sd_remote, result );
if result <> 0 then
   writeln ('ipccreate of remote socket failed');

{ Get the destination descriptor to the local socket from the remote   }
{ socket.  Notice that the remote must know the address of the local   }
{ socket.  This arrangement must be made beforehand.                   }

{ specify the address of the local socket                              }
adrs[0] := SOCK_ADDR div 256;   { first 8 bits of 32000                }
adrs[1] := SOCK_ADDR mod 256;   { last 8 bits of 32000                 }
```

```
location := 'bigblue';
ipcdest ( 3, location, 7, 4, adrs, 2, , , dd, result );


if result <> 0 then
   writeln ('ipcdest failed ' , result );

{ Connect to the local socket using the destination descriptor.         }
ipcconnect ( sd_remote, dd, , , cd_remote, result );

 if result <> 0 then
   writeln ('ipcconnect failed ', result );

{ remote side does a receive to complete the connection }
ipcrecv ( cd_remote, , , , , result );
if result <> 0 then
   writeln ('ipcrecv to complete connection failed');

{ set up vectors ready for sending and receiving data                 }
$if 'native_mode'$
vd.d_desc_dataptr1 := globalanyptr ( addr ( messag1 ) );
vd.d_desc_dataptr2 := globalanyptr ( addr ( messag2 ) );
$else$
vd.d_desc_dst1     := 0;    { this is ignored              }
vd.d_desc_dataptr1 := baddress ( messag1 );
vd.d_desc_dst2     := 0;    { this is ignored              }
vd.d_desc_dataptr2 := baddress ( messag2 );
$endif$
vd.d_desc_type1    := DESC_TYPE;
vd.d_desc_type2    := DESC_TYPE;
flags              := [ F_VECTORED ];

{ send a non-vectored message to the local side                       }

messag1  := '40 four oh forty XL 40 four tens fortify';
ipcsend ( cd_remote, messag1, 40, , , result );
if result <> 0 then
   writeln ('ipcsend failed');

{ receive a message in a single vector                                  }
messag1 := '                                                ';      { 46 }
vd.d_desc_len1  := 46;              { max we are willing to receive    }
dlen            := DESC_LEN;

ipcrecv ( cd_remote, vd, dlen, flags, , result );
if result <> 0 then
   writeln ('ipcrecv data failed');
if dlen <> 1 then
   writeln ('dlen was not = 1');

{ Check that the correct data was received }
expect := '.';
error := false;
for i := 1 to dlen do
```

```
   if messag1[i] <> expect[i] then
      error := true;
if error then
   begin
   writeln ('did not receive expected single vector data, got:');
   writeln ( messag1 );
   end;

{ send a double vectored message to the local side                    }
messag1 := 'Abaracadabara  ';
messag2 := 'magic!';
vd.d_desc_len1     := 15;             { byte size of message          }
vd.d_desc_len2     :=  6;             { byte size of message          }
dlen := DESC_LEN * 2;                 { there are 2 descriptors       }
ipcsend ( cd_remote, vd, dlen, flags, , result );
if result <> 0 then
   writeln ('ipcsend failed');

{ do a dummy receive so that the other side can receive the last      }
{ message before disconnection                                        }
dlen := 1;
ipcrecv ( cd_remote, messag1, dlen );

{ sockets are released on process termination                         }
end.
```

Example 3

Example 3 includes a pair of programs designated requester (X25CHECK) and server (X25SERV) using direct access to X.25 at level 3. These programs must be compiled in compatibility mode. The X.25 features used in these programs are the set supported on MPE-V. Example 4 uses the additional X.25 features supported on MPE XL. The program functions are described in the comments included with the program listings.

Program 3A (X25CHECK)

```
{*********************************************************}
{         Declarations for X52CHECK and X25SERVR           }
{*********************************************************}
CONST
   c_prot_addr_x25chk = 31000;  {X25CHECK protocol address}
   c_prot_addr_server = 31001;  {X25SERV  protocol address}
           {These decimal addresses are in the range 30767..32767 where PM }
           { is not required }
   c_patern='abcdefghijklmnopqrstuvwxyz0123456789';
   c_buffer_len = 36;
   c_nb_loop =10;
   c_calling_add_code = 141;
   c_prot_add_code    = 128;
   c_net_name_code    = 140;
   c_clear_rcvd       = 67;     {SOCKERR for a CLEAR packet received}

TYPE
    shint = -32768..32767;
    nibble = 0..15;
    byte  = 0..255;
    rc_type = (done,
               error,
               no_vc_desc,
               no_dest_desc,
               no_call_desc);
    event_type = (i_addopt,
                  i_create,
                  i_dest,
                  i_connect,
                  i_recv_call_conf,
                  i_send,
                  i_recv,
                  i_shut_source,
                  i_shut_dest,
                  i_shut_connection);
    event_msg_type = array [event_type] of string[80];
    opt_type = packed record                          {            }
               length : shint;                        {            }
               num_entries : shint;                   {Declarations}
```

```
                data : packed array [1..256] of shint;{            }
               end;                                   {    for     }
    buffer_type = string [c_buffer_len] ;             {            }
                                                      {  NetIPC    }
    socket_type  = (call,destination,vc);             {            }
    name_type = string [50];                          {            }
    name_len = shint;

CONST
  c_event_msg = event_msg_type
               ['construction of option record',
                'creation of the local call descriptor',
                'creation of the destination descriptor',
                'CALL packet sending',
                'CALL CONF packet reception',
                'DATA packet sending',
                'DATA packet reception',
                'shutdown of call descriptor',
                'shutdown of destination descriptor',
                'CLEAR packet sending'];

VAR
  rc              : rc_type;
  result          : integer;
  r               : shint;
  p_call_desc     : integer;
  p_vc_desc       : integer;
  p_dest_desc     : integer;
  p_retry         : boolean;
  p_set_up_time   : integer;
  p_transit_time  : integer;
{*****************************************************************}
{*******    Declaration for the NetIPC intrinsics          ******}
{*****************************************************************}
PROCEDURE Addopt       ;INTRINSIC;
PROCEDURE Initopt      ;INTRINSIC;
PROCEDURE Readopt      ;INTRINSIC;
PROCEDURE IPCControl   ;INTRINSIC;
PROCEDURE IPCCreate    ;INTRINSIC;
PROCEDURE IPCDest      ;INTRINSIC;
PROCEDURE IPCConnect   ;INTRINSIC;
PROCEDURE IPCRecvcn    ;INTRINSIC;
PROCEDURE IPCRecv      ;INTRINSIC;
PROCEDURE IPCSend      ;INTRINSIC;
PROCEDURE IPCShutdown  ;INTRINSIC;
PROCEDURE IPCErrmsg    ;INTRINSIC;
PROCEDURE GETPRIVMODE  ;INTRINSIC;
PROCEDURE GETUSERMODE  ;INTRINSIC;
{******  Other intrinsics used in the programs          ******}
PROCEDURE quit         ;INTRINSIC;
FUNCTION timer:integer ;INTRINSIC;
{}
{}
```

```
{*****************************************************************}
{                                                                 }
{ SOURCE       :     CHECK                                        }
{                                                                 }
{ DESCRIPTION  :                                                  }
{  Simplified version.                                            }
{  This program  checks that connections to remote nodes or even  }
{  to local node can  be   actually achieved. It also allows to   }
{  estimate the performances of the network. It communicates with }
{  program X25SERV that runs on remote nodes.                     }
{  X25CHECK sends 10 times a message to the remote server which   }
{  echoes them back.                                              }
{  It checks for both connection and communication errors.        }
{  This version of X25CHECK is not compatible with the version of }
{  the product (doesn't work with the official server).           }
{  Compile in compatibility mode.                                }
{*****************************************************************}
$GLOBAL$
PROGRAM x25chk (input,output);
$include 'decl'$
FUNCTION ask_y_n : boolean;
var
  c : string [1];
begin {ask_y_n}
    repeat
      writeln;
      prompt ('Do you want to run the test once again?(y/n) >> ');
      readln (c);
    until (c='y') or (c='Y') or (c='n') or (c='N') or (c='');
    if (c='y') or (c='Y') then ask_y_n := true    else ask_y_n := false;
end;  {ask_y_n}
PROCEDURE check (result : integer;
                 event  : event_type);

var
  msg : string [80];
  len : integer;
  r   : integer;
begin   {check}
  IPCErrmsg  (result,msg,len,r);
  setstrlen (msg,len);
  if r <> 0 then
  begin
    writeln ('Can''t get the error message ...');
    QUIT (123);
  end
  else
  begin
    writeln ('An error occurred during ',c_event_msg [event]);
    writeln ('with the following identification : ');
    writeln (msg);
    p_retry := ask_y_n;
  end;
```

```
end;  {check}
{----------------------INIT_desc-----------------------------------------}

{ Create call descriptor with  dedicated protocol relative address       }
{ Create destination desc   to connect with the server                   }
{------------------------------------------------------------------------}
PROCEDURE init_desc ( var rc : rc_type);
var
  j,  prot_addr     : shint;
  opt           : opt_type;
  net_name,
  node_name     : string [8];
  net_name_len,
  node_name_len : shint;
begin
                                        {---------------------------------}
                                        { Creation of the call descriptor. }
                                        {---------------------------------}
    Initopt (opt,2,r);
    if r <> 0 then
    begin
       check (r,i_addopt);
       rc  := no_call_desc;
    end
    else
    begin  {initopt}
      prot_addr := c_prot_addr_x25chk;
      Addopt (opt,0,c_prot_add_code,2,prot_addr,r);
      if r <> 0 then
      begin
         check (r,i_addopt);
         rc  := no_call_desc;
      end
      else
      begin
        prompt('Enter the name of the network you are working on >> ');
        readln (net_name);
        net_name_len := strlen(net_name);
        Addopt (opt,1,c_net_name_code,net_name_len,net_name,r);
        if r <> 0 then
        begin
          check (r,i_addopt);
          rc := no_call_desc;
        end
        else
        begin
          IPCCreate (3,2,,opt,p_call_desc,result);
          if result <> 0 then
          begin
            check (result,i_create);
            rc := no_call_desc;
          end
          else
```

```
          begin
                                    {-----------------------------------}
                                    {Creation of the destination desc    }
                                    {-----------------------------------}
            writeln;
            prompt ('Enter the name of the node you want to check >> ');
            readln (node_name);
            node_name_len := strlen(node_name);
            prot_addr := c_prot_addr_server;
            IPCDest(3,node_name,node_name_len,2,prot_addr,2,,,
                     p_dest_desc,result);
            if result <> 0 then
            begin
               check (result,i_dest);
               rc := no_dest_desc;
            end;{else dest}
          end;{else create}
        end;{else addopt}
      end;{else addopt}
  end;{else initopt}
end;{init_desc}
{------------------------------CONNECT--------------------------------}
{ Send CALL to the server and wait for CALL CONF                       }
{ Evaluate  the set up time                                            }
{----------------------------------------------------------------------}
PROCEDURE connect ( var rc : rc_type);

var
    chrono : integer;
begin
    chrono := timer;
                                      {-----------------------------------}
                                      { Send CALL packet to remote server  }
                                      {-----------------------------------}
    IPCConnect (p_call_desc,p_dest_desc,,,p_vc_desc,result);
    if result <> 0 then
    begin
      check (result,i_connect);
      rc := no_vc_desc;
    end
    else
    begin
      writeln ('CALL packet sent ...');
                                      {-----------------------------------}
                                      {Get CALL CONF packet from the server}
                                      {-----------------------------------}
      IPCRecv (p_vc_desc,,,,,result);
      p_set_up_time := timer-chrono;
      if result <> 0 then
      begin
        check (result,i_recv_call_conf);
        rc := error;
      end
```

```
      else
      begin
        writeln ('CALL CONF packet received ...');
        writeln;
      end;
                                        {-----------------------------------}
                                        { The connection is now opened.     }
                                        {-----------------------------------}
    end; {else connect}
end; {connect}
PROCEDURE data_transfer ( var rc : rc_type);

var
  buffer        : buffer_type;
  buffer_len    : integer;
  chrono        : integer;
  i             : shint;
{------------------------DATA_TRANSFER-----------------------------}
{ PURPOSE : Manage  the data transfer with the server               }
{           Evaluate  the transit time                              }
{-----------------------------------------------------------------}
begin  {data transfer}
  i := 1;
  chrono := timer;
  while (i <= c_nb_loop) and (rc = done) do
  begin
    buffer     := c_patern;
    buffer_len := c_buffer_len;
                                        {-----------------------------------}
                                        { Send data packet on the line.     }
                                        {-----------------------------------}
    IPCSend (p_vc_desc,buffer,buffer_len,,,result);
    writeln ('DATA packet sent ...');
    if result <> 0 then
    begin
        check (result,i_send);
        rc := error;
    end
    else
    begin
                                        {-----------------------------------}
                                        { Receive data packet echoed by the  }
                                        { remote server.                    }
                                        {-----------------------------------}
     IPCRecv (p_vc_desc,buffer,buffer_len,,,result);
      writeln ('DATA packet received ...');
      writeln;
      if result <> 0 then
      begin
        check (result,i_recv);
        rc := error;
      end
      else
```

```
        i := i+1;
    end;{else send}
  end;{while}
  p_transit_time := timer - chrono;
end;{data transfer}
{-------------------------SHUTDOWN-----------------------------------}
{ PURPOSE : Shutdown call, destination and vc descriptor              }
{           according to the value of rc.                             }
{           Display the results of set up and transit time            }
{           Ask to retry                                              }
{-------------------------------------------------------------------}
PROCEDURE shutdown;
begin
    if rc <= error then
    begin
                                      {-----------------------------------}
                                      { Shutdown the vc descriptor.       }
                                      { Send CLEAR on the line.           }
                                      {-----------------------------------}
      IPCShutdown (p_vc_desc,,,result);
      if result <> 0 then check (result,i_shut_connection);
      writeln ('CLEAR packet sent ...');
    end;
    if rc <= no_vc_desc then
    begin
                                      {-----------------------------------}
                                      { Shutdown the destination desc.    }
                                      {-----------------------------------}
      IPCShutdown (p_dest_desc,,,result);
      if result <> 0 then check (result,i_shut_dest);
    end;
    if rc <= no_dest_desc then
    begin
                                      {-----------------------------------}
                                      { Shutdown the call descriptor.     }
                                      {-----------------------------------}
      IPCSHUTDOWN (p_call_desc,,,result);
      if result <> 0 then check (result,i_shut_source)
    end;
    if rc = done then
    begin
                                      {-----------------------------------}
                                      { Display the results.              }
                                      {-----------------------------------}
  writeln ('The following figures have been measured on the network :');
  writeln ('           Set up  time : ',p_set_up_time:10,' ms');
  writeln ('           Transit time : ',(p_transit_time/(c_nb_loop*2)):10:0,
                                        ' ms');
      p_retry := ask_y_n ;
    end;
end;{shutdown}
```

```
BEGIN
  p_retry := false;
  repeat
    rc := done;
    INIT_DESC (RC);
    if rc = done then
    begin
       CONNECT (rc);
       if rc = done then
       begin
          DATA_TRANSFER (rc);
       end;
    end;
    SHUTDOWN;
  until p_retry = false;
END.
{}
```

Program 3B (X25SERV)

```
{*****************************************************************}
{                                                                 }
{ SOURCE      :   X25SERV                                         }
{                                                                 }
{ DESCRIPTION :                                                   }
{                                                                 }
{ The purpose of that program is to answer to a remote program    }
{ X25CHECK which verifies that the connections have been actually }
{ established.                                                    }
{ The server receives messages and echoes them to the remote      }
{ calling node.                                                   }
{ The server has a dedicated protocol relative address.           }
{ This version of X25SERV is not compatible with the version of   }
{ the product.                                                    }
{ Compile in compatibility mode.                                  }
{*****************************************************************}

program x25serv (input,output);
$include 'decl'$ {include file of type and constants}
{---------------------------Check_init-----------------------------}
{ PURPOSE : Checks the results of IPC calls. Used during the initi- }
{           alization phase when errors are not discarded but dis-  }
{           played to the operator.                                 }
{                                                                   }
{-------------------------------------------------------------------}

PROCEDURE check_init (result:integer);
VAR
  msg     : string [80];
  msg_len : integer;
  r       : integer;
BEGIN
  if result <> 0 then
  begin
```

```
   IPCErrmsg (result,msg,msg_len,r);
    setstrlen(msg,msg_len);
    if r <> 0 then
    begin
      writeln('Can''t get the error message');
      QUIT (123);
    end{if}
    else
    begin
      writeln('X25SERV: error occurred during initialization of the');
      writeln('         server with the following identification:');
      writeln (msg);
      QUIT (125);
    end;{else}
  end;{if}
END;{check_init}
PROCEDURE create_descriptor;
var
  prot_addr    : shint;
  opt          : opt_type;
  net_name     : name_type;
  net_name_len : shint;
  wrtdata      : shint;
begin {create_descriptor}
                                    {-----------------------------------}
                                    { Creation of the descriptor dedicated}
                                    { to the server.                    }
                                    {-----------------------------------}
  Initopt (opt,2);
  prot_addr := c_prot_addr_server;
  Addopt (opt,0,c_prot_add_code,2,prot_addr,result);
  check_init (result);
  prompt ('Enter the name of the network you are working on >> ');
  readln (net_name);
  net_name := strltrim (net_name);
  net_name := strrtrim (net_name); {eliminates blanks}
                                   {useful when server is run from a stream}
  net_name_len:= strlen (net_name);
  Addopt (opt,1,c_net_name_code,net_name_len,net_name,result);
  check_init(result);
  IPCCreate (3,2,,opt,p_call_desc,result);
  check_init (result);
  writeln('Call descriptor : ',p_call_desc);
                                    {-----------------------------------}
                                    { Disable the timer on the call     }
                                    { descriptor.                       }
                                    {-----------------------------------}
  wrtdata := 0  ;
  IPCControl (p_call_desc,3,wrtdata,2,,,,result);
  check_init (result);
end; {create_descriptor}
```

```
PROCEDURE echo;
var
  opt             : opt_type;
  calling_address : packed array [1..16] of nibble;
  i,
  option_code,
  addr_len,
  data_len        : shint;
  buffer          : buffer_type;
  buffer_len      : integer;
begin {echo}
                                    {-----------------------------------}
                                    { Initialize an option field to get  }
                                    { the calling node address.          }
                                    {-----------------------------------}
  Initopt (opt,1);

  Addopt (opt,0,c_calling_add_code,8,calling_address,r);
                                    {-----------------------------------}
                                    { Wait for a connection request.     }
                                    { ie Incoming CALL.                  }
                                    {-----------------------------------}
  IPCRecvcn (p_call_desc,p_vc_desc,,opt,result);
  if result = 0 then
  begin
    writeln('Call Received.........');
                                    {-----------------------------------}
                                    { Get the calling address from the   }
                                    { CALL pkt.                          }
                                    {-----------------------------------}
    data_len := 8;
    option_code := c_calling_add_code;
    Readopt (opt,0,option_code,data_len,calling_address,r);
    writeln ('Calling node address = ');
    addr_len := calling_address [1];   {the first nibble contains the addr
len}
   for i:= 2 to addr_len+1 do write (calling_address [i]:1);
    writeln ;
                                    {-----------------------------------}
                                    { Loop on data transfer.             }
                                    {-----------------------------------}
    i:= 1;
    while (i <= c_nb_loop) and (result = 0) do
    begin
      buffer_len := c_buffer_len;
                                    {-----------------------------------}
                                    { Receive pkt from X25CHECK.         }
                                    {-----------------------------------}
      IPCRecv (p_vc_desc,buffer,buffer_len,,,result);
      if result = 0 then
      begin
        writeln('Data packet received.........');
```

```
                                        {-----------------------------------}
                                        { Echo the same buffer.             }
                                        {-----------------------------------}
        IPCSend (p_vc_desc,buffer,buffer_len,,,result);
        if result = 0 then i:=i+1;
      end;{if}
    end; {while}
  end;
end;{echo }

PROCEDURE shutdown_connection;
var
  buffer          : buffer_type;
  buffer_len      : integer;
begin
                                        {-----------------------------------}
                                        { End of connection.                }
                                        { Wait for X25CHECK to CLEAR first  }
                                        {-----------------------------------}
  if result = 0 then
  begin
    buffer_len := 1;
    IPCRecv (p_vc_desc,buffer,buffer_len,,,result);
                                        {-----------------------------------}
                                        { This IPCRECV should complete with }
                                        { an error indicating a CLEAR recvd.}
                                        {-----------------------------------}
    if result = c_clear_rcvd then
                                        {-----------------------------------}
                                        { We can shutdown the vc descriptor }
                                        {-----------------------------------}
      IPCShutdown (p_vc_desc,,,result);
  end;
end;{shutdown_connection}
PROCEDURE shutdown_call_desc;
begin {shutdown_call_desc}
  IPCShutdown (p_call_desc,,,result);
end;  {shutdown_call_desc}
begin   {main server}
  CREATE_DESCRIPTOR;
  while true do  {endless loop}
  begin
    ECHO;
    SHUTDOWN_CONNECTION;
  end;
  SHUTDOWN_CALL_DESC;
end.  {main server}
```

Example 4

Example 4 is a pair of programs designated SNMIPC1 and SNMIPC2 using direct access to X.25 level 3. These programs can be compiled in native mode. The program comments describe which of the X.25 features are used in these sample programs.

Program 4A (SNMIPC1)

```
(************************************************************************)
(* This program pair, SNMIPC1 and SNMIPC2, tests X25 features including:*)
(*                                                                      *)
(*    1. Call User Data up to 128 bytes                                 *)
(*    2. fast select                                                    *)
(*    3. special facilities                                             *)
(*    4. catch all socket                                               *)
(*    5. transfer of 2000 bytes packet                                  *)
(*    6. no address flag                                                *)
(*    7. deferred connection acceptance                                 *)
(*    8. display calling node's x25 address                             *)
(*                                                                      *)

(* Since this program uses the catch all socket, the NA capability is   *)
(* required.                                                            *)
(* To compile in Native Mode, compile with "pasxl SNMIPC1,,$null" and   *)
(* link with "link $oldpass, NMIPC1".  Run NMIPC1 before running NMIPC2.*)
(*                                                                      *)
(* CONFIGURATION ENVIRONMENT :                                          *)
(* Network Name  : DIRECT                                               *)
(* Facility Name : FACFULL (contains Fast Select flag)                  *)
(* SVCPATH       : POOL with IO security                                *)
(* X25 address   : 30101                                                *)
(* No Network Directory entries needed.                                 *)
(*                                                                      *)
(************************************************************************)

$stats off$                                    (* compiler option *)
$statement_number on$
$code_offsets on$
$tables on$
$lines 120$
$standard_level 'hp_modcal'$
$type_coercion 'conversion'$

program nmipc1 (input,output);

TYPE
   byte     =      0..255;            (* this is one byte long         *)
   bit4     =      0..15;             (* this is one nibble long       *)
   shortint = -32768..32767;          (* this is two bytes long        *)
```

```
CONST
   X25          =     2;              (* X25 protocol                  *)
   CUD_MAX      =   128;              (* number bytes of CUD           *)

VAR
   sd_local:     integer;            (* local socket descriptor       *)
   cd_local:     integer;            (* local connection descriptor   *)
   optioncode:  shortint;            (* optioncode return from readopt *)
   optlen:       integer;            (* opt length                    *)
   dlen:         integer;            (* data length                   *)
   flag:     packed array[1..4] of byte;  (* flags parameter           *)
   x25_flags: packed array[1..4] of byte;      (* x25 flags parameter   *)
   result:       integer;            (* back from IPC call            *)
   result16:     shortint;           (* back from opt calls           *)
   i:            integer;            (* loop counter for messages     *)
   msg : packed array[1..2000] of byte; (* message for send and receive *)
   data : packed array[1..12] of char;   (* send data                  *)
   opt:  packed array [0..500] of byte;(* options array                *)
   cud:  packed array [1..CUD_MAX] of byte; (* # bytes of CUD          *)
   wdata : packed array[1..80] of char; (* for ipccontrol wdata        *)
   readdata :  packed array[1..500] of byte; (* for ipccontrol readdata *)
   rlen  : integer;                  (* length for readdata           *)
   sf    : packed array[1..109] of byte; (* 109 bytes of facility_field *)
   net_name : packed array[1..8] of char;(* network name               *)
   cnaddr : packed array[1..8] of byte ; (* calling node address       *)

 (* IPC intrinsics used *)
procedure readopt;     intrinsic;
procedure addopt;      intrinsic;
procedure initopt;     intrinsic;
procedure ipccheck;    intrinsic;
procedure ipcconnect;  intrinsic;
procedure ipccontrol;  intrinsic;
procedure ipccreate;   intrinsic;
procedure ipcdest;     intrinsic;
procedure ipcerrmsg;   intrinsic;
procedure ipcget;      intrinsic;
procedure ipcgive;     intrinsic;
procedure ipcrecv;     intrinsic;
procedure ipcrecvcn;   intrinsic;
procedure ipcsend;     intrinsic;
procedure ipcshutdown; intrinsic;


(******************)
(*  Program start *)
(******************)
begin
writeln ('*** Program nmipc1 : X25 features test program ***');

(***************************** IPCCREATE *******************************)
(* initialize opt array entry *)
initopt ( opt, 2, result16 );
if result16 <> 0 then
```

```
   writeln ('initopt for ipccreate failed');

(* add the option for Catch_all Socket : bit 2 *)
flag[1] := 32;   (* flag : 01000000000000000000000000000000 *)
flag[2] := 0;
flag[3] := 0;
flag[4] := 0;
addopt ( opt, 0, 144, 4, flag, result16 );
if result16 <> 0 then
   writeln ('addopt for ipccreate catch-all failed');

(* add network name *)
net_name := 'direct';
addopt ( opt, 1, 140, 6, net_name, result16 );
if result16 <> 0 then
   writeln ('addopt for ipccreate network name failed');

writeln;
writeln('***** IPCCREATE start ');
ipccreate ( 3, X25, , opt, sd_local, result );
if result <> 0 then
   writeln ('ipccreate of local socket failed ',  result );

(********************** IPCRECVCN ******************************)
initopt ( opt, 4, result16 );
if result16 <> 0 then
   writeln ('initopt for ipcrecvcn failed');

(* Set CUD receive for IPCRECVCN *)
for i := 1 to CUD_MAX do              (* clean up CUD *)
   cud[i] := 0;
addopt(opt, 0, 5, CUD_MAX, cud, result16);
if result16 <> 0 then
   writeln('addopt IPCRECVCN CUD failed ',result16);

(* Set facility_field for IPCRECVCN *)
for i := 1 to 109 do              (* clean up SF *)
   sf[i] := 0;
addopt(opt, 1, 145, 109, sf, result16);
if result16 <> 0 then
   writeln('addopt IPCRECVCN SF failed', result16);

(* add calling node address *)
addopt(opt, 2, 141, 8,cnaddr,result16);
if result16 <> 0 then
   writeln('addopt IPCRECVCN calling node address  failed', result16);

(* add X25_flags for receive fast select : bit 7 = 1 *)
x25_flags[1] := 0;                 (* clean up X25_flags *)
x25_flags[2] := 0;
x25_flags[3] := 0;
x25_flags[4] := 0;
addopt(opt, 3, 144, 4,x25_flags,result16);
```

```
if result16 <> 0 then
   writeln('addopt IPCRECVCN x25 flags failed', result16);

(* set deferred flags for receive connection : bit 18 *)
flag[1] := 0;     (* flags : 00000000000000000100000000000000 *)
flag[2] := 0;
flag[3] := 32;
flag[4] := 0;                   (* deferred *)


writeln;
writeln('***** IPCRECVCN deferred');
ipcrecvcn ( sd_local, cd_local, flag, opt, result );
if result <> 0 then
   writeln ('ipcrecvcn failed', result);

(* check receive CUD *)
optlen := CUD_MAX;
readopt(opt,0,optioncode,optlen,cud,result16);
if result16 <> 0 then
   writeln('readdata failed (cud) ',result16:1);
if optlen <> CUD_MAX then
   writeln('CUD length error : (length = ',optlen:1, ' )');
i := 1;
while i <= optlen do
  begin
  if cud[i] <> i then
     writeln('CUD error : #',i:1,' ',cud[i]);
  i := i + 1;
  end;

(* check facilities *)
optlen := 109;

readopt(opt,1,optioncode,optlen, sf,result16);
if result16 <> 0 then
   writeln('readdata failed (special facility) ',result16:1);
writeln('facilities received in Incoming call packet (length =
',optlen:1,')');
for i := 1 to optlen do
    write (sf[i]:1,' ');
writeln;

(* check calling node address *)
optlen := 8;
readopt(opt,2,optioncode,optlen, cnaddr,result16);
if result16 <> 0 then
   writeln('readdata failed (calling node address) ',result16:1);
writeln('Calling Node Address (length = ',optlen:1,')');
for i := 1 to optlen do
    begin
    write ((cnaddr[i] div 16):1);
    write ((cnaddr[i] mod 16):1);
    end;
```

```
writeln;

(* check X25_flag : 00000001000000000000000000000000 *)
optlen := 4;
readopt(opt,3,optioncode,optlen,x25_flags,result16);
if result16 <> 0 then
   writeln('readdata failed (X25_flags) ', result16:1);
if (x25_flags[1] <> 1) and (x25_flags[2] <> 0) and
   (x25_flags[3] <> 0) and (x25_flags[4] <> 0) then
   writeln('error fast select flag should be set');

(********************** IPCCONTROL **********************************)
(* set pass parameter for IPCCONTROL *)
initopt(wdata, 2, result16);
if result16 <> 0 then
   writeln('initopt for ipccontrol failed');

(* send Call User Data back to calling node *)
for i := 1 to CUD_MAX do
    cud[i] := CUD_MAX - i;
addopt(wdata, 0, 2,CUD_MAX, cud, result16);
if result16 <> 0 then
   writeln('addopt for ipccontrol cud failed ',result16);

(* add special facility *)
sf[1] := hex('04');
sf[2] := hex('01');
addopt(wdata, 1, 145, 2, sf, result16);
if result16 <> 0 then
   writeln('addopt for ipccontrol sf failed ',result16);

(* IPCCONTROL to accept the connection *)
writeln;
writeln('***** IPCCONTROL accept ');
ipccontrol( cd_local, 9, wdata,500, , , , result);
if result <> 0 then
   writeln('ipccontrol failed ', result);


(***************** IPCRECV (2000 bytes) ****************************)
for i := 1 to 2000 do msg[i] := 0;    (* initialize msg and dlen *)
dlen := 2000;
writeln;
writeln('***** IPCRECV (2000 bytes data) ');
ipcrecv ( cd_local, msg, dlen, , , result );
if result <> 0 then
   writeln('IPCRECV 2000 failed ', result:1);
if dlen <> 2000 then
   writeln('error data length = ',dlen:1);

(* Check that the correct data was received in the first vector *)
i := 1;
while i <= dlen do
```

```
   begin
   if msg[i] <> (i mod 100) then
      writeln ('receive error data : #',i:1,' ',msg[i]);
   i := i + 1;
   end;


(************************ IPCSEND ***********************************)
(* Now send a single vectored message to the local side *)
data         := 'ok last one.';
dlen         := 12;
writeln;
writeln('***** IPCSEND last message');
ipcsend ( cd_local, data, dlen, , , result );
if result <> 0 then
   writeln ('ipcsend failed', result);

(************************ IPCRECV (skip error #67) *******************)
(* wait for remote side to shutdown first *)
(* receive an error code #67 *)
dlen := 1;
writeln;
writeln('***** IPCRECV ');
ipcrecv ( cd_local, data, dlen, , , result );
(* receive an error code #67 *)
if result <> 67 then
   writeln('IPCRECV failed ', result:1);


(************************ IPCSHUTDOWN ********************************)
(* shutdown the local connection descriptor *)
writeln;
writeln('***** IPCSHUTDOWN (cd)');
ipcshutdown ( cd_local, , , result );
if result <> 0 then
   writeln ('ipcshutdown cd_local failed', result);

(************************ IPCSHUTDOWN ********************************)
(* shutdown the local socket descriptor *)
writeln;
writeln('***** IPCSHUTDOWN (sd)');

ipcshutdown ( sd_local, , , result );
if result <> 0 then
   writeln ('ipcshutdown sd_local failed', result);

end.
```

Program 4B (SNMIPC2)

```
(************************************************************************}
(* This program pair, SNMIPC1 and SNMIPC2, tests X25 features including:*)
(*                                                                      *)
(*    1. Call User Data up to 128 bytes                                 *)
(*    2. fast select                                                    *)
(*    3. special facilities                                             *)
(*    4. catch all socket                                               *)
(*    5. transfer of 2000 bytes packet                                  *)
(*    6. no address flag                                                *)
(*    7. deferred connection acceptance                                 *)
(*    8. display calling node's x25 address                             *)
(*                                                                      *)
(* Since this program uses the catch all socket, the NA capability is   *)
(* required.                                                            *)
(* To compile in Native Mode, compile with "pasxl SNMIPC2,,$null" and   *)
(* link with "link $oldpass, NMIPC2".  Run NMIPC1 before running NMIPC2.*)
(*                                                                      *)
(* CONFIGURATION ENVIRONMENT :                                          *)
(* Network Name  : DIRECT                                               *)
(* Facility Name : FACFULL (contains Fast Select flag)                  *)
(* SVCPATH       : POOL with IO security                                *)
(* X25 address   : doesn't matter                                       *)
(* No Network Directory entries needed.                                 *)
(*                                                                      *)
(************************************************************************)

$stats off$                              (* compiler option *)
$statement_number on$
$code_offsets on$
$tables on$
$lines 120$
$standard_level 'hp_modcal'$
$type_coercion 'conversion'$


program nmipc2 (input,output);


TYPE
   byte     =      0..255;            (* this is one byte long         *)
   bit4     =      0..15;             (* this is one nibble long       *)
   shortint = -32768..32767;          (* this is two bytes long        *)

CONST
   X25          =     2;              (* X25 lever III protocol        *)
   CUD_MAX      =   128;              (* number byte of CUD            *)
   SOCK_ADDR    = 32000;              (* socket's address              *)

VAR
   sd_remote:    integer;             (* remote socket descriptor     *)
   cd_remote:    integer;             (* remote connection descriptor *)
```

```
   dd:           integer;              (* destination descriptor       *)
   dlen:         integer;              (* data length                  *)
   optlen:    integer;                 (* opt length                   *)
   optioncode:   shortint;             (* option code                  *)
   flag : packed array[1..4] of byte;   (* flag                         *)
   x25_flags:    packed array [1..4] of byte;    (*x25_flags parameter  *)
   result:       integer;              (* back from IPC call           *)
   result16:     shortint;             (* back from opt calls          *)
   i:            integer;              (* loop counter for messages    *)
   msg : packed array[1..2000] of byte; (* message for send and receive *)
   expect : packed array[1..12] of char;   (* expect data               *)
   adrs: packed array[1..2] of byte;    (* socket's address             *)
   opt:  packed array [0..500] of byte;  (* options array               *)
   xchico : packed array [1..50] of bit4;    (* chico Dest_net_addre    *)
   cud:       packed array [1..CUD_MAX] of byte; (* CUD                 *)
   data: packed array[1..12] of char;      (* receive data              *)
   readdata : packed array[1..200] of byte; (* readdata for readopt     *)
   sf    : packed array[1..109] of byte; (* 109 bytes of facility_field *)
   net_name : packed array[1..8] of char;   (* network name            *)
   fac_name : packed array[1..8] of char;    (* facility name           *)

(* IPC intrinsics used *)

procedure readopt;     intrinsic;
procedure addopt;      intrinsic;
procedure initopt;     intrinsic;
procedure ipccheck;    intrinsic;
procedure ipcconnect;  intrinsic;
procedure ipccontrol;  intrinsic;
procedure ipccreate;   intrinsic;
procedure ipcdest;     intrinsic;
procedure ipcerrmsg;   intrinsic;
procedure ipcget;      intrinsic;
procedure ipcgive;     intrinsic;
procedure ipcrecv;     intrinsic;
procedure ipcrecvcn;   intrinsic;
procedure ipcsend;     intrinsic;
procedure ipcshutdown; intrinsic;

(*****************)
(* Program start *)
(*****************)
begin
writeln ('### Program nmipc2 : X25 features test program ### ');

******************** IPCCREATE ************************************)


(* initialize opt array entry *)
initopt ( opt, 1, result16 );
if result16 <> 0 then
   writeln('ipccreate initopt failed', result16);

(* add the option for Network Name *)
```

```
net_name := 'direct';
(* add the option for NETWORK NAME *)
addopt (opt, 0, 140, 6, net_name, result16);
if result16 <> 0 then
   writeln('ipccreate addopt failed', result16);

writeln;
writeln('##### IPCCREATE ');
ipccreate ( 3, X25, , opt, sd_remote, result );
if result <> 0 then
   writeln ('ipccreate of remote socket failed', result);

(*************************** IPCDEST ********************************)
(* Get the destination descriptor to the local socket from the remote   *)
(* socket.                                                              *)
(* We are calling the catch-all socket, so no address will be put in the*)
(* call, however we have the satisfy IPCDEST with something             *)
adrs[1] := SOCK_ADDR div 256;        (* first 8 bits of 32000   *)
adrs[2] := SOCK_ADDR mod 256;        (* last 8 bits of 32000    *)
initopt ( opt, 1, result16 );
if result16 <> 0 then
   writeln ('initopt for ipcdest failed');

(* add DEST_NET_ADDR opt to ipcdest *)
xchico[1] := 0;    (* 0002 : protocol is X25 *)
xchico[2] := 0;
xchico[3] := 0;
xchico[4] := 2;
xchico[5] := 0;    (* 0000 : address for the SVC *)
xchico[6] := 0;
xchico[7] := 0;
xchico[8] := 0;
xchico[9] := 5;    (* 5    : length of X25 address *)
xchico[10] := 3;   (* 30101: chico X25 address *)
xchico[11] := 0;
xchico[12] := 1;
xchico[13] := 0;
xchico[14] := 1;
(* add remote node X25 address *)
addopt(opt, 0, 16, 7, xchico, result16);
if result16 <> 0 then
   writeln ('addopt for ipcdest failed',result16);
writeln;
writeln('##### IPCDEST ');
ipcdest ( 3, , , X25, adrs, 2, , opt, dd, result );
if result <> 0 then
   writeln ('ipcdest failed ' , result );

(********************* IPCCONNECT ********************************)
initopt ( opt, 4, result16 );
if result16 <> 0 then
   writeln ('initopt for ipcconnect failed',result16);
```

```
(* this sets the "no_address" flag; and allows to access 128 bytes of CUD *)
(* with fast select                                                        *)
x25_flags[1] := 0; (* X25_flags : 00000000000000001000000000000000 *)
x25_flags[2] := 0;
x25_flags[3] := 64;
x25_flags[4] := 0;
addopt(opt, 0, 144, 4, x25_flags, result16);
if result16 <> 0 then
   writeln ('addopt for ipcconnect x25_flag failed',result16);

(* add call_user_data_send *)
for i := 1 to CUD_MAX do
   cud[i] := i;
addopt(opt, 1, 2, CUD_MAX, cud, result16);
if result16 <> 0 then
   writeln ('addopt for ipcconnect cud failed',result16);

(* add facility name *)
fac_name := 'FACFULL ';
addopt(opt, 2, 142, 8, fac_name, result16);
if result16 <> 0 then
   writeln('addopt for facility name failed ',result16);

(* add some special facilities *)
sf[1] := hex('04');
sf[2] := hex('01');
addopt(opt, 3, 145, 2, sf, result16);
if result16 <> 0 then
   writeln ('addopt for ipcconnect sf failed',result16);

(* Connect to the local socket using the destination descriptor.         *)
writeln;
writeln ('##### IPCCONNECT');
ipcconnect ( sd_remote, dd, , opt, cd_remote, result );
if result <> 0 then
   writeln ('ipcconnect failed ', result );

(***************************** IPCRECV ****************************)
initopt(opt, 2, result16);
if result16 <> 0 then
   writeln('initopt for ipcrecv failed');
for i := 1 to CUD_MAX do
    cud[i] := 0;

optlen := CUD_MAX;
addopt(opt, 0, 5, optlen, cud, result16);
if result16 <> 0 then
   writeln('addopt IPCRECV cud  failed ', result16:1);
for i := 1 to 50 do
    sf[i] := 0;
```

```
optlen := 50;
addopt(opt, 1, 145, optlen, sf,result16);
if result16 <> 0 then
   writeln('addopt IPCRECV sf failed ', result16:1);
writeln;
writeln('##### IPCRECV ');
ipcrecv ( cd_remote, , , ,opt, result );
if result <> 0 then
   writeln ('ipcrecv to complete connection failed ',result);

(* check receive CUD *)
optlen := CUD_MAX;
readopt(opt, 0, optioncode, optlen,cud,result16);
if result16 <> 0 then
   writeln('CUD readopt failed ', result16:1);
if optlen <> CUD_MAX then
   writeln('CUD length error : (length = ',optlen:1, ' )');
i := 1;
while i <= optlen do
   begin
   if cud[i] <> (CUD_MAX - i) then
      writeln('CUD error : #',i:1,' ',cud[i]);
   i := i + 1;
   end;


(* check special facility *)
optlen := 50;readopt(opt, 1, optioncode, optlen,sf,result16);
if result16 <> 0 then
   writeln('SF readopt failed ',result16:1);
writeln
('facilities received in call accepted packet (length = ',optlen:1,')');
for i := 1 to optlen do
    write(sf[i]:1,' ');
writeln;

(***************************** IPCSEND (2000 bytes) ****************)
for i := 1 to 2000 do
    msg[i] := i mod 100;
writeln;
writeln('##### IPCSEND (2000 bytes data)');
ipcsend ( cd_remote, msg, 2000, , , result );
if result <> 0 then writeln('send 2000 bytes data failed ',result:1);

(***************************** IPCRECV ******************************)
dlen := 12;
writeln;
writeln('##### IPCRECV last message');
ipcrecv( cd_remote, data, dlen,,, result);
if result <> 0 then
   writeln('ipcrecv failed ',result);

(* check the correct data was received *)
```

```
if dlen <> 12 then
   writeln('receive data length error (length = ',dlen:1,')')
else
   begin
   expect := 'ok last one.';
   for i := 1 to dlen do
      if data[i] <> expect[i] then
          writeln('receive data error #',i:1,' ',data[i]);
   end;

(************************ IPCSHUTDOWN (cd) ************************)
(* shutdown the connection descriptor *)
writeln;
writeln('##### IPCSHUTDOWN (cd) ');
ipcshutdown ( cd_remote, , , result );
if result <> 0 then
   writeln ('ipcshutdown cd_local failed',result);

(************************ IPCSHUTDOWN (dd) ************************)
(* shutdown the connection descriptor *)
writeln;
writeln('##### IPCSHUTDOWN (dd) ');
ipcshutdown ( dd, , , result );
if result <> 0 then
   writeln ('ipcshutdown dd failed',result);

(************************ IPCSHUTDOWN (sd) ****************************)
(* shutdown the socket descriptor *)
writeln;
writeln('##### IPCSHUTDOWN (sd) ');
ipcshutdown ( sd_remote, , , result );
if result <> 0 then
   writeln ('ipcshutdown sd_local failed',result);
end.
```

A IPC Interpreter (IPCINT)

The IPC interpreter (IPCINT) is a software utility provided with the NS X.25/XL link product. IPCINT can be used as a learning tool for programmers and as a troubleshooting tool by network administrators.

IPCINT executes NetIPC intrinsics one at a time in response to commands typed at the keyboard. IPCINT can only be used for X.25 direct access to level 3.

Using IPCINT

To use IPCINT you must have an NS X.25 link up and running on a local HP 3000 node and on a remote node. In order to exercise the intrinsics between nodes, the remote node must be running either IPCINT or an X.25 direct access to level 3 server program.

You must have NA or NM capability to run IPCINT. To use IPCINT you enter `RUN IPCINT.NET.SYS` at the MPE XL prompt. At the IPCINT prompt (>) you can enter a NetIPC intrinsic abbreviation or `EX` to exit.

You will be prompted for parameters required for the intrinsic you specified. The intrinsic is executed by IPCINT and any output parameters or errors returned are displayed. IPCINT creates a log file called IPCLOG to contain the actions of each intrinsic executed.

Comparison of IPCINT to Programmatic NetIPC

The following examples show the difference between programmatic access and IPCINT used to execute the IPCCREATE intrinsic.

Example: Programmatic Access to X.25

For a program using direct access to X.25 level 3, a call to IPCCREATE can be specified as follows:

```
IPCCREATE (3,2,,opt,calldesc,result)
```

The value 3 for parameter *socketkind* specifies a call socket. The value 2 (for parameter *protocol*) indicates the protocol access is X.25. At a minimum, the *opt* array would contain the X.25 network name, and optionally either define a catch-all socket (*opt* code 144, bit 2) or specify a protocol relative address (*opt* code 128). The *calldesc* will contain the call socket descriptor, and result will contain an error (if any).

Example: IPCINT for X.25 Direct Access

For example, to execute the IPCCREATE intrinsic using IPCINT, enter CR from the IPCINT prompt (see example below). You are prompted for the IPCCREATE X.25 parameters. In this example, no catch-all socket is specified; therefore, a protocol relative address is specified. The network name is a required parameter. The network name X25NET is used in this example. After the required parameters are entered, press **[RETURN]** and the IPCCREATE intrinsic is executed.

```
CR
Protocol: 2
Catch All Socket (Y/N)? N
Protocol Relative Address: 3100
Network name (8 chars): X25NET
-----> Executing: IPCCREATE
CALL = 6
```

Syntax of IPCINT

The following paragraphs describe the syntax of IPCINT commands. This includes:

- Abbreviations for the intrinsics.
- Pseudovariables for socket descriptors.
- Prompts for parameters.
- Call user data field.

Abbreviated Intrinsic Names

The IPCINT program uses abbreviations for NetIPC intrinsics. Table A-1 shows the supported IPC intrinsics and the IPCINT abbreviations.

Table A-1 NetIPC Intrinsics IPCINT Abbreviations

Intrinsic	IPCINT Abbreviation
IPCCREATE	CR
IPCNAME	NAME
IPCNAMERASE	NAMERASE
IPCDEST	DEST
IPCGIVE	GIVE
IPCGET	GET
IPCCONNECT	CN
IPCRECVCN	RCN
IPCRECV	R
IPCSEND	S
IPCCONTROL	CTR
IPCSHUTDOWN	SHUT
IOWAIT	WAIT
IODONTWAIT	NOWAIT
IPCCHECK	CHECK
IPCERRMSG	ERR

Pseudovariables

Three pseudovariables are used by IPCINT to store the most recently assigned socket descriptors as follows:

```
Pseudovariable      socket descriptor
--------------      -----------------
      C                  call
      D                  destination
      V                  virtual circuit
```

The pseudovariable names can be overridden by the user.

Prompts for Parameters

When you enter the IPCINT abbreviation for the selected intrinsic, IPCINT prompts you for the required parameter values which you then enter from the keyboard. Default values are provided for most input parameters. The parameter names correspond approximately to those used in the reference portion of this manual. IPCINT prompts for X.25 *opt* array parameters without your having to use the INITOPT or ADDOPT intrinsics. You are also prompted for X.25 *flags* parameter bit settings. Prompts requiring a Y/N (yes/no) answer default to N (no).

Output parameters are displayed on the screen following execution of the called intrinsic.

Call User Data Field

The call user data field can be entered as a concatenated ASCII string enclosed in single quotes. Hexadecimal digits can be included in an ASCII string by preceding the digits with an h. For example, h45'hello'h10 can be entered which represents a string of hexadecimal 45, the word “hello” followed by hexadecimal 10.

Sample IPCINT Session

The following example describes the steps to create a call socket, send and receive data over a connection, and then close the socket using IPCINT on a local node. This sample session assumes a remote node is also using IPCINT. The remote node running IPCINT sends the local node a message as described in step 7.

The steps below follow the SVC requestor processing example in Figure 1-10 in Chapter 1 , “NetIPC Fundamentals.” The remote node should follow the steps in the SVC server processing example in Figure 1-11 in Chapter 3 , “NetIPC Intrinsics.”

User input is bold in the examples provided. For information about NetIPC intrinsic parameters refer to the intrinsic descriptions in Chapter 3 , “NetIPC Intrinsics.” Intrinsic parameter names that differ from the names used as prompts in IPCINT are included in parentheses in the discussion of the examples.

Step 1 Run the IPCINT program from the MPE XL prompt. A log of the session will be written to a file named IPCLOG.

```
(1)    :RUN IPCINT.NET.SYS

(C) COPYRIGHT Hewlett-Packard Company 1989

>>>>  IPC Interpreter   B020000     FRI, SEP 15, 1989,  9:59 AM
>
```

To exit IPCINT at any time enter EX at the IPCINT prompt (>).

Step 2 Enter the IPCINT abbreviation for the desired intrinsic (See Table A-1). In this example, CR for IPCCREATE is entered.

You are prompted for all required input parameters. You must enter 2 for X.25 direct access at the Protocol prompt. In this example, enter N (no) at the Catch All socket prompt (*opt* code 144, bit 2). Enter the network name configured for your network at the Network name (*opt* code 140) prompt.

After entering all required parameters, the intrinsic is executed. The call socket descriptor (*calldesc*) is returned in the pseudovariable “C”.The output parameters are interpreted and displayed. In this example, a call socket has been created.

```
(2)    CR

       Protocol: 2
       Catch All socket (Y/N)? N
       Network name (8 chars): X25net
       -----> Executing  :      IPCCREATE
       CALL =     12
```

Step 3

Execute the IPCDEST intrinsic by entering DEST at the prompt. You are prompted for the remote Node name (*location*) where the destination socket will be created. In this example, RAINBOW is used. If you leave the node name prompt blank, you will be prompted for the remote X.25 address expressed in hexadecimal.

Enter a protocol relative address (*protoaddr*) in the decimal range 30767 to 32767 for the remote address. In this example, **31000** is used. The IPCDEST intrinsic is executed and a destination descriptor (*destdesc*) will be returned in pseudovariable "D".

```
(3)     DEST

        Node name (50 chars): RAINBOW
        Protocol relative address (16 bit integer): 31000
        -----> Executing  : IPCDEST
        DEST = - 1
```

Step 4

In order to execute this step, the remote node server program or IPCINT must have already executed an IPCCREATE followed by an IPCRECVCN. The remote waits for the local to send the connection request. NetIPC provides a timeout so the IPCRECVCN will not wait indefinitely.

Execute IPCCONNECT by entering CN at the prompt. You are prompted for the call socket descriptor. To use the default, press **[RETURN]** which is the value returned in pseudovariable "C" by the previous call to IPCCREATE.

You are prompted for the destination socket descriptor. To use the default, press **[RETURN]** which is the value returned in pseudovariable "D" by the previous call to IPCDEST.

You are prompted for access to the call user data (CUD) field (*opt* 144, protocol flags, bit 17). In this example, Y (yes) is entered. Selecting "yes" allows you to enter up to 128 bytes of user data at the Call User Data (128 chars) prompt (*opt* code 2).

Next, you are prompted for a facility set name (*opt* code 142). To use the default configured for you network, press **[RETURN]**. At the Special Facility Field (*opt* code 145) prompt, enter up to 109 characters representing additional features to be added to the facility set. Press **[RETURN]** for no additions to the facility field.

The IPCCONNECT intrinsic is executed and a virtual socket descriptor is returned.

In the example, the statement, "No address in CUD" refers to the fact that you requested full access to the CUD.

```
(4)     N

        Source socket desc (32 bit integer/C/D/V): [RETURN]
        Destination desc (32 bit integer /C/D/V): [RETURN]
        No address in CUD (Y/N)? Y
        Call User Data (128 chars):hFCAA0001
        Facility name (8 chars): [RETURN]
        Special Facility Field (109 chars):[RETURN]
        -----> Executing  : IPCCONNECT
        VC =            7
        No address in CUD
```

Step 5 Execute IPCRECV by entering R at the prompt to receive the response to the previous connection request.

The default value for the VC socket descriptor is the value returned in the last IPCCONNECT (or in the case of an incoming call, by IPCRECVCN). This value is the default for any subsequent IPCSEND or IPCRECV calls.

To use default values, press **[RETURN]**. Buffer length (*dlen*) defaults to 4096 bytes. Preview data and Destroy data (*flags* 30 and 29) default to no (N). Data offset (*opt* code 8) is defaulted to none.

```
(5)     R

        Connect socket desc (32 bit integer /C/D/V): [RETURN]
        Buffer length (bytes): [RETURN]
        Preview data (Y/N)? [RETURN]
        Destroy data (Y/N)? [RETURN]
        Data offset (bytes): [RETURN]
        -----> Executing  : IPCRECV
        MAX_LEN = 4096
        RECV_LEN = 0
        BUFFER = ''
```

Note that there is no data returned in "Buffer" because the function of this call to IPCRECV is to accept the connection request from the remote node.

Step 6 Execute a call to IPCSEND by entering S at the prompt.

Enter a value for the buffer length. IPCINT will send a string of characters equal to the number of bytes specified. If you enter 0 for buffer length, you will be prompted to enter the contents of the data you are sending. You can specify up to 80 characters of data. At the Buffer prompt enter the data to send. In this example, 'Hello from local' is entered.

Pressing **[RETURN]** at the VC socket desc prompt which default to the VC socket descriptor returned by the previous call to IPCCONNECT (in this example). To use default values, press **[RETURN]**. Q bit set and D

bit set (*opt* code 144, bit 19 and bit 18) are defaulted to no (N). Data offset (*opt* code 8) defaults to none.

```
(6)     S

        Buffer length (bytes): 0
        Buffer: 'Hello from local'
        Connect socket desc (32 bit integer /C/D/V): [RETURN]
        Q bit set (Y/N): [RETURN]
        D bit set (Y/N)? [RETURN]
        Data offset (bytes): [RETURN]
        -----> Executing  : IPCSEND
```

In order for the remote node to receive the sent data, an IPCRECV must be executed from the remote node with IPCINT (or a server program).

Step 7

Before executing step 7, the remote must execute IPCSEND data to the local node (see step 6, IPCSEND).

Execute IPCRECV to receive data by entering R at the prompt. Step 7 assumes a remote node using IPCINT has sent you a message.

Press **[RETURN]** to use the default VC socket descriptor (*vcdesc*). To use default values, press **[RETURN]**. Buffer length defaults to 4096 bytes. Preview data and Destroy data (flags 30 and 29) default to no (N). Data offset (*opt* code 8) is defaulted to none.

Values returned by IPCRECV include data sent from the remote displayed at the prompt: Buffer = (data), length of the received data (dlen), and the buffer length input displayed as MAX_LEN (*dlen*, from input).

```
(7)     R

        Connect socket desc (32 bit integer /C/D/V: [RETURN]
        Buffer length (bytes): [RETURN]
        Preview data (Y/N)? [RETURN]
        Destroy data (Y/N)? [RETURN]
        Data offset (bytes): [RETURN]
        -----> Executing  : IPCRECV
        MAX_LEN = 4096
        RECV_LEN = 17
        BUFFER = 'Hello from remote'
```

Step 8

Execute IPCSHUTDOWN to shutdown the socket by entering SHUT at the prompt.

At the descriptor prompt, enter a descriptor (C, D or V) in order to indicate which socket needs to be shutdown. In this example, the VC socket descriptor, V is entered.

You are prompted for a reason code (*opt* code 143). In this example, 100 is entered which will cause a clear packet to be sent. The clear packet will contain a cause code zero (0), and diagnostic code 100. (IPCCONTROL is used to access cause and diagnostic codes.)

```
(8)     SHUT

        Descriptor (32 bit integer /C/D/V): V
        Reason code (16 bit integer): 100
        Call User Data (128 chars) :
        -----> Executing  : IPCSHUTDOWN
```

Step 9 Exit from the IPCINT program by entering EX at the prompt.

```
(9)     EX
```

B Cause and Diagnostic Codes

Cause and diagnostic can be read from X.25 packets using NetIPC intrinsics. The following tables show possible cause and diagnostic codes generated by the X.25 XL system access product. These codes are a subset of the CCITT (1984 X.25 recommendation) specified values.

Diagnostic Codes in X.25 Clear Packets

The following lists the diagnostic codes (in decimal) sent and received in X.25 clear packets. You can include cause and diagnostic codes with the IPCCONTROL or IPCSHUTDOWN intrinsics that will be included in the clear packet sent by the X.25 protocol. This function is only available with SVCs.

Table B-1 Diagnostic Codes Sent/Received in Clear Packets

Diagnostic Code	Meaning/Cause
0	No additional information
48	Timer expired
65	(1) Invalid facility code used. (2) the facility requested is not supported or allowed here: • Reverse charge in CALL CONF packet. • Fast select. • Throughout class negotiation (not configured). • Closed User Group facility in CALL CONF packet (not allowed). • Bilateral closed user group (not supported). • Packet size negotiation (not configured). • Window size negotiation (not configured). • RPOA facility (not supported).
70	Incoming call barred. The configuration does not allow a call from this address.
71	No logical channel available.
160	DTE specific. The facility set used does not allow acceptance of a reverse charge call
231	NSAP unreachable. The requested socket is unavailable.
233	Greater than 30,000 bytes of data were received.
241	A normal disconnect has occurred where the IPC user did not specify a diagnostic code.

Diagnostic Codes From a Remote Host

The following table lists diagnostic codes from a remote MPE XL system that could be received on the local system.

Table B-2 X.25 Diagnostic Codes From a Remote Host

Diagnostic Code	Meaning/Cause
48	Inactivity timer expired.
65	Facility not allowed.
70	Incoming call barred.
160	DTE specific for example: restricted access, unknown circuit, unknown local facility, circuit in use, destination not allowed, or reverse charge call not accepted.
161	DTE operational
162	DTE not operational. Level 3 out of order, or Level 2 out of order.
163	Network congestion.
166	D-bit procedure not supported

C Error Messages

This appendix includes the mapping of X.25 SOCKERRs to protocol module errors, and the complete table of possible NetIPC errors (SOCKERRs).

In the IPCCHECK intrinsic, both socket errors (SOCKERRs) and the corresponding protocol module errors (*pmerrs*) are returned. The following SOCKERRs are mapped to *pmerrs*. Other SOCKERRs can be returned to NetIPC with a corresponding *pmerr* of zero (0).

SOCKERR 46 **MESSAGE: PMERR = 5 Intrinsics : IPCCONNECT**

CAUSE: UNABLE TO INTERPRET RECEIVED PATH REPORT — Unable to find an X.25 address to call from the remote node name given.

ACTION: Check consistency between configuration file and network directory. In order to map the node name to the X.25 address, both the address key and the IP address are used.

SOCKERR 55 **MESSAGE: PMERR = 1 Intrinsic : IPCCREATE**

CAUSE: EXCEEDED PROTOCOL MODULE'S SOCKET LIMIT. — All call socket entries in the X.25 internal tables are in use.

ACTION: Remember to release call sockets when no IPCCONNECT and IPCRECVCN are expected.

MESSAGE: PMERR = 45 Intrinsic : IPCCONNECT

CAUSE: All connection entries in X.25 internal tables are in use.

ACTION: Remember to shut the VC's that are no longer in use.

SOCKERR 65 **MESSAGE: PMERR = 21 Intrinsic : IPCRECV, IPCSEND, IPCCONTROL**

CAUSE: CONNECTION ABORTED BY LOCAL PROTOCOL MODULE. — Greater than 30,000 bytes of data was received in a single message.

ACTION: Alter the remote application program to send smaller messages.

MESSAGE: PMERR = 36 Intrinsic : IPCRECV, IPCSEND, IPCCONTROL

CAUSE: The inactivity timer has timed out.

ACTION: Shutdown the connection before re-opening it.

SOCKERR 67 **MESSAGE: PMERR = 2 Intrinsic : IPCRECV, IPCSEND, IPCCONTROL**

CAUSE: CONNECTION FAILURE DETECTED. — A clear packet has been received. The remote system or network aborted the connection.

ACTION: Retrieve the cause/diagnostic field with IPCCONTROL (to examine the cause), and issue IPCSHUTDOWN on the virtual circuit.

SOCKERR 106 **MESSAGE: PMERR = 4 Intrinsic : IPCCREATE**

CAUSE: ADDRESS CURRENTLY IN USE BY ANOTHER SOCKET. — The requested protocol relative address is already used by another process through another IPCCREATE call.

ACTION: Use another protocol relative address or wait for previous process to release its call socket.

SOCKERR 107: **MESSAGE: PMERR = 7 Intrinsic : IPCCREATE, IPCRECV, IPCSEND, IPCCONTROL, IPCRECVCN**

CAUSE: TRANSPORT IS GOING DOWN. — A NETCONTROL STOP command has been issued. All connections have been aborted.

ACTION: Issue an IPCSHUTDOWN on all call sockets and virtual circuit sockets.

SOCKERR 116 **MESSAGE: PMERR = 13 Intrinsic : IPCCONNECT**

CAUSE: DESTINATION UNREACHABLE. — Outgoing access not allowed. The destination X.25 address is either not configured in the SVC path table or the security flags do not allow outbound calls to this address. It could also be a problem with the network directory.

ACTION: Check the configuration of SVC path table security and the network directory.

SOCKERR 117 **MESSAGE: PMERR = 17 Intrinsic : IPCRECV completing IPCCONNECT**

CAUSE: ATTEMPT TO ESTABLISH CONNECTION FAILED. — A connection could not be opened between the XL node and the DTC.

ACTION: Issue IPCSHUTDOWN on the virtual circuit, check the DTC XNA card is working, and re-issue IPCCONNECT.

SOCKERR 131 **MESSAGE: PMERR = 52 Intrinsic : IPCCONNECT**

CAUSE: PROTOCOL MODULE DOES NOT HAVE SUFFICIENT RESOURCES — The protocol module has run out of buffers.

ACTION: Try again later. Reduce network load by closing some connections or increase the number of buffers in the configuration.

SOCKERR 142 **MESSAGE: PMERR = 51 Intrinsic : IPCSHUTDOWN, IPCCONNECT**

CAUSE: INVALID CALL USER DATA OPT RECORD ENTRY — Too much call user data has been sent. The amount is determined by the “no address flag” and use of the fast select facility. The maximum call user data that can be sent is 128 bytes (with fast select).

ACTION: Send a smaller CUD, use fast select, or use the “no address” option.

SOCKERR 143 **MESSAGE: PMERR = 14 Intrinsic : IPCCONNECT**

CAUSE: INVALID FACILITIES SET OPT RECORD ENTRY — The facility set passed as a parameter has not been found in the internal facility set table.

ACTION: Use one of the facility sets defined in the configuration or add a new one.

SOCKERR 146 **MESSAGE: PMERR = 10 Intrinsic : IPCRECV**

CAUSE: RESET EVENT OCCURRED ON X.25 CONNECTION. — An unsolicited reset packet was received.

ACTION: Use IPCCONTROL (request 12) to examine the cause/diagnostic field. The connection is still up and operational but some data may have been lost.

SOCKERR 153 **MESSAGE: PMERR = 3 Intrinsic : IPCCREATE**

CAUSE: SOCKET IS ALREADY IN USE. — A single socket per network interface can be created with the catch-all capability.

ACTION: Wait for catch-all socket to be released.

SOCKERR 156 **MESSAGE: PMERR = 12 Intrinsic : IPCRECV**

CAUSE: INTERRUPT EVENT OCCURRED ON X.25 CONNECTION. — An interrupt packet was received.

ACTION: Use IPCCONTROL (request 12) to get interrupt data. The connection is still up and operational.

SOCKERR 158 **MESSAGE: PMERR = 18 Intrinsic : IPCRECV**

CAUSE: CONNECTION REQUEST REJECTED BY REMOTE. — The outgoing call was rejected either by the local DTC, the network, the remote DTC or the remote host.

ACTION: Use IPCCONTROL (request 12) to retrieve the cause/diagnostic field. Take action depending on cause/diagnostic using table given.

SOCKERR 160 **MESSAGE: PMERR = 24 Intrinsic : IPCSEND, IPCCONTROL**

CAUSE: INCOMPATIBLE WITH PROTOCOL STATE. — This connection is currently in the reset state. Either a reset was sent and the protocol is waiting for a reset confirmation, or a reset has been received.

ACTION: If you issued the reset, then wait and reissue the call later. Otherwise, issue IPCRECV> to complete an incoming reset.

NetIPC Errors

This section includes NetIPC error messages (SOCKERRs) and the form for submitting a service request (SR).

SOCKERRS

NetIPC are (32-bit) integers that are returned in the *result* parameter of NetIPC intrinsics when the intrinsic execution fails. (A result of 0 indicates that the intrinsic succeeded.) In addition, both NetIPC errors and Transport Protocol errors are returned in the IPCCHECK intrinsic: NetIPC errors in the *ipcerr* parameter and Transport Protocol errors in the *pmerr* parameter. Transport Protocol errors are documented in the *NS 3000/XL Error Message Reference Manual*.

0 **MESSAGE: SUCCESSFUL COMPLETION.**

CAUSE: No error was detected.

ACTION: None.

1 **MESSAGE: INSUFFICIENT STACK SPACE.**

CAUSE: Area between S and Z registers is not sufficient for execution of the intrinsic.

ACTION: :PREP your program file with a greater MAXDATA value.

3 **MESSAGE: PARAMETER BOUNDS VIOLATION.**

CAUSE: A specified parameter is out of bounds.

ACTION: Check all parameters to make certain they are between the user's DL and S registers. If an array is specified, make certain all of it is within bounds.

4 **MESSAGE:** TRANSPORT HAS NOT BEEN INITIALIZED.

CAUSE: A :NETCONTROL was not issued to bring up the network transport.

ACTION: Notify your operator.

5 **MESSAGE: INVALID SOCKET TYPE.**

CAUSE: Specified socket type parameter is of an unknown value.

ACTION: Check and modify your socket type parameter.

6 **MESSAGE: INVALID PROTOCOL.**

CAUSE: Specified protocol parameter is of an unknown value.

ACTION: Check and modify protocol parameter.

7 **MESSAGE: ERROR DETECTED IN flags PARAMETER.**

CAUSE: An unsupported bit in the flags parameter was set, or a nonprivileged user set a privileged bit.

ACTION: Verify that the proper bits are specified in the flags parameter. Bit numbering is from left to right (0..31).

8 **MESSAGE: INVALID OPTION IN THE opt RECORD.**

CAUSE: An unsupported option was specified in the *opt* record, or a nonprivileged user attempted to specify a privileged option.

ACTION: Check the options added to the *opt* record and remove or modify the option. Verify that the *opt* record was initialized correctly using INITOPT.

9 **MESSAGE: PROTOCOL IS NOT ACTIVE.**

CAUSE: A NETCONTROL has not been issued to activate the requested protocol module.

ACTION: Notify your operator.

10 **MESSAGE: PROTOCOL DOES NOT SUPPORT THE SPECIFIED SOCKET TYPE.**

CAUSE: The type of socket you are trying to create is not supported by the protocol to be used.

ACTION: Use a different socket type or protocol.

12 **MESSAGE: ERROR DETECTED WITH MAXIMUM MESSAGES QUEUED OPTION.**

CAUSE: Invalid option length specified or value of option is not positive.

ACTION: Correct option specification.

13 **MESSAGE: UNABLE TO ALLOCATE AN ADDRESS.**

CAUSE: No addresses were available for dynamic allocation.

ACTION: If unsuccessful the second time, see “Submitting an SR” at the end of this appendix.

14 **MESSAGE: ADDRESS OPTION ERROR.**

CAUSE: The address option in the *opt* record has an error in it (e.g., invalid length).

ACTION: Check the values being placed in the *opt* record.

15 **MESSAGE: ATTEMPT TO EXCEED LIMIT OF SOCKETS PER PROCESS.**

CAUSE: User has already reached the limit of 64 sockets per process.

ACTION: Shut down any sockets which are not being used or have been aborted.

16 **MESSAGE: OUT OF PATH DESCRIPTORS OR PATH DESCRIPTOR EXTENSIONS.**

CAUSE: Transport's Path Cache or Path Descriptor table is full.

ACTION: Contact your operator to see if the table can be expanded.

18 **MESSAGE: FORMAT OF THE opt RECORD IS INCORRECT.**

CAUSE: NetIPC was unable to parse the specified *opt* record.

ACTION: Check your INITOPT and ADDOPT calls.

19 **MESSAGE: ERROR DETECTED WITH MAXIMUM MESSAGE SIZE OPTION.**

CAUSE: Maximum message size option in the *opt* record had an error associated with it (e.g., too many bytes specified, invalid message size value).

ACTION: Check the values being placed in the *opt* record.

20 **MESSAGE: ERROR WITH DATA OFFSET OPTION.**

CAUSE: Data offset option in the *opt* record had an error associated with it (e.g., too many bytes specified).

ACTION: Check the values being placed in the *opt* record.

21 **MESSAGE: DUPLICATE opt RECORD OPTION SPECIFIED.**

CAUSE: The same *opt* record option was specified twice.

ACTION: Remove the redundant call.

24 **MESSAGE: ERROR DETECTED IN MAXIMUM CONNECTION REQUESTS QUEUED OPTION.**

CAUSE: Maximum connection requests queued option in the *opt* record had an error associated with it (e.g., too many bytes specified, bad value).

ACTION: Check the values being placed in the *opt* record.

25 **MESSAGE: SOCKETS NOT INITIALIZED; NO GLOBAL DATA SEGMENT.**

CAUSE: Error occurred attempting to initialize NetIPC, or Network Management is still initializing.

ACTION: See "Submitting an SR" at the end of this appendix.

26 **MESSAGE: UNABLE TO ALLOCATE A DATA SEGMENT.**

CAUSE: The attempt to create a data segment failed because the DST table was full or there was not enough virtual memory.

ACTION: Contact your operator to see if these tables can be expanded.

27 **MESSAGE: REQUIRED PARAMETER NOT SPECIFIED.**

CAUSE: A required parameter was not supplied in an option variable intrinsic call.

ACTION: Check your calling sequence.

28 **MESSAGE: INVALID NAME LENGTH.**

CAUSE: Specified name length was too large or negative.

ACTION: Check your name length parameter. Shorten the name if necessary.

29 **MESSAGE: INVALID DESCRIPTOR.**

CAUSE: Specified descriptor is not a valid socket, connection, or destination descriptor.

ACTION: Check the value being specified.

30 **MESSAGE: UNABLE TO NAME CONNECTION SOCKETS.**

CAUSE: The socket descriptor given in the IPCNAME call was for a VC socket; VC sockets may not be named.

ACTION: Check if the correct descriptor was specified.

31 **MESSAGE: DUPLICATE NAME.**

CAUSE: Specified name was previously given.

ACTION: Use a different name.

32 **MESSAGE: NOT CALLABLE IN SPLIT STACK.**

CAUSE: The particular NetIPC intrinsic cannot be called from split stack.

ACTION: Recode to call the intrinsic from the stack. Vectored data may be required.

33 **MESSAGE: INVALID NAME.**

CAUSE: Name is too long or has a negative length.

ACTION: Check the name's length. Shorten the name if necessary.

34 **MESSAGE: CRITICAL ERROR PREVIOUSLY REPORTED; MUST SHUTDOWN SOCKET.**

CAUSE: NetIPC previously detected and reported an irrecoverable error; most likely it was initiated by the protocol module.

ACTION: The socket can no longer be used. Call IPCSHUTDOWN to clean up.

35 **MESSAGE: ATTEMPT TO EXCEED LIMIT OF NAMES PER SOCKET.**

CAUSE: A socket can have only four names; the caller attempted to give it a fifth.

ACTION: Use no more than four names.

36 **MESSAGE: TABLE OF NAMES IS FULL.**

CAUSE: Socket registry or give table is full.

ACTION: Shut down unused sockets, call IPCNAMERASE on any sockets that no longer need to be looked up, or get given sockets. See "Submitting an SR" at the end of this appendix.

37 **MESSAGE: NAME NOT FOUND.**

CAUSE: Name was not previously specified in an IPCNAME or IPCGIVE call; IPCNAMERASE or IPCGET was previously issued with the name; or socket no longer exists.

ACTION: Check names specified, make sure names were properly agreed on, determine if a timing problem exists.

38 **MESSAGE: USER DOES NOT OWN THE SOCKET.**

CAUSE: Attempted to erase a name of a socket you do not own.

ACTION: Have the owner of the socket call IPCNAMERASE.

39 **MESSAGE: INVALID NODE NAME SYNTAX.**

CAUSE: Syntax of the node name is invalid.

ACTION: Check the node name being supplied.

40 **MESSAGE: UNKNOWN NODE.**

CAUSE: Unable to resolve the specified node name as an NS node name.

ACTION: Check the node name to see if it is correct. The node name may be valid but the specified node's transport may not be active.

41 **MESSAGE: ATTEMPT TO EXCEED PROCESS LIMIT OF DESTINATION DESCRIPTORS.**

CAUSE: User has already reached the limit of 261 destination descriptors per process.

ACTION: Call IPCSHUTDOWN on any unneeded destination descriptors.

43 **MESSAGE: UNABLE TO CONTACT THE REMOTE REGISTRY SERVER.**

CAUSE: Send to remote socket registry process failed. This is often caused by the fact that the PXP protocol module is not active on the local node.

ACTION: Contact your operator. If unable to resolve the problem, see "Submitting an SR" at the end of this appendix.

44 **MESSAGE: NO RESPONSE FROM REMOTE REGISTRY SERVER.**

CAUSE: No reply was received from the remote registry process. This is often due to the remote node not having initialized its transport.

ACTION: Contact your operator. If unable to resolve the problem, see "Submitting an SR" at the end of this appendix.

46 **MESSAGE: UNABLE TO INTERPRET RECEIVED PATH REPORT.**

CAUSE: Unable to interpret the information returned by the remote socket registry process regarding the looked-up socket.

ACTION: See "Submitting an SR" at the end of this appendix.

47 **MESSAGE: INVALID MESSAGE RECEIVED FROM REMOTE SERVER.**

CAUSE: The message received from the remote registry process does not appear to be a valid socket registry message.

ACTION: See "Submitting an SR" at the end of this appendix.

50 **MESSAGE: INVALID DATA LENGTH.**

CAUSE: Specified data length parameter is too long or negative.

ACTION: Check and modify the value.

51 **MESSAGE: INVALID DESTINATION DESCRIPTOR.**

CAUSE: Supplied destination descriptor value is not that of a valid destination descriptor.

ACTION: Verify that you are passing an active destination descriptor.

52 **MESSAGE: SOURCE AND DESTINATION SOCKET PROTOCOL MISMATCH.**

CAUSE: The source socket is not of the same protocol as the socket described by the destination descriptor.

ACTION: Validate that you are using the correct destination descriptor. Make certain both processes have agreed on the same protocol. Determine the correct socket was looked up.

53 **MESSAGE: SOURCE AND DESTINATION SOCKET TYPE MISMATCH.**

CAUSE: The source socket cannot be used for communication with the socket described by the destination descriptor.

ACTION: Validate that you are using the correct destination descriptor. Make certain both processes have agreed on the same method of communication. Determine the correct socket was looked up.

54 **MESSAGE: INVALID CALL SOCKET DESCRIPTOR.**

CAUSE: Specified descriptor is not for a call socket.

ACTION: Validate the value being passed.

55 **MESSAGE: EXCEEDED PROTOCOL MODULE'S SOCKET LIMIT.**

CAUSE: Protocol module being used cannot create any more sockets.

ACTION: Contact your operator; the limit may be configurable.

57 **MESSAGE: ATTEMPT TO EXCEED LIMIT OF NOWAIT SENDS OUTSTANDING.**

CAUSE: User tried to send data too many times in nowait mode without calling IOWAIT.

ACTION: Call IOWAIT to complete a send. The limit is 7.

58 **MESSAGE: ATTEMPT TO EXCEED LIMIT OF NOWAIT RECEIVES OUTSTANDING.**

CAUSE: User tried to issue too many consecutive nowait receives without calling IOWAIT.

ACTION: Call IOWAIT to complete a receive. The limit is 1.

59 **MESSAGE: SOCKET TIMEOUT.**

CAUSE: The socket timer popped before data was received.

ACTION: If this is not desired, call IPCCONTROL to increase or disable the timeout.

60 **MESSAGE: UNABLE TO ALLOCATE AN AFT.**

CAUSE: User has no space for allocating an Active File Table entry.

ACTION: Close unnecessary files or sockets.

62 **MESSAGE: CONNECTION REQUEST PENDING; CALL IPCRECV TO COMPLETE.**

CAUSE: User called IPCCONNECT without a subsequent IPCRECV before issuing the current request.

ACTION: Call IPCRECV.

63 **MESSAGE: WAITING CONFIRMATION; CALL IPCCONTROL TO ACCEPT/REJECT.**

CAUSE: IPCRECV called with deferred connection option. IPCCONTROL has not been called to accept/reject.

ACTION: Use the call IPCCONTROL with the accept/reject option.

64 **MESSAGE: REMOTE ABORTED THE CONNECTION.**

CAUSE: Remote protocol module aborted the connection. This will occur when a peer has called IPCSHUTDOWN on the connection.

ACTION: Call IPCSHUTDOWN to clean up your end of the connection.

65 **MESSAGE: CONNECTION ABORTED BY LOCAL PROTOCOL MODULE.**

CAUSE: Local protocol module encountered some error which caused it to abort the connection.

ACTION: Call IPCSHUTDOWN to clean up your end of the connection. See "Submitting an SR" at the end of this appendix.

66 **MESSAGE:** INVALID CONNECTION DESCRIPTOR.

CAUSE: Supplied value is not that of a valid VC socket (connection) descriptor.

ACTION: Check the value being given.

67 **MESSAGE: CONNECTION FAILURE DETECTED.**

CAUSE: An event occurred which caused the local protocol module to determine that the connection is no longer up (e.g., retransmitted data was never acknowledged).

ACTION: Call IPCSHUTDOWN to clean up your end of the connection.

68 **MESSAGE: RECEIVED A GRACEFUL RELEASE OF THE CONNECTION.**

CAUSE: Informational message.

ACTION: Do not attempt to receive any more data.

69 **MESSAGE: MUTUALLY EXCLUSIVE flags OPTIONS SPECIFIED.**

CAUSE: Bits in the *flags* parameter were set which indicate requests for mutually exclusive options.

ACTION: Check and clear the appropriate bits.

71 **MESSAGE: I/O OUTSTANDING.**

CAUSE: Attempted an operation with nowait I/O outstanding.

ACTION: Call IOWAIT to complete the I/O or to abort any receives.

74 **MESSAGE: INVALID IPCCONTROL REQUEST CODE.**

CAUSE: Request code is unknown or a nonprivileged user requested a privileged option.

ACTION: Validate the value being passed.

75 **MESSAGE:** UNABLE TO CREATE A PORT FOR LOW LEVEL I/O.

CAUSE: Unable to create an entity used for communication between NetIPC and the protocol module.

ACTION: See "Submitting an SR" at the end of this appendix.

76 **MESSAGE: INVALID TIMEOUT VALUE.**

CAUSE: Value specified for the timeout is negative.

ACTION: Modify the value.

77 **MESSAGE: INVALID WAIT/NOWAIT MODE.**

CAUSE: Mode of socket cannot be used.

ACTION: Use IPCCONTROL to specify correct mode.

78 **MESSAGE: TRACING NOT ENABLED**

CAUSE: Attempted to turn off trace when tracing was not on.

ACTION: Remove the call.

79 **MESSAGE: INVALID TRACE FILE NAME.**

CAUSE: Requested trace file name is not valid.

ACTION: Validate and modify the trace file name.

80 **MESSAGE: ERROR IN TRACE DATA LENGTH OPTION.**

CAUSE: An error was detected in the option specifying the maximum amount of data to be traced (e.g., negative value, too large, too many bytes used to specify the value).

ACTION: Modify the values being used.

81 **MESSAGE: ERROR IN NUMBER OF TRACE FILE RECORDS OPTION.**

CAUSE: An error was detected in the option specifying the maximum amount of records to be in the trace file (e.g., negative or too large a value, too many bytes used to specify the value).

ACTION: Modify the values being used.

82 **MESSAGE: TRACING ALREADY ENABLED.**

CAUSE: Attempted to turn on tracing when tracing already enabled.

ACTION: Remove the call or turn off trace before the call.

83 **MESSAGE: ATTEMPT TO TURN ON TRACE FAILED.**

CAUSE: The Node Management Subsystem (NMS) was unable to enable tracing.

ACTION: Call IPCCHECK; the protocol module error returned will be the Node Management error number. Refer to the Node Management Errors (NMERR) in the *NS 3000/XL Error Messages Reference Manual* to determine the appropriate action for the specified NMERR.

84 **MESSAGE: PROCESS HAS NO LOCAL SOCKET DATA STRUCTURES.**

CAUSE: IPCCHECK was called, but the user had no sockets or destination descriptors, and therefore no data structure for retaining error codes.

ACTION: None, but no NetIPC or protocol module errors are available.

85 **MESSAGE: INVALID SOCKET ERROR NUMBER.**

CAUSE: IPCERRMSG was called with an invalid NetIPC error code.

ACTION: Check the value being passed.

86 **MESSAGE: UNABLE TO OPEN ERROR CATALOG SOCKCAT.NET.SYS.**

CAUSE: The error message catalog does not exist, it is opened exclusively, or the caller does not have access rights to the file.

ACTION: Notify your operator.

87 **MESSAGE: GENMESSAGE FAILURE; NOT A MESSAGE CATALOG.**

CAUSE: MAKECAT was not successfully run on the message catalog.

ACTION: Notify your operator.

88 **MESSAGE: INVALID REQUEST SOCKET DESCRIPTOR**

CAUSE: Internal error

ACTION: See “Submitting an SR” at the end of this appendix.

89 **MESSAGE: INVALID REPLY SOCKET DESCRIPTOR**

CAUSE: Internal error

ACTION: See “Submitting an SR” at the end of this appendix.

91 **MESSAGE: WOULD EXCEED LIMIT OF REPLIES EXPECTED**

CAUSE: Internal error

ACTION: See “Submitting an SR” at the end of this appendix.

92 **MESSAGE: MUST REPLY TO BEFORE RECEIVING ANOTHER REQUEST.**

CAUSE: Internal error

ACTION: See “Submitting an SR” at the end of this appendix.

93 **MESSAGE: INVALID SEQUENCE NUMBER.**

CAUSE: Internal error

ACTION: See “Submitting an SR” at the end of this appendix.

94 **MESSAGE: NO OUTSTANDING REQUESTS.**

CAUSE: Internal error

ACTION: See “Submitting an SR” at the end of this appendix.

95 **MESSAGE: RECEIVED AN UNSOLICITED REPLY.**

CAUSE: Internal error

ACTION: See “Submitting an SR” at the end of this appendix.

96 **MESSAGE: INTERNAL BUFFER MANAGER ERROR.**

CAUSE: Attempted use of the buffer manager by NetIPC or the protocol module resulted in an error.

ACTION: See “Submitting an SR” at the end of this appendix.

98 **MESSAGE: INVALID DATA SEGMENT INDEX IN VECTORED DATA.**

CAUSE: Data segment index value in the vectored data array is not valid.

ACTION: Check the value being supplied.

99 **MESSAGE: INVALID BYTE COUNT IN VECTORED DATA.**

CAUSE: The count of data in the vectored data array is invalid.

ACTION: Check the values being given.

100 **MESSAGE: TOO MANY VECTORED DATA DESCRIPTORS.**

CAUSE: More than two data locations were specified in the vectored data array.

ACTION: Limit the number to two per operation. Use multiple sends or receives if necessary.

101 **MESSAGE: INVALID VECTORED DATA TYPE.**

CAUSE: Type of vectored data is unknown (must be a 0, 1, or 2) or the data type is for a data segment (1 or 2) and the user is not privileged

ACTION: Check the value being used.

102 **MESSAGE: UNABLE TO GRACEFULLY RELEASE THE CONNECTION**

CAUSE: Protocol module does not support graceful release, process tried to release connection that was not in the correct state, or output pending.

ACTION: Check command sequence.

103 **MESSAGE: USER DATA NOT SUPPORTED DURING CONNECTION ESTABLISHMENT.**

CAUSE: User data option is not supported for IPCRECV or IPCCONNECT.

ACTION: Do not use user data option.

104 **MESSAGE:** CAN'T NAME A REQUEST SOCKET

CAUSE: Internal error.

ACTION: See “Submitting an SR” at the end of this appendix.

105 **MESSAGE: NO REPLY RECEIVED.**

CAUSE: Internal error.

ACTION: See “Submitting an SR” at the end of this appendix.

106 **MESSAGE: ADDRESS CURRENTLY IN USE BY ANOTHER SOCKET.**

CAUSE: Address being specified for use is already being used.

ACTION: If you are a privileged user trying to specify a well known address, try again later. If you are nonprivileged, then see “Submitting an SR”.

107 **MESSAGE: TRANSPORT IS GOING DOWN.**

CAUSE: The transport is being shut down.

ACTION: Call IPCSHUTDOWN on all sockets and destination descriptors.

108 **MESSAGE: USER HAS RELEASED CONNECTION; UNABLE TO SEND DATA.**

CAUSE: Process tried to send after initiating a graceful release.

ACTION: Check command sequence.

109 **MESSAGE: PEER HAD RELEASED THE CONNECTION; UNABLE TO RECEIVE DATA.**

CAUSE: Process tried to receive after remote initiated graceful release.

ACTION: Check command sequence.

110 **MESSAGE: UNANTICIPATED ERROR.**

CAUSE: NetIPC received a protocol module error which it was unable to map.

ACTION: Call IPCCHECK to get the protocol module error. Call IPCSHUTDOWN to clean up. See “Submitting an SR” at the end of this appendix.

111 **MESSAGE: INTERNAL SOFTWARE ERROR DETECTED.**

CAUSE: An internal error was detected.

ACTION: See “Submitting an SR” at the end of this appendix.

112 **MESSAGE: NOT PERMITTED WITH SOFTWARE INTERRUPTS ENABLED.**

CAUSE: A request was made which cannot be performed with software interrupts enabled.

ACTION: Disable software interrupts or remove the request.

113 **MESSAGE: INVALID SOFTWARE INTERRUPT PROCEDURE LABEL.**

CAUSE: Procedure label passed when enabling software interrupts is invalid.

ACTION: Check the PLABEL you are passing.

114 **MESSAGE:** CREATION OF SOCKET REGISTRY PROCESS FAILED.

CAUSE: Socket registry program missing.

ACTION: Contact your HP representative for assistance.

116 **MESSAGE: DESTINATION UNREACHABLE.**

CAUSE: The transport was unable to route the packet to the destination.

ACTION: Notify your operator.

117 **MESSAGE: ATTEMPT TO ESTABLISH CONNECTION FAILED.**

CAUSE: Protocol module was unable to set up the requested connection. This may be caused by the remote protocol module not being active.

ACTION: Notify your operator.

118 **MESSAGE: INCOMPATIBLE VERSION.**

CAUSE: NetIPC software was incompatible with the software being executed by the remote registry process.

ACTION: Notify your operator.

119 **MESSAGE: ERROR IN BURST SIZE OPTION.**

CAUSE: An unsupported option was specified in the *opt* record, or a nonprivileged user attempted to specify a privileged option.

ACTION: Check your *opt* record and remove or modify the option.

120 **MESSAGE: ERROR IN WINDOW UPDATE THRESHOLD OPTION.**

CAUSE: An unsupported option was specified in the *opt* record, or a nonprivileged user attempted to specify a privileged option.

ACTION: Check your *opt* record and remove or modify the option.

124 **MESSAGE: ENTRY NUMBER NOT VALID FOR SPECIFIED OPT RECORD.**

CAUSE: User error. Entry number of option is either negative or higher than specified in INITOPT value.

ACTION: Correct and reissue command.

125 **MESSAGE: INVALID OPTION DATA LENGTH.**

CAUSE: User error. Data length for option is either negative or higher than specified in INITOPT value.

ACTION: Correct and reissue command.

126 **MESSAGE: INVALID NUMBER OF EVENTUAL OPT RECORD ENTRIES.**

CAUSE: Number of option entries is either too high or negative. Either an internal restriction or a user mistake.

ACTION: Remove the cause by making the number positive or smaller in value.

127 **MESSAGE: UNABLE TO READ ENTRY FROM OPT RECORD.**

CAUSE: The option record indicates that the entry is not valid or the buffer supplied by the user was too small to hold all of the data.

ACTION: Check entry number, make sure the option record has not been written over and check output buffer length.

131 **MESSAGE: PROTOCOL MODULE DOES NOT HAVE SUFFICIENT RESOURCES.**

CAUSE: Protocol module is temporarily out of buffers or internal data descriptors.

ACTION: Retry later when the system load is lighter.

141 **MESSAGE: X.25 NETWORK NAME INCORRECTLY SPECIFIED**

CAUSE: Invalid X.25 network name specified or not configured.

ACTION: Correct the network name or notify the operator.

142 **MESSAGE: INVALID CALL USER DATA OPT RECORD ENTRY.**

CAUSE: The length of the call user data is invalid for the transport protocol type.

ACTION: Check the length of the call user data option in the *opt* array. The call user data *opt* record must be greater than 1 for IPCCONNECT and 4 for IPCRECVCN. The maximum length is protocol specific.

143 **MESSAGE: INVALID FACILITIES SET OPT RECORD ENTRY**

CAUSE: The facility set passed as a parameter has not been found in the internal facility set table.

ACTION: Use one of the facility sets defined in the configuration or add a new one

144 **MESSAGE: INVALID CALLING NODE OPT ENTRY.**

CAUSE: The user may request the address of the calling node. Address of 8 bytes will be returned.

ACTION: The length of the option entry must be exactly 8 bytes.

146 **MESSAGE: RESET EVENT OCCURRED ON X.25 CONNECTION**

CAUSE: An unsolicited reset packet was received.

ACTION: Use IPCCONTROL (request 12) to examine the cause/diagnostic field. The connection is still up and operational but some data may have been lost.

151 **MESSAGE: COULD NOT OBTAIN A SEMAPHORE.**

CAUSE: The attempt to obtain a semaphore before sending a message to the protocol module failed.

ACTION: See “Submitting an SR” at the end of this appendix.

153 **MESSAGE: SOCKET IS ALREADY IN USE.**

CAUSE: A single socket per network interface can be created with the catch-all capability.

ACTION: Wait for catch-all socket to be released.

155 **MESSAGE: INVALID X.25 FLAGS OPT RECORD ENTRY.**

CAUSE: Invalid flag bits set in protocol specific flags option, or invalid length specified for option.

ACTION: Check bits set and length specified. Bit numbering is from left to right (0..31).

156 **MESSAGE: INTERRUPT EVENT OCCURRED ON X.25 CONNECTION**

CAUSE: An unsolicited interrupt packet was received.

ACTION: Use IPCCONTROL (request 12) to get interrupt data. The connection is still up and operational.

158 **MESSAGE: CONNECTION REQUEST REJECTED BY REMOTE.**

CAUSE: The remote node received the connection request and rejected it.

ACTION: The call may be retried later. Otherwise, the reason for the reject must be known.

160 **MESSAGE: INCOMPATIBLE WITH PROTOCOL STATE.**

CAUSE: The user requested an operation which is not supported by the protocol module.

ACTION: Verify the sequence of intrinsic calls.

163 **MESSAGE: PERMANENT VIRTUAL CIRCUIT ALREADY ESTABLISHED.**

CAUSE: A connection request was issued on a PVC which is in use by another process.

ACTION: Select a different PVC or retry later.

164 **MESSAGE:** ADDRESS VALUE IS OUT OF RANGE.

CAUSE: Address specified in *opt* parameter is out of range.

ACTION: Specify an address in the range 30767 to 32767

165 **MESSAGE: INVALID ADDRESS LENGTH.**

CAUSE: An invalid address length was specified in the *opt* parameter.

ACTION: The address length is 2 bytes (for non-privileged users).

166 **MESSAGE: CONNECTION NOT IN VIRTUAL CIRCUIT WAIT CONFIRM STATE.**

CAUSE: Attempt was made to accept or reject a connection that is open or in the process of closing.

ACTION: Use flags parameter in IPCRECVCN to defer acceptance or rejection of the connection request.

167 **MESSAGE: TIMEOUT NOT ALLOWED ON SHARED CONNECTION.**

CAUSE: Attempt to set a send time out on a shared connection.

ACTION: Use IPCCONTROL to disallow sharing of the connection or do not attempt to set send time out on this connection

171 **MESSAGE: INVALID FACILITY FIELD.**

CAUSE: For IPCCONNECT, IPCRECVCN, or IPCRECV, the *opt* parameter "facility field length" is wrong.

ACTION: Check the facility field length. The length may be 1 to 109 bytes inclusive.

172 **MESSAGE: CONNECTION MUST BE REJECTED.**

CAUSE: An IPCCONTROL request 9, accept the connection, cannot be performed because fast select restricted has been configured.

ACTION: Use IPCCONTROL request 15 to reject the connection.

173 **MESSAGE: MORE DATA IS AVAILABLE.**

CAUSE: Warning message. READOPT request was for less data than available.

ACTION: Specify a greater length in READOPT.

Submitting an SR

For further assistance from Hewlett-Packard, document the problem as an SR (Service Request) and forward it to your Hewlett-Packard Service Representative. Include the following information where applicable:

- A characterization of the problem. Describe the events leading up to and including the problem. Attempt to describe the source of the problem. Describe the symptoms of the problem and what led up to the problem.

  Your characterization should include: MPE XL commands; communication subsystem commands; job streams; result codes and messages; and data that can reproduce the problem.

  Illustrate as clearly as possible the context of any message(s). Prepare copies of information displayed at the system console and user terminal.

- Obtain the version, update and fix information for all software using `NMMAINT.PUB.SYS`. This allows Hewlett-Packard to determine if the problem is already known, and if the correct software is installed at your site.

- Record all error messages and numbers that appear at the user terminal and the system console.

- Run `NMDUMP.PUB.SYS` to format the NM log file that was active when the problem occurred (`NMLGnnnn.PUB.SYS`). You may need to issue the MPE XL command `SWITCHNMLOG` to free the NM log file.

  Using NMDUMP, format the log file for NETXPORT (3), NETIPC (5), Network Services (6) and Link Manager (8) information. Inspect the formatted output and try to locate errors. Prepare the formatted output and a copy of the log file for your Hewlett-Packard representative to further analyze.

- Prepare a listing of the configuration file and the MPE XL I/O configuration you are using for your Hewlett-Packard representative to further analyze. Inspect the output and try to locate errors.

- Try to determine the general area within the software where you think the problem exists. Refer to the appropriate reference manual and follow the guidelines on gathering information for problems:

  — The *NS 3000/XL User/Programmer Reference Manual* for NS 3000/XL.

  — The *Online Diagnostics Subsystem Utilities* for the IEEE 802.3 links.

— The *HP 36923 Central Bus Programmable Serial Interface Installation and Reference Guide* for the Point-to-Point link.

- Issue the `LINKCONTROL linkname; STATUS=` for each link. Retain the output for your Hewlett-Packard representative to further analyze.
- Document your interim, or "workaround" solution. The cause of the problem can sometimes be found by comparing the circumstances in which it occurs with the circumstances in which it does not occur.
- Create copies of any NS 3000/XL or NetIPC user trace, Network Transport trace and communication link trace files that were active when the problem occurred for your Hewlett-Packard representative to further analyze.
- If the problem involves NMMGR, give a copy of `NMMGRF.PUB.SYS` to your Hewlett-Packard representative.
- In the event of a system failure, a full memory dump must be taken. Always send the unformulated memory dump, a listing of the configuration file, a copy of the file `LOADMAP.PUB.SYS`, and the I/O configuration.

D Migration From PTOP to NetIPC and RPM

The PTOP (Program-to-Program communication) service is available on DS/3000 and NS 3000/V. It is a master-slave protocol wherein a master process creates a slave process on a remote node, the master sends data-exchange requests to the slave, and the slave accepts or rejects the master requests.

The NetIPC (Network InterProcess Communication) and RPM (Remote Process Management) services are available on NS 3000/V and NS 3000/XL. These services provide a more flexible alternative to PTOP for developing distributed applications. Since PTOP is not supported on NS 3000/XL, any PTOP application to be migrated to an MPE XL system must be rewritten to use NetIPC and RPM. These guidelines provide steps for the conversion.

A distributed application using PTOP or NetIPC/RPM can be divided into three phases:

- Creation of the remote (slave) process and the establishment of a communications channel between the local (master) and the remote (slave) processes.
- Data exchange over the communications channel.
- Process termination and the shut down of the communications channel.

The rest of this appendix is divided according to creating processes, exchanging data, and terminating processes. The sections are then subdivided according to what you have to do on a master program and on a slave program.

Creating Remote Processes

With PTOP, the creation of the slave process and the set up of the communications channel is done by the POPEN call in the master and the first GET and ACCEPT (or REJECT) calls in the slave. The POPEN call specifies the remote node's location, the name of the program on the remote node to be created, and various process-creation parameters. The *dsnum* parameter returned by POPEN identifies the slave process and its communications channel. On the slave side, the GET and ACCEPT calls complete the set up.

With NetIPC and RPM, the actions of creating the slave process and setting up the communications channel are split up. RPM handles the process creation, while NetIPC handles the communications. The calls necessary for these tasks are more complicated than the POPEN and GET/ACCEPT calls. The master process creates a call socket and registers it in the local socket registry. It passes the name of the socket and the local node name to the slave by using the RPM string feature of RPMCREATE. The master calls RPMCREATE to create the slave, including any process-creation parameters. The slave retrieves the master socket and node names from the RPM strings. It creates its own socket, looks up the master's socket, and establishes a virtual-circuit connection between the two sockets. The steps for each side are given below.

Creating Remote Processes: In the Master Program

To convert the PTOP intrinsic listed below, perform the following steps.

Syntax

```
dsnum := POPEN (location,progname [,itag][,entryname]
[,param][,flags][,stacksize][,dlsize][,maxdata][,bufsize])
```

- Get the local node name. You can use the NSINFO intrinsic, provided that you execute a DSLINE or a REMOTE command prior to the call. This should normally be the case for a PTOP application. You can use local node name and length item numbers (19 and 18) in NSINFO.

NOTE NSINFO is supported on NS 3000/V starting with UB-delta-1 MIT and on NS 3000/XL with Release 1.1.)

- Create a TCP call socket for the master program and name the socket, using the IPCCREATE and IPCNAME intrinsics. You can use a randomly generated name from IPCNAME.
- Build the opt array for the RPMCREATE call, using the INITOPT and ADDOPT intrinsics. In the opt array, include RPM strings (opt code

20000) for the socket name and the local node name. (These will be used by the slave program to set up the virtual circuit connection.) If any process-creation options are included in the POPEN call (*entryname*, *param*, *flags*, *stacksize*, *dlsize*, or *maxdata*), include them in the opt array with the corresponding RPM opt codes:

Parameter	Opt Code
entry name	22001
param	22002
flags	22003
stacksize	22004
dlsize	22005
maxdata	22006

- Call RPMCREATE to create the slave process on the remote node. Use the *progname* and *location* parameters as they appear in the POPEN call, although you will have to supply the (byte) lengths of *progname* and location. You should set the dependent bit of the *flag* parameter, so the slave will terminate if the master does. Save the program descriptor, returned by RPMCREATE, for a future RPMKILL.

NOTE RPMCREATE requires the program to be linked with Process Handling (PH) capability. PTOP does not require PH capability.

- Call IPCRECVCN to wait for the connection request from the slave. Save the returned virtual-circuit descriptor for subsequent IPCSEND and IPCRECV calls.
- Now that the virtual-circuit connection has been set up, call IPCSHUTDOWN to delete the master's call socket and IPCNAMERASE to delete the socket name.
- If a tag is specified, call IPCSEND to send the tag on the virtual circuit to the slave.
- If the slave could respond to the POPEN with either an ACCEPT or a REJECT, call IPCRECV to receive a one-byte accept or reject indication from the slave.
- If a tag is specified, call IPCRECV to receive the tag from the slave (but see the comment on IPCRECV under “Exchanging Data”).

Creating Remote Processes: In the Slave Program

To convert the PTOP intrinsic listed below, perform the following steps.

Syntax

```
ifun := GET [([itag][il][,ionumber]
```

- Call RPMGETSTRING twice to get the master's socket name and node name.
- Create a TCP call socket by using the IPCCREATE intrinsic.
- Call IPCLOOKUP to look up the master's socket, using the master socket name and node name passed in the RPM strings. This returns a destination descriptor to be used in the IPCCONNECT call.
- Set up a virtual-circuit connection between the master and the slave sockets, using IPCCONNECT. Call IPCRECV to wait for the connection acknowledgment from the slave. Save the returned virtual-circuit descriptor for subsequent IPCSEND and IPCRECV calls.
- After setting up the connection, delete the call socket and destination descriptor using IPCSHUTDOWN.
- If a tag is specified, call IPCRECV to receive the master's tag on the virtual circuit (see the comment on IPCRECV under Data Exchange).

To convert the PTOP intrinsic listed below, perform the following steps.

Syntax

```
ACCEPT [([itag][,target][,tcount])]
```

- If the slave can call either ACCEPT or REJECT in response to the POPEN, send a one byte accept or reject indication on the virtual circuit to the master.
- If a tag is specified, call IPCSEND to send the slave's tag on the virtual circuit back to the master.

Exchanging Data

The PTOP data-exchange calls are master-slave. The master initiates an exchange by calling PWRITE (to send data to the slave), PREAD (to receive data from the slave) or PCONTROL. The slave calls GET to receive the request and ACCEPT to perform the actual data movement into or out of its buffer. Each of these operations may also exchange a tag between the master and the slave. The master call sends the master tag to the slave and returns a tag from the slave. The slave's GET call receives the master tag. The slave's ACCEPT or REJECT call returns the slave's tag to the master.

The GET intrinsic can return an indication of the master request (0 = error, 1 = POPEN, 2 = PREAD, 3 = PWRITE, 4 = PCONTROL). In many applications, the slave will always know the next request that it will receive from the master, so the function returned by GET is superfluous. But in some applications, the slave does not know in advance what the next master request will be, and so it depends on the GET function to decide on its response to the request. The GET call can also return the requested length of data to be sent or received. These applications typically have a loop with a GET that receives the master requests and a case statement with cases for each of the different functions.

Data exchange with NetIPC is peer-to-peer. A process on either side of a virtual-circuit connection can send or receive data independently from its partner. The master-slave data exchange of PTOP can be simulated using NetIPC calls. For example, a PWRITE in the master can be replaced by an IPCSEND, while the corresponding GET and ACCEPT in the slave can be replaced by an IPCRECV. Tags can also be exchanged using sends and receives.

In applications where the sequence of master requests is not known by the slave, or where the length of data sent to or received from the slave is not known, some information in addition to the exchanged data and tags may need to be transmitted. This includes a) a master request indication, b) master request lengths, and c) the slave accept or reject indication.

Because of the way that NetIPC stream mode operates on the HP 3000, an IPCRECV may not receive all of the data requested. For this reason, we recommend that you write a procedure that calls IPCRECV in a loop to receive chunks of data until the entire requested amount is received. An example of this is in Chapter 4 , “NetIPC Examples,” in this manual.

Exchanging Data: In the Master Program

To convert the PTOP intrinsic listed below, perform the following steps.

```
lgth := PREAD (dsnum,target,tcount[,itag]
```

- If the slave requires a master function, send the PREAD request function (=2) on the virtual circuit to the slave.
- If the slave requires the requested data length, send *tcount* on the virtual circuit.
- If a tag is specified, send the master tag on the virtual circuit.
- If the master needs to know the actual data length sent from the slave, receive the actual data length from the virtual circuit.
- If the slave may call either ACCEPT or REJECT, receive a one byte accept or reject indication on the virtual circuit from the slave. Otherwise assume the slave accepted the request.
- If the slave accepted the request, receive the target data from the slave, using either a predetermined length or the actual data length received in Step 4.
- If a tag is specified, receive the slave tag from the virtual circuit.

To convert the PTOP intrinsic listed below, perform the following steps:

Syntax

```
PWRITE(dsnum,target,tcount[,itag]
```

- If the slave requires a master function, send the PWRITE request function (=3) on the virtual circuit to the slave.
- If the slave requires the requested data length, send *tcount* on the virtual circuit.
- If a tag is specified, send the master tag on the virtual circuit.
- If the slave may call either ACCEPT or REJECT, receive a one byte accept or reject indication on the virtual circuit from the slave, Otherwise assume the slave accepted the request.
- If the slave accepted the requested, send the target data on the virtual circuit to the slave, using the *tcount* length.
- If a tag is specified, receive the slave tag from the virtual circuit.

To convert the PTOP intrinsic listed below, perform the following steps.

Syntax

```
PCONTROL (dsnum[,itag]
```

- If the slave requires a master function, send the PCONTROL request function (=4) on the virtual circuit to the slave.
- If a tag is specified, send the master tag on the virtual circuit, and receive the slave tag from the virtual circuit.

Exchanging Data: In the Slave Program

To convert the PTOP intrinsic listed below, perform the following steps.

Syntax

```
ifun := GET [([itag][,il][,ionumber]
```

- If the slave requires a master function, receive a one byte master function number from the virtual circuit from the master.
- If the slave requires the request length, receive the length from the virtual circuit.
- If a tag is specified, receive the master tag from the virtual circuit.

To convert the PTOP intrinsic listed below, perform the following steps.

Syntax

```
ACCEPT [([itag[,target][,tcount)]
```

- Depending on the master function, either known to the application, or received from the master in Step (1):

PREAD: Send the target data, of length fncttcount, on the virtual circuit to the master.

PWRITE: Receive the data into target, using either the known fncttcount or the length received from the master in Step (2).

- If a tag is specified, send the slave tag on the virtual circuit.

Terminating Processes

The final phase is the termination of the slave process, and the shutting down of the communications channel between the master and the slave. With PTOP, both of these functions are performed by the PCLOSE intrinsic. With NetIPC and RPM, the process termination is done by RPMKILL, while the channel is shut down by IPCSHUTDOWN. In both PTOP and NetIPC/RPM, clean up on the slave is done automatically upon its termination.

To convert the PTOP intrinsic listed below, perform the following steps.

Syntax

PCLOSE (*dsnum*)

- Call RPMKILL to terminate the slave by using the program descriptor returned by RPMCREATE.
- Call IPCSHUTDOWN to shutdown the master's end of the virtual-circuit connection.

Example: Client-Server Application

The following sets of programs illustrate the principles for converting a PTOP application to use NetIPC and RPM.

The sample application is a simple name server, where you run a client program that creates a server on the node that contains a data file. The client program sends names to the server. The server looks up the names in the data file and returns associated information to the client.

The client and server are first presented as PTOP master and slave programs. Then they are converted to use NetIPC and RPM.

The major points of the conversion are:

- The `POPEN` call made by the client is replaced by the NetIPC and RPM calls as detailed in the earlier section “Creating Remote Processes.” At the beginning of the server, the corresponding NetIPC and RPM calls are inserted.
- The client’s `PWRITE` of the name to the server is replaced by an `IPCSEND`. The server’s `GET` and `ACCEPT` are replaced by an `IPCRECV` (actually, one or more `IPCRECV`s in the `RECV` procedure).
- The server can `ACCEPT` or `REJECT` the client’s `PREAD` for the name information, depending on whether the name is found in the data file. So, in the converted application, the server sends an accept or reject indication to the client. The `ACCEPT` is replaced by an `IPCSEND` of the name information.
- The accepted `PREAD` in the client becomes an `IPCRECV` for the name information.
- The client’s `PCLOSE` is replaced by an `RPMKILL` and `IPCSHUTDOWN`.

PCLIENT: Sample PTOP Master Program

```
$standard_level 'HP3000', uslinit$ program pclient( input, output );
{------------------------------------------------------------------------}
{                                                                        }
{ PCLIENT: Sample PTOP Master Program                                    }
{                                                                        }
{------------------------------------------------------------------------}
{                                                                        }
{ PURPOSE:                                                               }
{ The PCLIENT and PSERVER programs illustrate the use of the PTOP        }
{ service to implement a simple name server application.  The user       }
{ runs PCLIENT on his local node, and PCLIENT creates PSERVER on         }
{ the node which contains the data.  The user inputs names to            }
{ to PCLIENT, PCLIENT sends the names to PSERVER, PSERVER                }
{ looks up the names in its name file, and sends the associated info     }
{ for the names back to PCLIENT.                                         }
{                                                                        }
{------------------------------------------------------------------------}
{                                                                        }
{ INTERACTION:                                                           }
{ PTOP is a master-slave protocol.  The master PCLIENT sends             }
{ requests (PREAD and PWRITE) to the slave PSERVER.  The slave           }
{ GETs the request from the master and either ACCEPTs them or            }
{ REJECTs them.  The GET indicates the function requested by the         }
{ master, and the ACCEPT transfers the actual data, from the master      }
{ for a PWRITE and to the master for a PREAD.  REJECT can be used        }
{ to reject the master request (used here if the name cannot be          }
{ found in the data file.                                                }
{    PCLIENT                              PSERVER
{
{    get remote node name                                                }
{    POPEN PSERVER on remote node ------> GET                            }
{                                <-------  ACCEPT (POPEN)                }
{    get name                                                            }
{    PWRITE name ---------name----------> GET                            }
{                < ---------------------  ACCEPT name                   }
{    PREAD info  ----------------------->  GET                           }
{                                         look up name, found info       }
{                <---------info---------  ACCEPT info                   }
{    print info                                                          }
{    get name                                                            }
{    PWRITE name ---------name----------> GET                            }
{                < ----------------------ACCEPT                         }
{    PREAD info  ----------------------->  GET                           }
{                                         look up name, not found        }
{                < ----------------------REJECT                         }
{    print error                                                         }
{    . . .                                                               }
{    PCLOSE      ----------------------->  GET                           }
{                                         (terminate)                    }
{------------------------------------------------------------------------}
```

```
label 1;                        {for error exit      }
    const maxnodelength = 51;  {all lengths in bytes}
          maxproglength = 24;
          namelength    = 20;
          infolength    = 60;
          ccg           =  0;  {condition codes     }
          ccl           =  1;
          cce           =  2;
    type  shortint      = -32768..32767;
          msgtype       = packed array[1..30] of char;
    var   location: packed array [1..maxnodelength] of char;
          progname: packed array [1..maxproglength] of char;
          name:     packed array [1..namelength   ] of char;
          info:     packed array [1..infolength   ] of char;
          dsnum:    shortint;
          length:   shortint;

function  POPEN:
  shortint; intrinsic; {PTOP master intrinsics}
procedure PWRITE;            intrinsic;
function  PREAD:   shortint; intrinsic;
procedure PCLOSE;            intrinsic;
function  PCHECK:  shortint; intrinsic;

procedure ERROR( msg: msgtype; errnum: shortint );

   {-----------------------------------------------------------------}
   { ERROR prints out an error message and associated PTOP error     }
   { number, and then goes to the error exit to terminate the        }
   { program.  The PTOP slave will be terminated automatically.      }
   {-----------------------------------------------------------------}
 begin
    writeln( 'Client: ', msg, 'PTOP error = ', errnum:3 );
    goto 1;
 end;

begin
    prompt('Client: Enter the remote node name: ');
    readln( location );
      {-----------------------------------------------------------------}
      { Create PSERVER slave on remote node (location).  This           }
      { requires a previous REMOTE HELLO for the remote node.           }
      {-----------------------------------------------------------------}
    progname := 'PSERVER ';
    dsnum := POPEN( location, progname );
    if ccode <> cce then
       ERROR( 'POPEN on server failed', PCHECK(0) );
{-------------------------------------------------------------}
{ Each pass of this loop gets a name, PWRITEs it to PSERVER,  }
{ PREADs the info, and prints the info.  If PSERVER cannot    }
{ find the name, it will REJECT the PREAD.                    }
{-------------------------------------------------------------}
    repeat
       prompt('Client: Enter name (or EOT to exit):');
```

```
      readln( name );
      if name <> 'EOT' then
         begin
         PWRITE( dsnum, name, -namelength );
         if ccode <> cce then
            ERROR( 'PWRITE to server failed.', PCHECK(dsnum) );
         length := PREAD( dsnum, info, -infolength );
         if ccode = cce then       {ACCEPT}
            writeln('Client data is: ', info )
         else if ccode = ccg then  {REJECT}
            writeln('Client data could not be found.')
         else  {ccode = ccl}
            ERROR( 'PREAD from server failed.', PCHECK(dsnum) );
         end;
   until name = 'EOT';
     {-------------------------------------------------------------}
     { All names have been processed.  Terminate the PSERVER.      }
     {-------------------------------------------------------------}
   PCLOSE( dsnum );
   if ccode <> cce then
      ERROR( 'PCLOSE on server failed.', PCHECK(dsnum) );
 1: {error exit}
end.
```

PSERVER: Sample PTOP Slave Program

```
 Standard_level 'HP3000', uslinit$
program pserver( input, output                                          }
{-----------------------------------------------------------------------}
{                                                                       }
{ PURPOSE:                                                              }
{ The PCLIENT and PSERVER programs illustrate the use of the PTOP}
{ service for a simple name server application.  See the PCLIENT    }
{ program for details.                                                  }
{                                                                       }
{-----------------------------------------------------------------------}
   label 1;                 {for error exit      }
    const namelength = 20;  {all lengths in bytes}
          infolength = 60;
          cce        =  2;  {condition code      }
    type  shortint   = -32768..32767;
          msgtype    = packed array[1..30] of char;
          nametype   = packed array[1..namelength] of char;
          infotype   = packed array[1..infolength] of char;
    var   name:        nametype;
          info:        infotype;
          func:        shortint;
          found:       boolean;

function  GET:     shortint; intrinsic;  {PTOP slave intrinsics}
 procedure ACCEPT;           intrinsic;
 procedure REJECT;           intrinsic;
 function  PCHECK: shortint; intrinsic;

procedure ERROR( msg: msgtype; errnum: shortint );
   {-----------------------------------------------------------------}
   { ERROR prints an error message and an associated PTOP error     }
   { number.  It terminates the program by going to the error exit. }
   {-----------------------------------------------------------------}
 begin
    writeln( 'Server: ', msg, 'PTOP error = ', errnum:3 );
    goto 1;
 end; {ERROR}
procedure FIND_NAME    ( var reqname: nametype;
                     var info:    infotype;
                  ar found:   boolean );

{-----------------------------------------------------------------}
{ FIND_NAME sequentially searches the data file for the requested}
{ name.  It returns an indication of whether the name was found, }
{ and if it was found, the information field for the name.  (In  }

{ a real name server, a more efficient look up method would be   }

{ used.)                                                         }

{-----------------------------------------------------------------}
    var filename: packed array[1..9] of char;
        datafile: text;
        name:     nametype;
```

```
 begin
    filename := 'DATAFILE ';
    reset( datafile, filename );
    found := false;
    while not found and not eof(datafile) do
       begin
       readln( datafile, name, info );
       if name = reqname then
          found := true
       end;
 end; {FIND_NAME}

begin
      {---------------------------------------------------------------}
      { Each pass of this loop GETs one master request and ACCEPTs  }
      { or REJECTs the request, based on the type of request.  The  }
      { loop continues until the master issues its PCLOSE to        }
      { terminate the slave.                                        }
      {---------------------------------------------------------------}
    repeat
       func := GET;
       case func of
          0:{error}
             ERROR( 'Bad GET in server', PCHECK(0) );
          1:{POPEN}
             begin
             ACCEPT;
             if ccode <> cce then
                ERROR( 'ACCEPT for POPEN failed', PCHECK(0) );
             end;
          2:{PREAD}
             begin

               {-------------------------------------------------}
               { Look up name from previous PWRITE.  If the name    }
               { is found, ACCEPT the PREAD with the name info.     }
               { If the name is not found, REJECT the PREAD.        }
               {-------------------------------------------------}
             FIND_NAME( name, info, found );
             if found then
                begin
                ACCEPT( , info, -infolength );
                if ccode <>  cce then
                   ERROR( 'ACCEPT for PREAD failed', PCHECK(0) );
                end

else
                begin
                REJECT;
                if ccode <> cce then
                   ERROR( 'REJECT for PREAD failed', PCHECK(0) );
                end;
             end;
          3:{PWRITE}
```

```
            begin
              {-----------------------------------------------------}
              { ACCEPT the name from the PWRITE.  This name will    }
              { be used in the case for the following PREAD.        }
              {-----------------------------------------------------}
            ACCEPT( , name );
            if ccode <> cce then
               ERROR( 'ACCEPT for PWRITE failed', PCHECK(0) );
            end;
       end;
    until false;
1:{error exit}
 end.
```

RCLIENT: Sample NetIPC/RPM Master Program

```
$standard_level 'HP3000', uslinit$ program rclient( input, output );
{------------------------------------------------------------------------}
{                                                                        }
{ RCLIENT: Sample NetIPC/RPM Master Program                              }
{                                                                        }
{------------------------------------------------------------------------}
{                                                                        }
{ PURPOSE:                                                               }
{ The RCLIENT and RSERVER programs illustrate the use of the RPM         }
{ and NetIPC services to implement a simple name server application.     }
{ The user runs RCLIENT on his local node, and RCLIENT creates           }
{ RSERVER on the node which contains the data.  The user inputs          }
{ names to RCLIENT, RCLIENT sends the names to RSERVER,                  }
{ RSERVER looks up the names in its name file and sends the associated}
{ info for the names back to RCLIENT.                                    }
{                                                                        }
{ The RCLIENT and RSERVER programs are converted from the                }
{ PCLIENT and PSERVER programs, which use PTOP to                        }
{ implement the name server.                                             }
{ The PTOP-to-RPM/NetIPC conversion guidelines in the beginning          }
{ of this appendix were used.                                            }
{                                                                        }
{------------------------------------------------------------------------}
{                                                                        }
{ INTERACTION:                                                           }
{ The original PTOP implementation of the name server used a master-     }
{ slave relationship between the client and server.  The client          }
{ sends requests, and the server can accept or reject the requests.      }
{ This relationship is preserved in the NetIPC/RPM implementation.       }
{ RCLIENT must first create RSERVER and they must set up a virtual       }
{ circuit connection between them.  RCLIENT creates and names a          }
{ call socket.  It then calls RPMCREATE to create RSERVER, passing       }
{ the client's socket name and node name as RPM strings in the opt       }
{ array.  When it is created, RSERVER retrieves the client's socket      }
{ and node name, creates its own socket, looks up the client's           }
{ socket, and establishes a connection between its socket and the        }
{ client's socket.  At this point, the client and server are ready       }
{ to exchange data.                                                      }
{ For each input name, RCLIENT sends the name to RSERVER.  RSERVER       }
{ looks up the name in its data file.  If the name is found, RSERVER     }
{ sends a one byte "accept" indication back to RCLIENT, followed by      }
{ the name information.  If the name is not found, RSERVER sends a       }
{ "reject" indication to RCLIENT.  This simulates the original use       }
{ of ACCEPT and REJECT in the PTOP implementation.                       }
( RCLIENT                                  RSERVER                       }
{    NSINFO for client node name                                         }
{    IPCCREATE socket 1                                                  }
{    IPCNAME socket 1, clientsock                                        }
{    ADDOPT rpmstring, clientsock                                        }
{    ADDOPT rpmstring, clientnode                                        }
{    get server node name                                                }
```

```
{    RPMCREATE RSERVER on server                                          }
{       node      ----------------------> RPMGETSTRING clientsock        }
{    IPCRECVCN socket 1                   RPMGETSTRING clientnode        }
{    .                                    IPCCREATE socket 2             }
{    .                                    IPCLOOKUP clientsock,          }
{    .                                             clientnode,           }
{    .                                             dest                  }
{    .           <-----------------------IPCCONNECT socket 2, dest      }
{    .           ----------------------> IPCRECV                        }
{    IPCNAMERASE clientsock               IPCSHUTDOWN socket 2           }
{    IPCSHUTDOWN socket 1                 IPCRECV name                   }
{    get name                             .                              }
{    IPCSEND name---------name----------> .                              }
{    IPCRECV ind                          look up name, found info       }
{    .           <-----indaccept-------- IPCSEND indaccept              }
{    IPCRECV info < -------info-----------IPCSEND info                   }
{    print info                           IPCRECV name                   }
{    get name                             .                              }
{    IPCSEND name---------name----------> .                              }
{    IPCRECV ind                          look up name, not found        }
{    .          < -----indreject-------- IPCSEND indreject              }
{    print error                          IPCRECV name                   }
{    . . .                                .                              }
{    RPMKILL     ----------------------> .                               }
{    IPCSHUTDOWN vc                       (terminate)                    }
{                                         (IPCSHUTDOWN vc)               }
{                                                                        }
{------------------------------------------------------------------------}
```

```
label 1;
    const maxnodelength   = 51;   {all lengths in bytes   }
          maxproglength   = 24;
          namelength      = 20;
          infolength      = 60;
          clocalnodelength= 18;   {NSINFO item number     }
          clocalnode      = 19;   {NSINFO item number     }
          callsocket      =  3;   {IPCCREATE socket type  }
          tcpprotocol     =  4;   {IPCCREATE protocol type}
          socketnamelength=  8;   {created by IPCNAME     }
          maxoptlength    = maxnodelength + socketnamelength + 20;
          dependent       = 31;   {RPMCREATE flags bit    }
          optrpmstring    = 20000;{RPMCREATE opt number   }
          indaccept       =  0;   {accept indication      }
          indreject       =  1;   {reject indication      }

type  shortint        = -32768..32767;
          byte            = 0..255;
          msgtype         = packed array [1..30] of char;
          buftype         = array [1..80] of char;
    var   clientnode:       packed array [1..maxnodelength] of char;
          clientsockname:   packed array [1..socketnamelength] of char;
          location:         packed array [1..maxnodelength] of char;
          progname:         packed array [1..maxproglength] of char;
          name:             packed array [1..namelength   ] of char;
```

```
          info:             packed array [1..infolength   ] of char;
          opt:              packed array [1..maxoptlength ] of char;
          rpmflags:         packed array [0..31] of boolean;
          progdesc:         array [1..8] of shortint;
          buf:              buftype;
          clientnodelength: shortint;
          loclength:        shortint;
          prognamelength:   shortint;
          socketdesc:       integer;
          vcdesc:           integer;
          status:           shortint;
          result:           integer;
          envnum:           shortint;
          i:                integer;

procedure NSINFO;          intrinsic;  {NS intrinsic     }
procedure IPCCREATE;       intrinsic;  {NetIPC intrinsics}
procedure IPCNAME;         intrinsic;
procedure IPCNAMERASE;     intrinsic;
procedure IPCRECVCN;       intrinsic;
procedure IPCSEND;         intrinsic;
procedure IPCRECV;         intrinsic;
procedure IPCSHUTDOWN;     intrinsic;
procedure INITOPT;         intrinsic;
procedure ADDOPT;          intrinsic;
procedure RPMCREATE;       intrinsic;  {RPM intrinsics   }
procedure RPMKILL;         intrinsic;
procedure ERROR( msg: msgtype; result: integer );

   {-----------------------------------------------------------------}
   { ERROR prints out an error message and an associated NetIPC or  }
   { RPM result code, and then goes to the error exit to terminate  }
   { the program.  Because the server was created with the dependent}
   { flag, the server will automatically terminate.  Any NetIPC     }
   { objects (socket, socket name, or virtual circuit) will also be }
   { deleted at termination.                                        }
   {-----------------------------------------------------------------}
 begin
    writeln( 'Client: ', msg, 'Result = ', result:3 );
    goto 1;
 end;

procedure RECV(      vcdesc: integer;
                 var buf:    buftype;
                     length: integer;
                 var result: integer );
    var nextbufchar: integer;
        recvlength:  integer;
```

```
   {-------------------------------------------------------------}
   { RECV receives a specified number of bytes from the virtual  }
   { circuit (vc) connection.  This compensates for the stream mode }
   { operation of NetIPC on the HP 3000, where an IPCRECV can return}
   { less than the requested number of bytes.  The loop in RECV  }
   { calls IPCRECV to receive the next chunk of data, until the  }
   { requested amount of data has been received.  Note that buf  }
   { must be unpacked to allow it to be indexed in the IPCRECV call.}
   {-------------------------------------------------------------}
 begin
    result      := 0;
    nextbufchar := 1;
    while (length > 0) and (result = 0) do
       begin
       recvlength := length;
       IPCRECV( vcdesc, buf[nextbufchar], recvlength, , , result );
       nextbufchar := nextbufchar + recvlength;
       length      := length - recvlength;
       end;
 end; {RECV}

begin
      {----------------------}
      { Get client node name. }
      {----------------------}
NSINFO( , , envnum, status,
            clocalnodelength, clientnodelength,
            clocalnode, clientnode );
    if status  <> 0 then
       ERROR( 'Couldn't get client node name.', status );
      {-------------------------------------------------------------}
      { Create and name client's socket.  The socket length of 0 in }
      { IPCNAME will cause it to return a random 8-byte socket name.}
      {-------------------------------------------------------------}
    IPCCREATE( callsocket, tcpprotocol, , , socketdesc, result );
    if result  <> 0 then
       ERROR( 'Couldn't create local socket.', result );
    IPCNAME( socketdesc, clientsockname, 0, result );
    if result  <> 0 then
       ERROR( 'Couldn't name client socket.', result );

      {-------------------------------------------------------------}
      { Build the opt array for the RPMCREATE call, including RPM   }
      { strings for the client's socket name and node name.         }
      {-------------------------------------------------------------}

INITOPT( opt, 2 );
    ADDOPT ( opt, 0, optrpmstring, socketnamelength, clientsockname );
    ADDOPT ( opt, 1, optrpmstring, clientnodelength, clientnode );
      {-----------------------------------------}
      { Get the server's node name from the user. }
      {-----------------------------------------}
    prompt('Client: Enter the remote node name: ');
    readln( location );
```

```
    loclength := 0;
    while location[loclength+1] <> ' ' do
       loclength := loclength + 1;
    progname := 'RSERVER';
    prognamelength := 7;
      {-------------------------------------------------------------}
      { Set the dependent flag for the RPMCREATE.  This causes the  }
      { the server to terminate if the client terminates, or if the }
      { connection between them fails.                              }
      {-------------------------------------------------------------}
    for i := 0 to 31 do
       rpmflags[i] := false;
    rpmflags[dependent] := true;

      {--------------------------------------}
      { Create the server on the remote node. }
      {--------------------------------------}
RPMCREATE( progname,  prognamelength,
               location, loclength,
                 , , , ,
             rpmflags,  opt, progdesc, result );
    if result  <> 0 then
       ERROR( 'Couldn't create server', result );
      {-------------------------------------------------------------}
      { Once active, the server will create its own socket, look up }
      { the client's socket, and set up a vc connection between its }
      { socket and the client's socket.  Wait here for the connect  }
      { request from the server.                                    }
      {-------------------------------------------------------------}
IPCRECVCN( socketdesc, vcdesc, , , result );
    if result  <> 0 then
       ERROR( 'Connect receive failed', result );
      {-------------------------------------------------------------}
      { Now that the vc connection has been set up, the client's    }
      { socket name and socket can be deleted.                      }
      {-------------------------------------------------------------}
    IPCNAMERASE( clientsockname, socketnamelength, result );
    if result  <> 0 then
       ERROR( 'Couldn't delete socket name.', result );
     IPCSHUTDOWN( socketdesc, , , result );
    if result  <> 0 then
       ERROR( 'Couldn't shutdown socket.', result );
      {-------------------------------------------------------------}
      { Each pass of this loop gets a name, sends it to the server, }
      { and receives an accept/reject indication from the server.   }
      { If the server accepts the name, the client will receive the }
      { name information sent by the server.                        }
      {-------------------------------------------------------------}
    repeat
       prompt('Client: Enter name (or EOT to exit):');
       readln( name );
       if name <> 'EOT' then
          begin
```

```
         IPCSEND( vcdesc, name, namelength, , , result );
         if result  0 then
            ERROR( 'Send to server failed.', result );
         RECV( vcdesc, buf, 1, result );
         if result  <> 0 then
            ERROR( 'Receive from server failed.', result );
         if ord(buf[1]) = indaccept then
            begin
            RECV( vcdesc, buf, infolength, result );
            if result  <> 0 then
               ERROR( 'Receive from server failed.', result );
            for i := 1 to infolength do
               info[i] := buf[i];
            writeln('Client data is: ', info);
            end
         else{indicator = indreject}
            writeln('Client data could not be found.');
         end;
   until name = 'EOT';
     {-------------------------------------------------------------}
     { All names have been processed.  Terminate RSERVER and delete}
     { this end of the vc connection.  (RSERVER will automatically }
     { delete its end of the connection.)                          }
     {-------------------------------------------------------------}
   RPMKILL( progdesc, , , result );
   if result  <> 0 then
      ERROR( 'Couldn't kill server.', result );
   IPCSHUTDOWN( vcdesc, , , result );
   if result  <> 0 then
      ERROR( 'Couldn't shut down local vc.', result );
1:{error exit}
end.
```

RSERVER: Sample NetIPC/RPM Slave Program

```
$standard_level 'HP3000', uslinit$ program rserver( input, output );
{-------------------------------------------------------------------}
{                                                                   }
{ RSERVER: Sample NetIPC/RPM Slave Program                          }
{                                                                   }
{-------------------------------------------------------------------}
{                                                                   }
{ PURPOSE:                                                          }
{ The RCLIENT and RSERVER programs illustrate the use of the NetIPC }
{ and RPM services to implement a simple name server application.   }
{ See the RCLIENT program for details.                              }
{-------------------------------------------------------------------}
    label 1;                    {error exit             }
    const namelength      = 20; {all lengths in bytes   }
          infolength      = 60;
          maxnodelength   = 51;
          socketnamelength=  8; {returned by IPCNAME    }
          callsocket      =  3; {IPCCREATE socket type  }
          tcpprotocol     =  4; {IPCCREATE protocol type}
          indaccept       =  0; {accept indication      }
          indreject       =  1; {reject indication      }
    type  shortint        = -32768..32767;
          msgtype         = packed array[1..30] of char;
          nametype        = packed array[1..namelength] of char;
          infotype        = packed array[1..infolength] of char;
          buftype         = array [1..80] of char;
    var   clientsockname:   packed array[1..socketnamelength] of char;
          clientnode:       packed array[1..maxnodelength] of char;
          name:             nametype;
          info:             infotype;
          buf:              buftype;
          clientsocklength: integer;
          clientnodelength: integer;
          socketdesc:       integer;
          destdesc:         integer;
          vcdesc:           integer;
          result:           integer;
          i:                integer;
          found:            boolean;

procedure RPMGETSTRING;     intrinsic;  {RPM intrinsic    }
procedure IPCCREATE;        intrinsic;  {NetIPC intrinsics}
procedure IPCLOOKUP;        intrinsic;
procedure IPCCONNECT;       intrinsic;
procedure IPCRECV;          intrinsic;
procedure IPCSEND;          intrinsic;
procedure IPCSHUTDOWN;      intrinsic;
procedure ERROR( msg: msgtype; result: integer );
```

```
   {-----------------------------------------------------------------}
   { ERROR prints an error message and an associated NetIPC or RPM  }
   { result code.  It terminates the program by going to the error  }
   { exit.  Any NetIPC objects (sockets or virtual circuits) will   }
   { be deleted upon termination.                                   }
   {-----------------------------------------------------------------}

begin
    writeln( 'Server: ', msg, 'Result = ', result:3 );
    goto 1;
 end; {ERROR}

procedure RECV(      vcdesc: integer;
                var     buf: buftype;
                     length: integer;
                var  result: integer );

   {-----------------------------------------------------------------}
   { RECV receives a specified number of bytes from the virtual     }
   { circuit (vc) connection.  This compensates for the stream mode }
   { operation of NetIPC on the HP 3000, where an IPCRECV can return}
   { less than the requested number of bytes.  The loop in RECV     }
   { calls IPCRECV to receive the next chunk of data, until the     }
   { requested amount of data has been received.  Note that buf     }
   { must be unpacked to allow it to be indexed in the IPCRECV call.}
   {-----------------------------------------------------------------}
    var nextbufchar: integer;
        recvlength:  integer;
 begin
    result      := 0;
    nextbufchar := 1;
    while (length  <> 0) and (result = 0) do
       begin
       recvlength := length;
       IPCRECV( vcdesc, buf[nextbufchar], recvlength, , , result );
       nextbufchar := nextbufchar + recvlength;
       length      := length - recvlength;
       end;
 end; {RECV}

procedure FIND_NAME( var reqname: nametype;
                     var info:    infotype;
                     var found:   boolean );
   {-----------------------------------------------------------------}
   { FIND_NAME sequentially searches the data file for the requested}
   { name.  It returns an indication of whether the name was found, }
   { and if it was found, the information field for the name.  (In  }
   { a real name server, a more efficient look up method would be   }
   { used.)                                                         }
   {-----------------------------------------------------------------}

    var filename: packed array[1..9] of char;
        datafile: text;
        name:     nametype;
```

```
 begin
    filename := 'DATAFILE ';
    reset( datafile, filename );
    found := false;

while not found and not eof(datafile) do
       begin
       readln( datafile, name, info );
       if name = reqname then
          found := true       end;
 end; {FIND_NAME}

begin
      {-------------------------------------------------------------}
      { Retrieve the client's socket name and node name, passed as  }
      { RPM strings.                                                }
      {-------------------------------------------------------------}
    clientsocklength := socketnamelength;
    RPMGETSTRING( clientsockname, clientsocklength, result );
    if result  <> 0 then
       ERROR( 'Couldn't get socket name.', result );
    clientnodelength := maxnodelength;
    RPMGETSTRING( clientnode, clientnodelength, result );
    if result  <> 0 then
       ERROR( 'Couldn't get local nodename.', result );
      {-------------------------------------------------------------}
      { Create the server's socket, look up the client's socket,    }
      { and set up a vc connection between the server and the client}
      { sockets.                                                    }
      {-------------------------------------------------------------}
    IPCCREATE( callsocket, tcpprotocol, , , socketdesc, result );
    if result  <> 0 then
       ERROR( 'Couldn't create socket.', result );
    IPCLOOKUP( clientsockname, clientsocklength, clientnode,
clientnodelength, destdesc, , , result );
    if result  <> 0 then
       ERROR( 'Socket look up failed.', result );
    IPCCONNECT( socketdesc, destdesc, , , vcdesc, result );
    if result  <> 0 then
       ERROR( 'Socket connection failed', result );
      {-------------------------------------------------------------}
      { Wait for the connection acknowledgement from the client.    }
      {-------------------------------------------------------------}
    IPCRECV( vcdesc, , , , , result );
    if result  <> 0 then
       ERROR( 'Socket connect receive failed.', result );
      {-------------------------------------------------------------}
      { Once the connection is established, the socket and destina- }
      { tion descriptors are no longer needed.  So delete them.     }
      {-------------------------------------------------------------}
    IPCSHUTDOWN( socketdesc, , , result );
    if result  <> 0 then
       ERROR( 'Couldn't shut down socket.', result );
    IPCSHUTDOWN( destdesc, , , result );
```

```
    if result  <> 0 then
ERROR( 'Couldn't shut down dest.', result );

      {-------------------------------------------------------------}
      { Each pass of this loop receives one name from the client,   }
      { and looks up the name.  If the name is found, an accept     }
      { indication is sent back to the client, followed by the name }
      { information.  If the name is not found, a reject indication }
      { is returned to the client.  The server will remain in this  }
      { loop until it is terminated by the client.  On termination, }
      { the vc connection will automatically be shut down.          }
      {-------------------------------------------------------------}
    repeat
       RECV( vcdesc, buf, namelength, result );
       if result  <> 0 then
          ERROR( 'Receive from client failed.', result );
       for i := 1 to namelength do
          name[i] := buf[i];
       FIND_NAME( name, info, found );
       if found then
          begin
          buf[1] := chr(indaccept);
          IPCSEND( vcdesc, buf, 1, , , result );
          if result  <> 0 then
             ERROR( 'Send to client failed.', result );
          IPCSEND( vcdesc, info, infolength, , , result );
          if result  <> 0 then
             ERROR( 'Send to client failed.', result );
          end
       else{not found}
          begin
          buf[1] := chr(indreject);
          IPCSEND( vcdesc, buf, 1, , , result );
          if result  <> 0 then
             ERROR( 'Send to client failed', result );
          end;
    until false;
 1:{error exit}
 end.
```

E C Program Language Considerations

This appendix contains programming language differences that affect how NetIPC intrinsics are used in programs written in C programming language.

C Program Language Differences

Parameters

The *data* (*char^*) data in IPCSEND and IPCRECV in C programs to designate a long pointer.

Example

For example, in a Pascal program, the IPCSEND intrinsic can be written as:

```
ipcsend(vcdesc, data, 1, , , result)
```

In a C program the same intrinsic call would be written as:

```
ipcsend(vcdesc, (char^)data, 1, , , result)
```

Glossary

A

address A numerical identifier defined and used by a particular protocol and associated software to distinguish one node from another.

address key address resolution

address resolution In NS networks, the mapping of node names to IP addresses and the mapping of IP addresses to subnet addresses.

ASCII American National Standard Code for Information Interchange. A character set using 7-bit code used for information interchange among data processing and data communications systems. The American implementation of International Alphabet No. 5.

asynchronous Term used to describe a device's mode of operation whereby a sequence of operations are executed irrespective of time coincidence with any event. Devices that are directly accessible by people (for example, terminal keyboards) operate in this manner.

B

binary mode Data transfer scheme in which no special character processing is performed. All characters are considered to be data and are passed through with no control actions being taken.

bit Binary digit. A unit of information that designates one of two possible states, which are represented by either 1 or 0.

bps Bits per second. The number of bits passing a point per second.

buffer A logical grouping of a system's memory resources used by NS 3000/XL.

byte A sequence of eight consecutive bits operated on as a unit.

C

call In X.25, a call is an attempt to set up communication between two DTEs using a virtual circuit. Also known as a virtual call.

call collision A conflict that occurs at a DTE/DCE interface when there is a simultaneous attempt by the DTE and DCE to set up a call using the same logical channel identifier.

called address When a node sends out a call request packet, the packet contains the address of the destination node. The address of the destination node is the called address.

calling address When a node receives an incoming call packet, the packet contains the address of the sending node. The address of the sending node is the calling address.

CCITT Consultative Committee for International Telephony and Telegraphy. An international organization of communication carriers, especially government telephone monopolies, responsible for developing telecommunication standards by making recommendations. The emphasis is on "recommendations"; no carrier is required to adhere to a CCITT recommendation, although most do so in their own interests.

closed user group An X.25 user facility that allows communication to and from a pre-specified group of users and no one else.

compatibility mode Processing mode on HP 3000 Series 900 computers that allows applications written for MPE V/E-based systems to be ported and run without changes or recompilation.

computer network A group of computer systems connected in such a way that they can exchange information and share resources.

CUG *See* **closed user group.**

D

Datacommunications and Terminal Controller

Transmitted data that is sent faster than the equipment on the receiving end is capable of receiving it. The resulting overflow data is lost. *See also* **flow control.**

D bit Delivery confirmation bit. Used in the X.25 protocol, the setting of the D bit in DATA packets indicates whether delivery acknowledgment of the packet is required from the local DCE or from the remote DTE. It therefore allows the choice between local and end-to-end acknowledgment.

DCE Data circuit-terminating equipment. The interfacing equipment required in order to interface to data terminal equipment (DTE) and its transmission circuit. Synonyms: data communications equipment, dataset.

DTC Datacommunications and Terminal Controller. The DTC is a hardware device, configured as a node on a LAN, that enables asynchronous devices to access HP 3000 Series 900 computers. Terminals can either be directly connected to the DTC, or they can be remotely connected through a Packet Assembler Disassembler (PAD). The DTC can be configured with DTC/X.25 Network Access cards and DTC/X.25 Network Access software. A DTC/X.25 XL Network Link consists of two software modules: the X.25 XL System Access software (running on the host) and the DTC/X.25 Network Access software (running on the DTC).

DTC/X.25 Network Access The software that resides on the Datacommunications and Terminal Controller (DTC). To configure access to an X.25 network, you must configure two software components, the X.25 XL System Access (residing on the HP 3000 host and configured through use of NMMGR software) and the DTC/X.25 Network Access (configured on the OpenView Windows Workstation through use of the OpenView DTC Manager software).

DTC/X25 Network Access card This is the hardware card and channel adapter that provides X.25 Network Access. It resides in the Datacommunications and Terminal Controller (DTC).

DTC/X.25 XL Network Link

Software and hardware that provides MPE XL access to private and public X.25 networks. The X.25 XL System Access software resides on an HP 3000 host and is configured through use of NMMGR. The DTC/X.25 Network Access software resides on the Datacommunications and Terminal Controller and is configured at the OpenView Windows Workstation.

DTE Data terminal equipment. Equipment that converts user information into data-transmission signals or reconverts received data signals into user information. Data terminal equipment operates in conjunction with data circuit-terminating equipment.

DTS Distributed Terminal Subsystem. This consists of all the Datacommunications and Terminal Controllers (DTCs) on a LAN, their LANIC cards (attached to the host), the LAN cable, and the host and DTC software that controls all related DTS hardware.

E

environment A session that is established on a remote node.

Ethernet A Local Area Network system that uses baseband transmission at 10 Mbps over coaxial cable. Ethernet is a trademark of Xerox Corporation.

extended packet sequence numbering One of the optional Network Subscribed Facilities that provides packet sequence numbering using modulo 128. If not subscribed, modulo 8 is used.

F

facility An optional service offered by a packet switching network's administration and requested by the user either at the time of subscription for network access or at the time a call is made. Also known as user facility.

facility set A facility set defines the various X.25 connection parameters and X.25 facilities that can be negotiated for each virtual circuit on a per-call basis.

fast select An optional packet-switching network facility by which user data may be transmitted as part of the control packets that establish and clear a virtual connection.

FCS Frame Check Sequence. A sequence of bits generated by X.25 at Level 2 that forms part of the frame and guarantees the integrity of its frame's content. The FCS is also used by the IEEE802.3 protocol to check the validity of frames.

file equation Assignment statement used to associate a file with a specific device or type of device during execution of a program.

file number Unique number associated with a file when the file is opened. The file number is returned in the FOPEN or HPFOPEN call used to open the file. It can be used to access that file until the file is closed.

file specification The name and location of a file. The full specification for a file includes the file name, group, and account.

file system The part of the operating system that handles access to input/output devices (including those connected through the DTC), data blocking, buffering, data transfers, and deblocking.

flow control A means of regulating the rate at which data transfer takes place between devices to protect against data overruns.

flow control negotiation One of the network subscribed facilities, selected at subscription time; this facility allows the Flow Control parameter to be negotiated at call set-up time, as opposed to having a predefined value.

formal file designator Name that can be used programmatically or in a file equation to refer to a file.

FOS Fundamental Operating System. The programs, utilities, and subsystems supplied on the Master Installation Tape that form the basic core of the MPE XL operating system.

G

Guided Configuration A method of configuring a node wherein a subset of the complete NMMGR interface is presented and defaults of configurable values are used automatically.

H

host-based network management Method of managing asynchronous communications for HP 3000 Series 900 computers. All of the control software is configured on a single MPE XL host and is downloaded to the DTCs that are managed by that host. With host-based management, there is a permanent relationship between each DTC and the host, and terminal users can access only the single MPE XL system that owns the DTC their terminal is connected to.

host computer The primary or controlling computer on a network. The computer on which the network control software resides. For HP purposes, it may also be used to distinguish the MPE XL system (host) from the DTC.

I

idle device timeout Timeout defined by the Configure:CPU command. ; When the timer lapses, a device connected to the DTC user interface that is still inactive will be disconnected.

IEEE 802.3 A standard for a broadcast local area network published by the Institute for Electrical and Electronics Engineers (IEEE). This standard is used for both the ThinLAN and ThickLAN implementations of the LAN.

INP Intelligent Network Processor. The card residing in the back of an MPE V-based node that provides a point-to-point or X.25 interface.

internet communication

Communication that occurs between networks.

Internet Protocol A protocol used to provide routing between different local networks in an internetwork, as well as among nodes in the same local network. The Internet Protocol corresponds to Layer 3, the Network Layer, of the OSI model. *See also* **IP address.**

internetwork Two or more networks joined by gateways.

intranet communication

Communication that occurs between nodes in a single network.

intrinsic System routine accessible by user programs which provides an interface to operating system resources and functions. Intrinsics perform common tasks such as file access and device control.

IP *See* **Internet Protocol.**

IP address Internet Protocol address. An address used by the Internet Protocol to perform internet routing. A complete IP address comprises a network portion and a node portion. The

network portion of the IP address identifies a network, and the node portion identifies a node within the network.

ISO International Organization of Standards. An international federation of national standards organizations involved in developing international standards, including communication standards.

L

LANIC *See* **Local Area Network Interface Controller**.

LANIC Self-Test A ROM-based program on a LANIC card that tests and reports the status of the LANIC hardware.

LAP-B Link Access Protocol - Balanced. The data link protocol specified by the 1980 version of X.25 at Level 2 that determines the frame exchange procedures. LAP-B must also be used over direct-connect NS Point-to-Point 3000/XL Links.

ldev *See* **logical device number**.

link name The name that represents a hardware interface card. The link name can contain as many as eight characters. All characters except the first can be alphanumeric; the first character must be alphabetic.

Local Area Network Interface Controller (LANIC) A hardware card that fits into the backplane of the HP 3000 Series 900 computer and provides a physical layer interface for IEEE 802.3 local area networks.

local connection *See* **direct connection**.

local node The computer that you are configuring or that you are logged on to.

logging The process of recording the usage of network resources. Events can be logged to both the OpenView workstation and to the MPE XL host.

logging class A number defining the severity of any given event logged. An operator uses the logging classes to specify which events are to be logged. Class 1 (catastrophic event) is always logged.

logical device number (ldev)

A value by which MPE XL recognizes a specific device.

loopback The routing of messages from a node back to itself.

LUG Local User Group. A list defined for a particular DTC and card that specifies which remote nodes this DTC can send data to and also which remote nodes this DTC can receive data from. (*See also* **Closed User Group**).

M

M bit More data bit. Setting this bit in a DATA packet indicates that at least one more DATA packet is required to complete a message of contiguous data.

modulo Value used as the counting cycle for determining the send sequence number (N(S)) of frames sent across an X.25 network.

MPE XL MultiProgramming Executive XL. The operating system of the HP 3000 Series 900 computers. The NS3000/XL network services operate in conjunction with the MPE XL operating system.

multiplexer MUX. A device that allows multiple communication links to use a single channel.

N

native mode The run-time environment of MPE XL. In Native Mode, source code has been compiled into the native instruction set of the HP 3000 Series 900 computer.

NetIPC Network Interprocess Communication. Software that enables programs to access network transport protocols.

network A group of computers connected so that they can exchange information and share resources.

network address This can be either 1) the network portion of an IP address as opposed to the node portion, or 2) when referring to X.25 networks, it is a node's X.25 address.

network boundary The logical division between networks in an internetwork.

network directory A file containing information required for one node to communicate with other nodes in 1) an internetwork, 2) an X.25 network, or 3) a network that contains non-HP nodes. The active network directory on a node must be named NSDIR.NET.SYS.

network interface (NI). The collective software that enables data communication between a system and a network. A node possesses one or more network interfaces for each of the networks to which it belongs. Network interface types are LAN802.3, router (point-to-point), X.25, loopback, and gateway half. The maximum number of supported NIs is 12, one of which is reserved for loopback.

network management The collective tasks required to design, install, configure, maintain, and if necessary, change a network.

Network Services NS. Software application products that can be used to access data, initiate processes, and exchange

information among nodes in the network. The HP 3000/XL Network Services include RPM, VT, RFA, RDBA, and NFT.

network subscribed facilities

A set of parameters that the user chooses when he subscribes to the X.25 network; they include Flow Control Negotiation, Use of D-bit, Throughput Class Negotiation and Extended Packet Sequence Numbering.

NFT Network File Transfer. The network service that transfers disc files between nodes on a network.

NI *See* **network interface**.

NMCONFIG.PUB.SYS A file that contains all the network configuration data for the HP 3000 Series 900 computer on which it resides. It includes information about the DTCs that can access the system as well as information about any Network Service (NS) products running on the system. This is the only file name allowed.

NMDUMP A utility used to format log and trace files.

NMMAINT A utility that lists the software module version numbers for all HP AdvanceNet products, including NS 3000/XL. It detects missing or invalid software modules.

NMMGR ode Management Services Configuration Manager. A software subsystem that enables you to configure DTC connectivity and network access parameters for an HP 3000 Series 900 computer.

NMMGRVER A conversion program called NMMGRVER.PUB.SYS. It converts configuration files created with NMMGR from an earlier version to the latest format.

NMSAMP1.PUB.SYS Sample configuration file supplied with FOS that can be used as a template for DTS configuration.

node A computer that is part of a network. The DTC is also considered to be a node and has its own address.

node address The node portion of an IP address, which consists of a node portion and a network portion.

Node Management Services Configuration Manager *See* **NMMGR**.

node name A character string that uniquely identifies each system in a network or internetwork. Each node name in a network or internetwork must be unique; however, a single node can be identified by more than one node name.

NS 3000/XL A Hewlett-Packard data communication product that provides networking capabilities for MPE XL based HP 3000 minicomputers. NS 3000/XL consists of a link and network services.

NS 3000/XL Link Software and hardware that provides the connection between nodes on a network. Some of the NS 3000/XL links available are the ThinLAN 3000/XL Link and its ThickLAN option, the DTC/X.25 XL Network Link, the NS Point-to-Point 3000/XL Link, and the StarLAN 10 3000/XL link.

NS 3000/XL Network Services

Software applications that can be used to access data, initiate processes, and exchange information among nodes in a network. The services are RPM, VT, RFA, RDBA, and NFT.

NSDIR.NET.SYS Name of the active network directory file. *See also* **network directory**.

O

octet An eight-bit byte operated upon as an entity.

OSI model Open Systems Interconnection model. A model of network architecture devised by the International Standards Organization (ISO). The OSI model defines seven layers of a network architecture, with each layer performing specified functions.

P

packet A block of data whose maximum length is fixed. The unit of information exchanged by X.25 at Level 3. There are DATA packets and various control packets. A packet type is identified by the encoding of its header.

Packet Exchange Protocol

PXP. A transport layer protocol used in NS3000/XL links to initially establish communication between nodes when NetIPC socket registry is used.

packet-switched network name The name of a data communication network adhering to the CCITT X.25 recommendation. This can be a PDN or a private network, such as the HP PPN.

PAD (packet assembler/disassembler) A device that converts asynchronous character streams into packets that can be transmitted over a packet switching network (PSN).

PAD name A name of up to eight characters that is associated with a configured PAD device. The PAD name is known to both the DTC (defined by the DTC Manager) and the MPE XL systems (defined by NMMGR) that the device can access.

PAD profile Terminal or printer profile that specifies the configuration characteristics for PAD-connected devices.

PDN Public data network. A data communication network whose services are available to any user willing to pay for them. Most PDNs use packet switching techniques.

port An outlet through which a device can be connected to a computer, consisting of a physical connection point and controlling hardware, controlling software, and configurable port characteristics. Ports can be thought of as data paths through which a device communicates with the computer.

Precision Architecture The hardware design structure for the HP 3000 Series 900 computer family.

privileged mode A capability assigned to accounts, groups, or users allowing unrestricted memory access, access to privileged CPU instructions, and the ability to call privileged procedures.

probe protocol An HP protocol used by NS 3000/XL IEEE 802.3 networks to obtain information about other nodes on the network.

probe proxy server A node on an IEEE 802.3 network that possesses a network directory. A probe proxy server can provide a node with information about other nodes on the same or other networks of an internetwork.

profile A method of grouping device connection specifications and characteristics so that the set of characteristics can be easily associated with groups of like devices. *See also* **printer profile, terminal profile**.

programmatic device A device operating under control of a program running on a computer. Programmatic devices can be used for input, output, or both, depending on the device and how it is opened by the controlling program.

protocol A set of rules that enables two or more data processing entities to exchange information. In networks, protocols are the rules and conventions that govern each layer of network architecture. They define what functions are to be performed and how messages are to be exchanged.

PSN Packet-Switching Network. Any data communication network using packet-switching techniques wherein data is disassembled into packets at a source interface and reassembled into a data stream at a destination interface. A public PSN offers the service to any paying customer.

PVC Permanent Virtual Circuit. A permanent logical association between two physically separate DTEs that does not require call set-up or clearing procedures.

PXP *See* **Packet Exchange Protocol**.

Q

Q bit Qualified bit. When set in DATA packets the Q bit signifies that the packet's user data is a control signal for the remote device, not a message for its user.

QuickVal A software program that tests whether Network Services are operating correctly between nodes.

R

RDBA Remote Data Base Access. A network service that allows users to access data bases on remote nodes.

remote node A node on an internetwork other than the node you are currently using or referring to.

retransmission count (N2) The maximum number of times a frame will be retransmitted following the expiration of the Retransmission Timer, T1.

retransmission timer (T1)

Length of time the transmitter will wait for an acknowledgment from the destination address before attempting to retransmit the frame. When choosing this value, factors like the line speed and maximum frame size should be taken into account.

RFA Remote File Access. A network service that allows users to access file and devices on remote nodes.

routing Routing refers to the process used to determine the path that packets, or fragments of a message, take through a network to reach a destination node.

RPM Remote Process Management. A network service that allows a process to programmatically initiate and terminate other processes throughout a network from any node on the network.

S

security string An alphanumeric character string that functions as a password for dial links. The security string is used by the Dial IP protocol.

SVC Switched Virtual Circuit. Path through an X.25 network that is established at call set-up time.

synchronous A mode of operation or transmission whereby a continuous data stream is generated without intervals between characters. The data stream is synchronized by clock signals at the receiver and

transmitter. As a result, fast transmission speeds (above 9600 bps) are attainable.

system configuration The way you tell MPE XL what peripheral I/O devices are attached to the DTC and what parameters are required for system operation.

T

TCP *See* **Transmission Control Protocol**

ThinLAN 3000/XL A LAN that conforms to the IEEE 802.3 Type 10 BASE 2 standard LAN.

throughput class A value assigned to a given virtual circuit that defines how many network resources should be assigned to a given call. It is determined by the access line speed, packet and window sizes, and the local network's internal mechanisms.

throughput class negotiation

One of the Network Subscribed Facilities defined at subscription time. This allows the user to negotiate the Throughput Class at call set-up time.

timer (T3) Length of time that a link can remain in an idle state. After the expiration of the timer, the link is considered to be in a non-active, non-operational state and is automatically reset. The value should be chosen carefully. In particular, it must be sufficiently greater than the Retransmission Timer (T1) so that there is no doubt about the link's state.

topology The physical arrangement of nodes in a network. Some common topologies are bus, star, and ring.

Transmission Control Protocol TCP. A network protocol that establishes and maintains connections between nodes. TCP regulates the flow of data, breaks messages into smaller fragments if necessary (and reassembles the fragments at the destination), detects errors, and retransmits messages if errors have been detected.

transparent mode Data transfer scheme in which only a limited number of special characters retain their meaning and are acted on by the system. All other characters are considered to be data and are passed through with no control actions being taken.

transport, network Software that corresponds to layers 4 and 3 of the OSI network architecture model. The function of this software is to send data out over the appropriate communications link, to receive incoming data, and to route incoming or outgoing data to the appropriate destination node.

U

unacknowledged frame number (K) The number of frames that can be transmitted without receiving an acknowledgment from the destination address. When this number (K) frame is reached, the same K frames are retransmitted.

unedited mode *See* **transparent mode**.

V

V.24 The CCITT recommendation that defines the function of the interchange circuits between a DTE and a DCE.

validation The process of ascertaining whether the network transport configuration file has been correctly configured. This is accomplished by using the NMMGR Validate Configuration File screen.

VAN Value-Added Network. A data communication network that uses and pays for facilities belonging to another carrier. The value-added package is then sold to a user.

VC *See* **virtual circuit**.

virtual circuit A logical association between two physically separate DTEs.

Virtual Terminal A network service that allows a user to establish interactive sessions on a node.

VPLUS Software used to generate screens such as those displayed by NMMGR.

V-Series (V.##) CCITT A set of CCITT recommendations related to data communication over a voice-grade telephone network.

VT *See* **Virtual Termina**l.

W

Workstation Configurator A utility available on MPE XL systems that allows users to create customized terminal and printer types by entering data through a series of VPLUS screens.

X

X.3 Defines the user facilities that should be internationally available from the packet assembler/disassembler (PAD) facility when this is offered by a public data network.

X.21 Defines the physical interface between a DTE and a DCE of a public data network where the access to the network is made over synchronous digital lines.

X.25 Defines the interface between a DTE and a DCE for packet mode operation on a Public Data Network (PDN).

X.25 address The X.25 address provided by the network administration if you are connected to a Public Data Network (PDN).

X.25 address key An X.25 address key is a label that maps a node's IP address to its X.25 address and its associated X.25 parameters. You have a combined maximum of 1024 X.25 address keys in the SVC and PVC path tables.

X.25 LUG address X.25 address of a node belonging to a LUG.

X.25 XL System Access he software that works in conjunction with the DTC/X.25 Network Access software to provide MPE XL access to X.25. The software resides on an HP 3000 host and is configured through use of NMMGR. To configure access to an X.25 network, you must configure two software components, the X.25 XL System Access and the DTC/X.25 Network Access (residing on the Datacommunications and Terminal Controller and configured at the OpenView Windows Workstation). Together, these two components provide a network connection on HP 3000 systems to private and public X.25 packet-switched networks PSNs).

X.29 Defines the interface for data exchange between a packet-mode DTE and a remote Packet Assembly/Disassembly (PAD) facility over a packet switching network.

X.Series (X.##) CCITT recommendations A set of recommendations for data communication networks governing their services, facilities, and the operation of terminal equipment and interfaces.

Index