HP 9000 Networking

NetIPC Programmer's Guide

Edition 2
E0792

98194-60532
Printed in U.S.A. 0792

Preface

Network Interprocess Communication (NetIPC) is a programmatic service provided by the HP 9000 Networking product. The *NetIPC Programmer's Guide* is the primary reference manual for programmers who write or maintain NetIPC applications on HP 9000 computers. This manual should also be read by Node Managers before designing an HP 9000 network, so that they have a clear understanding of the features provided by NetIPC.

This manual is organized as follows:

Chapter 1 "NetIPC Concepts," explains how NetIPC establishes and terminates connections between processes to exchange data. This chapter also introduces the NetIPC calls that perform these tasks.

Chapter 2 "Cross-System NetIPC," describes special programming considerations you should be aware of when writing an HP 9000 NetIPC program that will communicate with a peer NetIPC process at an HP 1000 A-Series, HP 3000, or HP 3000 Series 900.

Chapter 3 "NetIPC Calls," provides a detailed description of each NetIPC call in alphabetical order. This chapter also explains the structure and function of several parameters that are common to multiple NetIPC calls.

Appendix A "Sample NetIPC Programs," presents NetIPC sample programs in C and FORTRAN.

Appendix B "Error Messages," lists and describes the error messages that can be produced by NetIPC.

Appendix C "System Calls and NetIPC Sockets," lists and describes the HP-UX system calls that operate on NS sockets.

Appendix D "LAN/9000 Series 600/800 Migration," compares the NS/9000 Series 600/800 and HP 9000 Series 600/800 products to the NS/1000 and NS/9000 products.

Appendix E "Porting NetIPC Programs," summarizes the differences and provides information to help you port NetIPC programs between an HP 1000 and an HP 9000 Series 600/800 computer.

Documentation Map

The following documentation map lists the manuals containing information related to the product described in this manual. You may need information from one or all of these manuals.

NS/1000 User/Programmer Reference Manual

NetIPC 3000/V Programmer's Reference Manual

NetIPC 3000/XL Programmer's Reference Manual

HP 9000 Using Network Services

HP-UX Reference Manual

PORT/HP-UX Migration Analysis Utility Manual

HP FORTRAN 77/HP-UX Reference Manual

HP C Reference Manual

HP C/HP-UX Reference Manual Supplement

HP Pascal Reference Manual

FORTRAN 77 Reference Manual

Pascal/1000 Reference Manual

HP FORTRAN 77/HP-UX Migration Guide

HP Pascal/HP-UX Migration Guide

Contents

Chapter 3 NetIPC Calls

Appendix E Porting NetIPC Programs

Figures

Tables

NetIPC Concepts

Note The information contained in this manual applies to both the Series 300/400 and Series 600/700/800 HP 9000 computer systems. Any differences in installation, configuration, or operation are specifically noted.

Network Interprocess Communication (NetIPC) is a service that enables processes on the same or different nodes to communicate using a series of programmatic calls. Processes that use NetIPC calls gain access to the communication services provided by the network protocols utilized by the HP 9000 networking products. NetIPC does not encompass a protocol of its own, but acts as a generic interface to these protocols.

A NetIPC process running on an HP 9000 computer can communicate with a peer process at:

- Another HP 9000 computer (Series 600/700/800 or 300/400).
- An HP 1000 A-Series computer.
- An HP 3000 computer (MPE-V or Series 900).
- A PC on an HP OfficeShare Network.

NetIPC communication between processes running on computers of different types (between an HP 9000 and an HP 3000, for example) is referred to as cross-system NetIPC.

Note NetIPC communication between an HP 9000 Series 600/700/800 and HP 9000 Series 300/400 is not considered cross-system NetIPC because both systems are HP 9000s.

HP 9000 NetIPC for the Series 300/400 and Series 600/700/800 is not compatible with Berkeley IPC (also known as "Berkeley Sockets" or "BSD IPC") or the interprocess communication service that is part of the NS/9000 Series 500 product.

The "Cross-System NetIPC" chapter describes special programming considerations you should be aware of when writing an HP 9000 NetIPC program that will communicate with a peer NetIPC process at a different type of computer system. For information about writing a NetIPC program to run on an HP 1000 A-Series, PC, HP 3000, or HP 3000 Series 900, you must refer to the following manuals:

- *NS/1000 User/Programmer Reference Manual.*
- *NetIPC 3000/V Programmer's Reference Manual.*
- *NetIPC 3000/XL Programmer's Reference Manual.*
- *PC NetIPC/RPM Programmer's Reference Guide.*

Chapter Overview

The information presented in this chapter is organized into the following major sections:

- **Sockets.** Describes the fundamental building block of interprocess communication, the socket.
- **Connections.** Defines key terms used to describe NetIPC connections.
- **Establishing a VC Connection.** Explains how to use NetIPC calls to establish a virtual circuit (VC) connection.
- **Connection Establishment Summary.** Describes the sequences of NetIPC calls used to establish a virtual circuit connection.
- **Sending and Receiving Data.** Describes the different modes of data exchange provided by NetIPC and explains how to use NetIPC calls to send and receive data.
- **Shutting Down a Connection.** Explains how to use NetIPC calls to close a virtual circuit connection.
- **Summary of NetIPC Calls.** Presents a brief description of each of the HP 9000 NetIPC calls.

Sockets

NetIPC processes communicate with each other by means of **sockets**. A socket is an endpoint through which connections can be established, and data can be sent and received. Processes communicate through sockets via NetIPC calls. The Transport Layer's Transmission Control Protocol (TCP) regulates the transmission of data to and from sockets. Although data must pass through the control of lower-level protocols and, if necessary, through intervening nodes, these details are transparent to NetIPC processes when they send and receive data.

Connections

Before two processes can communicate, one side (the passive side or the **server**) must create a call socket by calling `ipccreate`. The process which creates the call socket may name the socket by calling `ipcname`. This allows the other side (the active side or **client**) to obtain address information regarding the server by calling `ipclookup`. Alternatively, the client may obtain address information regarding the server by calling `ipcdest`.

The routines `ipcname`, `ipclookup`, and `ipcnamerase` allow sockets to be referred to by ASCII names rather than protocol addresses. When `ipcname` is called, the ASCII name and information identifying the call socket being named are recorded in a table. When `ipclookup` is called, the nodename is examined first. If the nodename parameter specifies the local node, then the name table on that local node is searched for the specified socket name. If the nodename refers to a remote node, then

1. the address of that remote node is determined,
2. a request is sent to that node,
3. the name table on that remote is searched, and
4. the result of that search is returned in a reply message to the local node indicating `ipclookup` is complete.

An alternative to `ipclookup` is `ipcdest` which allows you to specify a protocol address, also known as a port, rather than a socket name. The network address of the node specified by the nodename is obtained and stored along with the protocol address, and `ipcdest` is complete.

Both `ipclookup` and `ipcdest` return a destination descriptor. A **destination descriptor** is an integer which indexes a data structure just as a file descriptor is an integer which indexes a file. A destination descriptor contains address information which identifies a node on the network and a call socket at that node. The information in a destination descriptor is the same address information passed to the BSD IPC networking routine "connect."

Once a client process has obtained a destination descriptor, it may initiate a virtual circuit connection by calling `ipcconnect`. A **virtual circuit** is a connection using a reliable transport protocol, in this case TCP, which guarantees that data are not corrupted, lost, duplicated, or received out of order.

Descriptors

NetIPC processes acting as clients reference **destination descriptors** and **virtual circuit socket descriptors.** NetIPC processes acting as servers reference virtual circuit socket descriptors and **call socket descriptors.** A single process can act as both a client and a server.

- **Call Socket Descriptor.** A call socket descriptor references a data structure created by calling `ipccreate` which allows server processes to create virtual circuit connections. The NetIPC routine `ipccreate` is equivalent to the BSD networking routines "socket," "bind," and "listen."
- **Destination Descriptor.** A destination descriptor references a data structure that contains address information about a destination call socket. A destination descriptor must be obtained before a process can connect to the destination call socket. A process obtains a destination descriptor by invoking `ipclookup()` or `ipcdest()`.
- **VC Socket Descriptor.** A VC socket descriptor refers to a VC socket. A VC socket is the endpoint of a virtual circuit connection betwen two processes. VC socket descriptors are returned by `ipcrecvn()` and `ipcconnect()`.

Socket descriptors are allocated from the same space as file descriptors. A process may have a maximum of 1024 socket and file descriptors. Therefore, sockets are accessible through the standard HP-UX file system calls such as `read()`, `write()`, `ioctl()`, `fcntl()`, `select()`, `stat()`, `dup()`, `writev()` and `readv()`. For more information on using these calls with NetIPC sockets, refer to Appendix C of this manual.

Table 1-1. Descriptor Type and Definitions

Descriptor Type	Parameter Name	Description	Returned as Output From
call socket descriptor	*calldesc*	Refers to a call socket. A call socket is used by server processes to build a VC socket.	`ipccreate()`
destination descriptor	*destdesc*	Refers to descriptor referencing address information 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()`

Socket Ownership

When a NetIPC process creates a call socket by calling `ipccreate()`, or creates a VC socket by calling `ipcconnect()` or `ipcrecvcn()`, it is said to **own** the socket.

A process can also become an owner of a socket by inheriting a socket descriptor. NetIPC descriptors (call socket, VC socket, and destination), like file descriptors, are copied to the child process when a process forks. As a result, more than one process can have a descriptor for the same socket. Any process that has a descriptor is considered to be an owner of that descriptor. As a programmer, you are responsible for regulating the use of shared descriptors.

A process may have access to a maximum of 1024 descriptors at one time. This limit includes file descriptors as well as socket descriptors.

Establishing a VC Connection

Establishing a connection between two processes requires that one process create a call socket which the other process can connect to. The process which creates the call socket is often referred to as the passive side or the server. The process which initiates the connection is often referred to as the active side or client. The typical use of the client-server model involves a server process which creates a call socket, receives a connection, and forks a child to handle that connection while the server listens for another connection.

As a programmer, you are responsible for synchronizing your NetIPC programs so that the NetIPC calls are executed in the manner illustrated by the following drawings and text.

Although only two processes are shown in this example, this is not meant to imply that communication cannot exist between more than two processes. 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.

NetIPC does not provide a call to schedule a remote process. Remote HP 9000 processes must be manually started or can be scheduled by user-written daemons. You can start the daemon at system start up by invoking the daemon from the `/etc/netlinkrc` file.

For information about scheduling remote programs on other HP computers, refer to the "Cross-System NetIPC" chapter.

1. Creating a Call Socket

Before communication can begin, the server process must create a call socket by calling `ipccreate`. The `ipccreate` routine creates a call socket and returns a call socket descriptor in its *`calldesc`* parameter. The call socket descriptor is used in subsequent NetIPC calls.

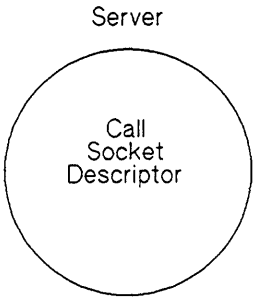

Figure 1-1. ipccreate() (Server)

2. Naming a Call Socket

After the server creates the socket, it may optionally name that socket. Naming the socket allows client processes to make a connection if they know the socket name, but not the protocol specific address. Alternatively, the server could create a call socket at a specific protocol address. In that case the client process would need to know the protocol address instead of the socket name. Socket names are considered an advantage over protocol addresses because when a server names a socket, that socket is guaranteed to get a unique protocol address. Several users or programs can operate using named sockets without danger of accidentally using a common protocol address.

The server process names a call socket by calling `ipcname`. The socket name is then recorded in the local socket registry name table. Remember that the server process must name the socket before the client process calls `ipclookup`.

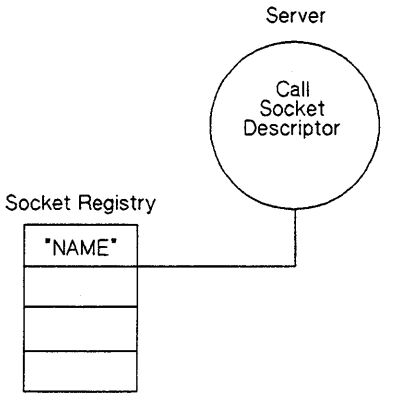

Figure 1-2. ipcname() (Server)

3. Finding A Call Socket Name

The client process must get address information regarding the server process by calling either `ipclookup` or `ipcdest`. If the server process named the call socket, then the client process must call `ipclookup`. If the server process created the call socket at a specific protocol address, then the client process must call `ipcdest`.

Both `ipclookup` and `ipcdest` return a destination descriptor to the user. The destination descriptor identifies a data structure which contains address information about the server's call socket.

It may be difficult to ensure that a socket name is placed in the socket registry prior to being "looked up" by another process. Several ways to avoid this timing problem are outlined in the discussion of `ipclookup()` in the "NetIPC Calls" chapter.

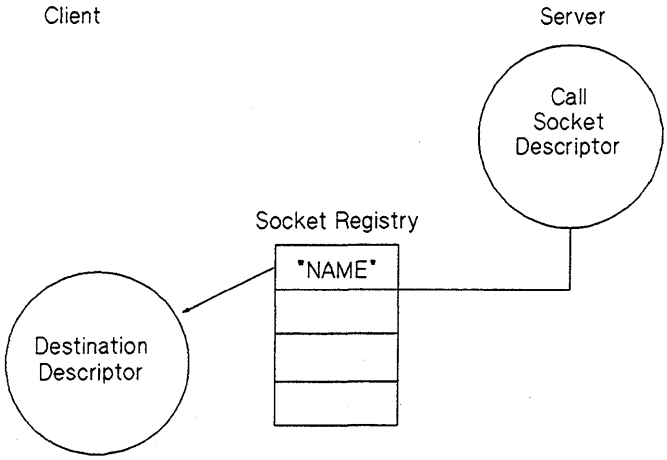

Figure 1-3. ipclookup() (Client)

4. Requesting a Connection

The client process specifies the destination descriptor returned by `ipclookup` or `ipcdest` when it calls `ipcconnect`. The routine `ipcconnect` will create a virtual circuit socket and initiate, but not complete, a connect. The virtual circuit socket (vc socket) is returned to the user in the *vcdesc* parameter. The vc socket may not be used to send or receive data until the connection has been completed. The client process must call `ipcrecv` to determine when the connection is complete.

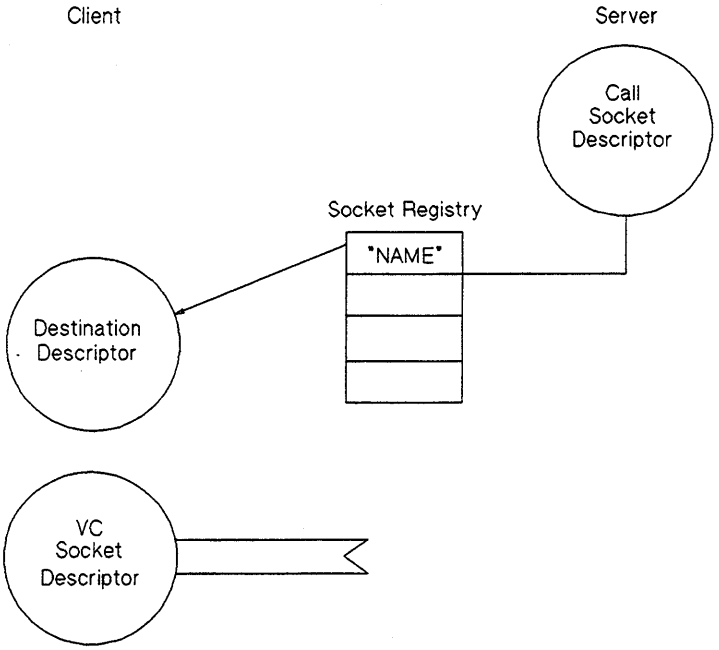

Figure 1-4. ipcconnect() (Client)

5. Receiving a Connection

The server process receives a connection by calling `ipcrecvcn`. The routine `ipcrecvcn` references the call socket descriptor and returns a virtual circuit socket descriptor to the user. The vc socket descriptor can be used to send and receive data. Note that the connection is automatically accepted on the server's behalf when the connection is initially requested by the client process. The client process can determine that the connection is "established" before the server calls `ipcrecvcn`. Any data which the client sends before the server calls `ipcrecvcn` is queued. If the client expects data from the server, it may timeout waiting for data even though the server has not done an `ipcrecvcn`. After the server calls `ipcrecvcn`, it can call `ipcrecv` to read data sent by the client.

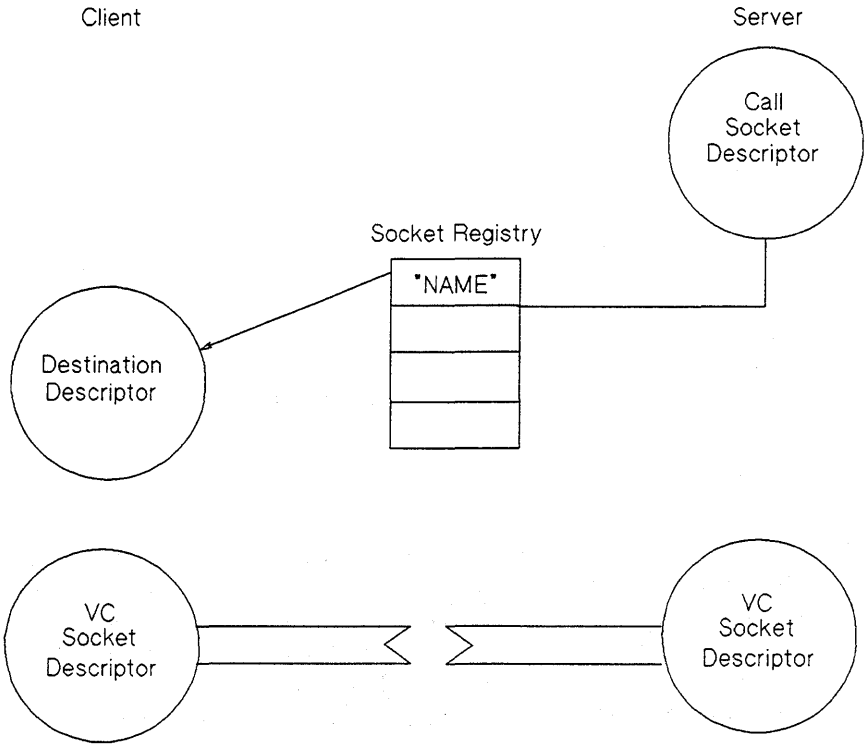

Figure 1-5. ipcrecvcn() (Server)

6. Checking the Status of a Connection

After the client calls `ipcconnect`, it must call `ipcrecv` to determine when the connection is completed.

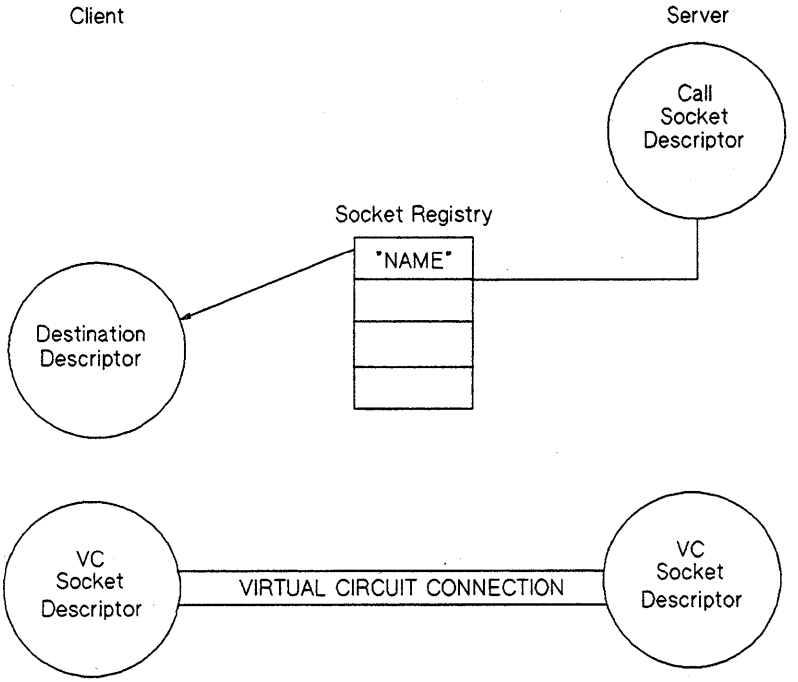

Figure 1-6. ipcrecv() (Client)

Note When the client considers the connection established, it may be different from when the server considers the connection established as described above. Once the connection is established, data transfer can begin using the vc socket descriptors and the `ipcsend` and `ipcrecv` commands. Refer to the "NetIPC Calls" chapter for a detailed description of `ipcsend` and `ipcrecv`.

Connection Establishment Summary

Figures 1-7 and 1-8 illustrate the two alternate sequences of NetIPC calls that are used to establish a virtual circuit connection. Figure 1-7 summarizes the information presented in Figures 1-2 through 1-6.

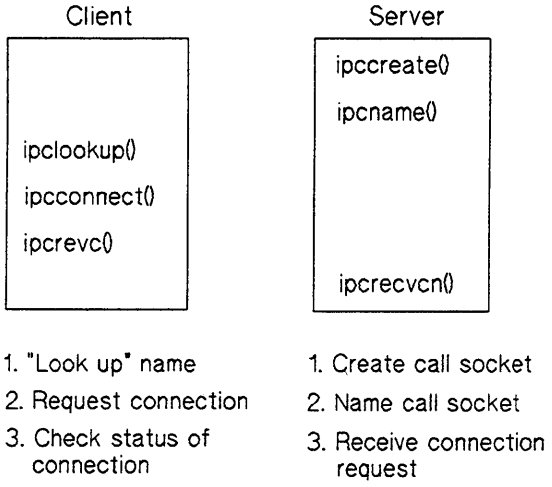

Figure 1-7. Establishing Connection with ipclookup Call

Figure 1-8 summarizes a different way to establish a virtual circuit connection using `ipcdest`.

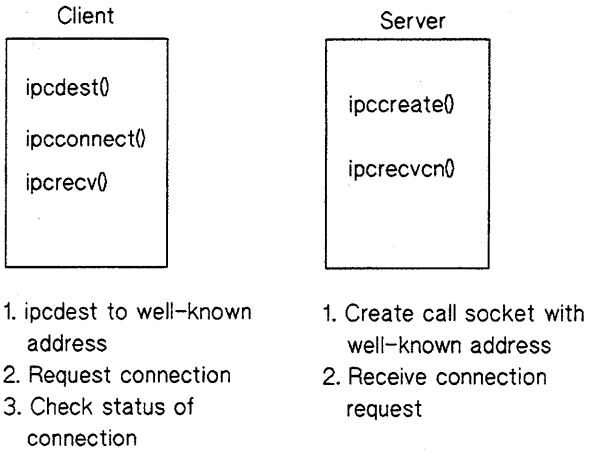

Figure 1-8. Establishing Connection with ipcdest Call

In both figures on the client side (that is, at the client's node), steps 2 and 3 are the same. `ipclookup` and `ipcname` reference a destination call socket by name whereas `ipcdest` references the destination call socket by its well-known address. Note that the advantage of using `ipclookup` is that names might be easier to remember and use. With `ipcdest`, the address must be unique and other processes must cooperate and not use that same address.

Sending and Receiving Data

Once a virtual circuit connection is established, processes can send and receive data using the NetIPC calls `ipcsend()` and `ipcrecv()`. `ipcsend()` is used to send data on an established connection. `ipcrecv()` is used to receive data on an established connection. (Note that `ipcrecv()` has a dual function: to establish a virtual circuit connection and to receive data on a previously established connection.)

Stream Mode

All data transfers between NetIPC processes are in **stream mode**. Stream mode adheres to the Transport Layer's Transmission Control Protocol (TCP). In stream mode, data is transmitted in a stream of bytes; there are no end-of-message markers. This means that the data received by an individual `ipcrecv()` request may not be equivalent to a message sent by an individual `ipcsend()` call. In fact, the data received may contain part of a message or multiple messages sent by multiple `ipcsend()` calls. Although no attempt is made to preserve boundaries between data sent at different times, the data received will always be in the correct order (in the order that the messages were sent).

You may specify the maximum number of bytes that you are willing to receive through a parameter of the `ipcrecv()` call. When the call completes, this parameter will contain the number of bytes *actually* received. The amount of data received will never be more than the amount that was requested, but it may be less. Whether or not an `ipcrecv()` call will receive less data than it requested is determined by the `NSF_DATA_WAIT` bit of the *`flags`* parameter. If the `NSF_DATA_WAIT` bit is set, `ipcrecv()` will never receive less than the requested amount; if the `NSF_DATA_WAIT` bit is *not* set, `ipcrecv()` may receive less data than was requested.

Caution The NetIPC NSF_DATA_WAIT flag can cause a program to block for an extreme period of time (for example, eight minutes for 8 bytes). It is recommended that NetIPC programs not use NSF_DATA_WAIT but loop until all data is received instead. Refer to the "Receiving Data" section of the "NetIPC Calls" chapter for the specific loop information.

If an ipcrecv() call requests more data than is queued on a VC socket, one of the following situations will result:

- If the VC socket is in synchronous mode, the calling process will suspend until enough data is queued to satisfy the ipcrecv() request. If enough data does not arrive within the synchronous time-out period to satisfy the request, a "time out" error (error code 59) will be returned.
- If the VC socket is in asynchronous mode, a "would block" error (error code 56) will be returned.

For more information on receiving data, refer to the discussion of ipcrecv() in the "NetIPC Calls" chapter.

Interpreting Data Received

As stated in the previous discussion of stream modes, the data received by an ipcrecv() call may contain part of a message or multiple messages sent by multiple ipcsend() calls. In addition, if the NSF_DATA_WAIT bit of the flags parameter is *not* set, the receiving process may receive less data than it requested.

If an application does not need to receive data in the form of individual messages, it can simply process the data on the receiving side. However, if an application *is* concerned about messages, the programmer must devise a scheme to allow the receiving side to determine what the messages are and whether all of the expected messages have been received.

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 the actual number of bytes received in its *dlen* parameter, the loop can continue to execute until all of the bytes of the message have been received as indicated by this parameter.

If the length of the messages are *not* known, the sending side may send the length of the message as the first part of each message. In this scenario, the receiving side must execute two `ipcrecv()` loops for each message: the first to receive the length; the second to receive the data.

Synchronous and Asynchronous Socket Modes

When a send operation is performed on a socket, data is moved out of process space into an outbound transmission buffer. Similarly, when a receive operation is performed on a socket, data is moved from an inbound transmission buffer into process space. Sometimes a send or receive request cannot be immediately satisfied. In the case of `ipcsend()`, an empty transmission buffer may not be available; an `ipcrecv()` request may not be satisfiable because data-filled transmission buffers are not queued on the referenced socket. When either of these situations occur, NetIPC must decide whether to fail the request or suspend the process until the request can be satisfied. This decision is based upon whether the socket being manipulated is in synchronous or asynchronous mode.

Sockets are automatically placed in synchronous mode when they are created. When a socket is in synchronous mode, send and receive requests that reference it cause the the calling process to be suspended if the requests cannot be immediately satisfied. A process that has been suspended will remain suspended until the request is satisfied, a synchronous time-out occurs, a signal arrives, or an error is detected. Each synchronous socket has a timer associated with it that can be modified with an `ipccontrol()` call. This timer determines how long a NetIPC call will block on the socket while waiting for its request to be satisfied. A NetIPC call will not block forever unless the synchronous time-out value is set to zero with an `ipccontrol()` call.

Three NetIPC calls, `ipcsend()`, `ipcrecv()` and `ipcrecvcn()`, support asynchronous as well as synchronous I/O. (The `ipcconnect()` call is by definition an asynchronous call; the remaining NetIPC calls do not support asynchronous I/O.) Sockets can be placed in asynchronous mode by calling `ipccontrol()` and specifying `NSC_NBIO_ENABLE` (code 1) in the *`request`* parameter. Send and receive requests directed against a socket in this mode do not cause the calling process to be suspended if the requests cannot be immediately satisfied. Instead, an `NSR_WOULD_BLOCK` (code 56) error is returned and the process is free to perform other tasks before retrying the request.

Refer to the discussion of `ipcrecvcn()` in the "NetIPC Calls" chapter for information about how this call functions in synchronous and asynchronous mode.

Altering the Synchronous Time-out

If the NetIPC calls `ipcsend()`, `ipcrecv()`, and `ipcrecvcn()` are used synchronously, it may be necessary to alter the synchronous time-out value by calling `ipccontrol()`. The default synchronous time-out is 60 seconds. The synchronous time-out determines:

- How long `ipcsend()` will suspend the calling program if it cannot immediately obtain the buffer space needed to accommodate its data or if the process on the receiving end cannot receive the data being sent to it.
- How long `ipcrecv()` will suspend the calling program if its request for data cannot be satisfied or if a "successful" connection status cannot be obtained.
- How long `ipcrecvcn()` will suspend the calling program while waiting for a connection request.

For information on changing the synchronous time-out for specific calls, refer to the call descriptions in the "NetIPC Calls" chapter.

Read and Write Thresholds

For efficiency, a process using asynchronous sockets must be able to determine whether a VC socket can satisfy an `ipcsend()` or `ipcrecv()` call *before* the request is issued. The `ipcselect()` call addresses this problem by providing socket status information. Included in this information is whether or not:

- A VC socket is readable.
- A VC socket is writable.

The `ipcselect()` call determines whether or not a VC socket is readable by examining the socket's read threshold. A VC socket is considered readable if it can immediately satisfy an `ipcrecv()` request for a number of bytes **greater than or equal to** its read threshold. The read threshold is used by `ipcselect()` to check if there are *at least* that many bytes in the system ready for reading.

Similarly, `ipcselect()` determines whether or not a VC socket is writable by examining the socket's write threshold. A VC socket is considered writable if it can immediately satisfy an `ipcsend()` request for a number of bytes **greater than or equal to** its write threshold. The write threshold is used by `ipcselect()` to check if there are *at least* that many bytes in the system to be used as a buffer space for writing. If `ipcselect` indicates that a socket is writable, the subsequent write may still fail due to lack of memory available. System memory may be consumed between the `ipcselect` and the subsequent `ipcsend`.

The `ipcselect()` call will not return accurate status information unless a socket's read and write thresholds are set to the correct number of bytes. (These thresholds are initially set to one byte. You can alter this value by calling `ipccontrol()`. Refer to the discussion of this call for more information.) The number of bytes that you expect to send or receive on a socket should determine the correct read and write threshold settings. As a general rule, you should **set a socket's read threshold to the same number of bytes as the length of the data you expect to receive on that socket.** Similarly, you should **set a socket's write threshold to the same number of bytes you expect to send on that socket.** Consider the following example: Process B will always issue `ipcsend()` calls with 64 bytes of data on VC socket X. Therefore, socket X's write threshold should also be 64 bytes. Similarly, if Process B expects to issue 64-byte `ipcrecv()` requests on socket X, socket X's read threshold should be set to 64 bytes as well.

If you expect to receive variable length data on a particular VC socket, the socket's read threshold should be set to the length of the **shortest** amount of data you expect to receive. If you expect to send variable length data on a particular VC socket, the socket's write threshold should be set to the length of the **longest** amount of data you expect to send.

Note The read and write thresholds are used exclusively by the `ipcselect()` call. They have no effect on other NetIPC calls.

For more information about using sockets in asynchronous mode, refer to the discussions of `ipcselect()`, `ipccontrol()`, `ipcsend()`, `ipcrecv()`, and `ipcrecvcn()`.

Signals

Signals will interrupt NetIPC calls that would otherwise suspend. NetIPC calls that are interrupted by signals are not restartable.

NetIPC calls behave the same way as interruptable HP-UX system calls with the following exception: When a NetIPC call is interrupted by a signal and the *sc_syscall_action* field is set to SIG_RETURN, the following occurs:

1. the NetIPC call aborts,
2. the interrupted call's result parameter is set to NSR_SIGNAL_INDICATION, and
3. the interrupted program continues past the previously blocked NetIPC call.

When an HP-UX system call is interrupted, the *errno* variable is set to EINTR. This does *not* occur when the call is a NetIPC call. Instead, the interrupted call's *result* parameter is set to NSR_SIGNAL_INDICATION.

NetIPC has also defined values to be returned to the sc_syscall field. These values are defined in the HP-UX include file /usr/include/sys/syscall.h.

For more information on signals, refer to signal(2) and sigvector(2) described in the *HP-UX Reference Manual.*

Shutting Down a Connection

Processes should close virtual circuit connections they no longer need by calling `ipcshutdown()` to release the VC socket descriptor that references the connection.

Note The `ipcshutdown()` call can also be used to release call socket descriptors and destination descriptors. Refer to the discussion of `ipcshutdown()` in the "NetIPC Calls" chapter for more information on releasing these types of descriptors.

Because `ipcshutdown()` takes effect very quickly, any data that is in transit on the connection, including any data that has already been queued on the destination VC socket, may be destroyed before its intended recipient is able to receive it. To ensure that no data is lost during connection shutdown, specify the NSF_GRACEFUL_RELEASE flag.

When a NetIPC process releases a VC socket descriptor that is shared by other processes (i.e., other processes have copies of that descriptor), the descriptors owned by the other processes are not affected. The `ipcshutdown()` call does not operate on the VC socket referenced by a VC socket descriptor unless the descriptor is the *only* descriptor for that socket. A VC socket is destroyed along with its VC socket descriptor *only when the descriptor being released is the sole* descriptor for that socket.

Summary of NetIPC Calls

The following is a summary of all the HP 9000 NetIPC calls.

Call	Description
`ipcconnect()`	Requests a virtual circuit to another program and returns a VC socket descriptor which identifies a VC socket endpoint at the calling program.
`ipccontrol()`	Performs special operations on sockets such as enabling synchronous and asynchronous mode, changing the synchronous timeouts, and setting read and write thresholds.
`ipccreate()`	Creates a call socket for the calling program.
`ipcdest()`	Returns a destination descriptor that the calling process can use to establish a connection to another process.
`ipcerrmsg()`	Returns an error message for a particular NetIPC error number.
`ipcerrstr()`	Provides text describing NetIPC error numbers.
`ipcgetnodename()`	Returns the NetIPC node name belonging to the local host.
`ipclookup()`	Searches the socket registry for a socket name and returns a destination descriptor that the calling process can use to establish a connection to another process.
`ipcname()`	Associates a name with a call socket descriptor or destination descriptor and stores it in the socket registry.
`ipcnamerase()`	Removes a name associated with a call socket descriptor or destination descriptor from the socket registry.
`ipcrecv()`	Checks the status of a connection or receives data on a previously established connection.

`ipcrecvcn()`	Receives a connection request from another program and returns a VC socket descriptor that describes a VC socket endpoint at the calling program.
`ipcselect()`	Enables a program to detect and/or wait for the occurrence of any of several events across multiple call or VC sockets. Note that if `ipcselect` indicates that a socket is writable, the subsequent write may still fail due to lack of memory available. System memory may be consumed between the `ipcselect` and the subsequent `ipcsend`.
`ipcsend()`	Sends data to another program on a virtual circuit.
`ipcsetnodename()`	Defines the NetIPC node name for the local host.
`ipcshutdown()`	Releases a descriptor. Also releases the socket referenced by the descriptor if the descriptor is the only descriptor that references that socket.

2

Cross-System NetIPC

NetIPC communication between processes running on computers of different types is referred to as cross-system NetIPC. This chapter describes the special programming considerations that you should be aware of when writing an HP 9000 NetIPC program that will communicate with a peer NetIPC process at an HP 1000 A-Series computer, an HP 3000 (MPE-V or Series 900) computer, or a PC.

NetIPC communication between an HP 9000 Series 600/700/800 and HP 9000 Series 300/400 is not considered cross-system NetIPC because both systems are HP 9000s.

Chapter Overview

Before reading this chapter, you must have a good understanding of the NetIPC concepts and calls. Read the "NetIPC Concepts" chapter and review the "NetIPC Calls" chapter before proceeding.

This chapter does not explain how to write a NetIPC program to run on an HP 1000 A-Series, PC, HP 3000 or HP 3000 Series 900 computer. For this information, refer to the following manuals:

- *NS/1000 User/Programmer Reference Manual.*
- *NetIPC3000/V Programmer's Reference Manual.*
- *NetIPC3000/XL Programmer's Reference Manual.*
- *PC NetIPC/RPM Programmer's Reference Guide.*

The remainder of the material presented in this chapter is organized into the following major sections:

- **Software Revision Codes.** Lists the software revision codes associated with the NetIPC software that provides the cross-system functionality described in this chapter.
- **Local and Remote NetIPC Calls.** Divides NetIPC calls into two categories, local and remote, and describes how these calls are used in cross-system programs.
- **HP 9000 to HP 1000 NetIPC.** Describes differences between the HP 9000 and HP 1000 NetIPC implementations.
- **HP 9000 to HP 3000 NetIPC.** Describes differences between the HP 9000 and HP 3000 NetIPC implementations.
- **HP 9000 to PC NetIPC.** Describes differences between the HP 9000 and PC NetIPC implementations.
- **Process Scheduling.** Describes how to schedule a peer NetIPC process at an HP 9000, HP 3000 and HP 1000 system.

Software Revision Codes

In order for cross-system NetIPC to function properly, the HP 9000, HP 1000 and HP 3000 NetIPC software revision codes must be as follows:

- LAN/9000 Series 600/800 software revision code 1.1 or later for Series 600/800 to HP 1000 A-Series NetIPC.
- LAN/9000 Series 600/800 software revision code 2.1 or later for Series 600/800 to HP 3000 NetIPC.
- NS-ARPA Services software revision code 6.2 or later for the Series 300/400.
- NS/1000 software revision code 5.0 or later for the HP 1000 A-Series.
- ThinLAN 3000/V Link revision code V-Delta-1 MIT or later (used with IEEE 802.3 LAN only) for the HP 3000 MPE-V.
- ThinLAN 3000/XL Link revision code 1.2 or greater for the HP 3000 Series 900.
- PC revision B.00.01.

Local and Remote NetIPC Calls

NetIPC calls can be separated into two categories: local and remote.

Local NetIPC Calls

Local NetIPC calls are used to set up or prepare the local node for interprocess communication with the remote node. The resulting impact of the local call is only to the local node; no information is passed to the remote node.

Because local NetIPC calls do not affect the peer process, there are no cross-system programming considerations associated with these calls. Table 2-1 lists the HP 1000, HP 9000, HP 3000 and PC NetIPC calls that only affect the local process. (An asterisk indicates that a particular call is not implemented.)

Table 2-1. NetIPC Calls Affecting the Local Process

HP 1000	HP 9000	HP 3000	PC
Addopt	addopt()	ADDOPT	AddOpt
Adrof	*	*	*
InitOpt	initopt()	INITOPT	InitOpt
*	*	IPCCHECK	*
IPCControl	ipccontrol()	IPCCONTROL	IPCControl
IPCCreate	ipccreate()	IPCCREATE	IPCCreate
*	ipcerrmsg()	IPCERRMSG	*
*	ipcerrstr()	*	*
*	ipcgetnodename()	*	*
IPCGet	*	IPCGET	*
IPCGive	*	IPCGIVE	*
IPCName	ipcname()	IPCNAME	*
IPCNamerase	ipcnamerase()	IPCNAMERASE	*
IPCSelect	ipcselect()	*	*
*	ipcsetnodename()	*	*
*	optoverhead()	OPTOVERHEAD	OptOverhead
Readopt	readopt()	READOPT (NetIPC 3000/V only)	ReadOpt
*	*	*	ConvertNetworkLong
*	*	*	ConvertNetworkShort
*	*	*	IPCWait

Although the calls listed in Table 2-1 do not affect cross-system communication, keep in mind that you may need to design NetIPC programs for different system types differently. This is because NetIPC calls, even those with the same name, differ from system type to system type. The following are some local call differences to be aware of:

- **Maximum number of sockets.** The maximum number of socket descriptors owned by an HP 9000 process at any given time is 2048 (including file descriptors); the HP 1000 maximum is 32; the HP 3000 maximum is 64; the PC maximum is 21. This number includes call socket, virtual circuit socket, and open file descriptors.

- **ipccontrol() parameters.** The `ipccontrol()` call supports a different set of request codes on different system types. Refer to the NetIPC documentation for a particular system (this manual for the HP 9000) for a full description of the request codes available on that system.
- **Destination descriptors.** On the HP 1000, destination descriptors are called path report descriptors. Both types of descriptors are used in the same way. They contain addressing information that is used by a NetIPC process to direct requests to a certain call socket at a certain node.
- **Manipulation of descriptors.** The HP 9000 and HP 1000 implementations of NetIPC allow you to manipulate call socket and destination descriptors with the `ipcname()` and `ipcnamerase()` calls; the HP 3000 only allows you to manipulate call sockets with these calls. When you use the IPCGive and IPCGet calls on the HP1000, you can manipulate call socket and destination descriptors; the HP 3000 only allows you to manipulate call and VC sockets with these calls.
- **Asynchronous I/O.** The HP 9000 and HP 1000 NetIPC implementations utilize the NetIPC `ipcselect()` call to perform asynchronous I/O; the HP 3000 NetIPC implementation utilizes the MPE intrinsics `IOWAIT` and `IODONTWAIT`. PC NetIPC uses `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 9000, HP 1000, HP 3000, and PC. Refer to the NetIPC documentation for each system for more information.

Remote NetIPC Calls

Unlike local NetIPC calls, remote NetIPC calls affect the peer process at the remote node. Because remote NetIPC calls affect the peer process, there may be cross-system programming considerations associated with these calls.

Table 2-2 lists the HP 9000, HP 1000, HP 3000, and PC NetIPC calls that affect the remote process.

Table 2-2. NetIPC Calls Affecting the Remote Process

HP 1000	HP 9000	HP 3000	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

The remainder of this chapter describes cross-system programming considerations for the remote NetIPC calls as they relate to the following cross-system pairs:

- HP 9000 to HP 1000 A-Series communication.
- HP 9000 to HP 3000 (MPE-V and Series 900) communication.
- HP 9000 to PC communication.

HP 9000 to HP 1000 NetIPC

When writing an HP 9000 NetIPC program that will communicate with an HP 1000 NetIPC peer process, you must be aware of certain differences in the HP 9000 and HP 1000 NetIPC implementations. These differences, and the NetIPC calls that are affected, are listed in Table 2-3.

Table 2-3. Calls That Affect HP 9000 to HP 1000 NetIPC

NetIPC Call	Cross-System Considerations
`ipcconnect()`	**Checksumming** - When an `ipcconnect()` call is executed on an HP 9000 node, then TCP checksumming is always enabled for the HP 9000-to-HP 1000 connection. **Send and Receive sizes** - The HP 1000 send and receive size range is 1 to 8,000 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. Just be sure to specify a buffer size within the correct range for the respective system.
`ipcdest()`	**TCP Protocol Address** - The HP 1000 and HP 9000 implementations of `ipccreate()` support different ranges of permitted TCP protocol addresses that can be specified in the *`opt`* parameter. However, both implementations recommend that users specify TCP addresses in the range of 30767 to 32767 decimal for cross-system use. The `ipcdest()` call uses the TCP protocol address specified in `ipccreate()` on the remote process.
`ipclookup()`	No differences that affect cross-system operations.

Table 2-3. Calls That Affect HP 9000 to HP 1000 NetIPC-con't

NetIPC Call	Cross-System Considerations
`ipcrecv()`	**Receive size** - The HP 1000 receive size range is 1 to 8,000 bytes. The HP 9000 receive size range is 1 to 32,767 bytes. Although the range sizes that can be specified in the *dlen* parameter are different, cross-system communication is not affected. Just be sure to specify a buffer size within the correct range for the respective system.
`ipcrecvcn()`	**Checksumming** - TCP checksumming is always enabled for the HP 9000-to-HP 1000 connection. **Send and Receive sizes** - The HP 1000 send and receive size range is 1 to 8,000 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. Just be sure to specify a buffer size within the correct range for the respective system.
`ipcsend()`	**Send size** - The HP 1000 send size range is 1 to 8,000 bytes. The HP 9000 send size range is 1 to 32,767 bytes. Although the range sizes that can be specified in the *dlen* parameter are different, cross-system communication is not affected. Just be sure to specify a buffer size within the correct range for the respective system.

Table 2-3. Calls That Affect HP 9000 to HP 3000 NetIPC-con't

NetIPC Call	Cross-System Considerations
`ipcshutdown()`	**Socket Shut Down** - The shutdown procedure for both HP 1000 and HP 9000 processes is the same, except that a "graceful release" flag is provided on the HP 9000. If the graceful release flag (*`flags`* bit 17) is set on the HP 9000, the HP 1000 will respond as though it were a normal shutdown. The HP 9000 also supports "shared sockets"; the HP 1000 does not. 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.

Note There are many additional differences between remote NetIPC calls for the HP 9000 and HP 1000 systems. However, these differences should not affect the cross-system communication capabilities of your program because they affect the local node only. Refer to Appendix E, "Porting NetIPC Programs," for a summary of the differences between the HP 9000 and HP 1000 NetIPC implementations.

NetIPC Error Codes

NetIPC calls with the same names on HP 9000 and HP 1000 systems may return different error codes. Refer to the system's corresponding NetIPC documentation for a complete list of the error codes that are applicable to that NetIPC implementation.

HP 9000 to HP 3000 NetIPC

When writing an HP 9000 NetIPC program that will communicate with an HP 3000 (MPE-V or Series 900) NetIPC peer process, you must be aware of certain differences in the HP 9000 and HP 3000 NetIPC implementations. These differences, and the NetIPC calls that are affected, are listed in Table 2-4.

Table 2-4. Calls That Affect HP 9000 to HP 3000 NetIPC

NetIPC Call	Cross-System Considerations
`ipcconnect()`	**Checksumming** - TCP checksumming is always enabled for the HP 9000-to-HP 3000 connection. **Send and Receive sizes** - The HP 9000 send and receive size range is 1 to 32,767 bytes. The HP 3000 send and receive size range is 1 to 30,000 bytes. Although the ranges are different, cross-system communication is not affected. Just be sure to specify a buffer size within the correct range for the respective system. Note that the default send and receive sizes differ on the HP 9000 and the HP 3000. On the HP 9000, the default send and receive size is 100 bytes. On the HP 3000, the default send and receive size is less than or equal to 1024 bytes.

Table 2-4. Calls That Affect HP 9000 to HP 3000 NetIPC-con't

NetIPC Call	Cross-System Considerations
`ipcdest()`	**TCP Protocol Address** - The HP 9000 and HP 3000 implementations of `ipccreate()` support different ranges of permitted TCP protocol addresses that can be specified in the *`opt`* parameter. However, both implementations recommend that users specify TCP addresses in the range of 30767 to 32767 decimal for cross-system use. The `ipcdest()` call uses the TCP protocol address specified in `ipccreate()` on the remote process.
`ipclookup()`	No differences that affect cross-system operations.
`ipcrecv()`	**Receive size** - The HP 9000 receive size range is 1 to 32,767 bytes. The HP 3000 receive size range is 1 to 30,000 bytes. Although the ranges are different, cross-system communication is not affected. Just be sure to specify a buffer size within the correct range for the respective system.
`ipcrecvcn()`	**Checksumming** - TCP checksumming is always enabled for the HP 9000-to-HP 3000 connection. **Send and Receive sizes** - The HP 9000 send and receive size range is 1 to 32,767 bytes. The HP 3000 send and receive size range is 1 to 30,000 bytes. Although the ranges are different, cross-system communication is not affected. Just be sure to specify a buffer size within the correct range for the respective system. Note that the default send and receive sizes differ on the HP 9000 and HP 3000. On the HP 9000, the default send and receive size is 100 bytes. On the HP 3000, the default send and receive size is less than or equal to 1024 bytes.

Table 2-4. Calls That Affect HP 9000 to HP 3000 NetIPC-con't

NetIPC Call	Cross-System Considerations
`ipcsend()`	**Send size** - The HP 9000 send size range is 1 to 32,767 bytes. The HP 3000 send size range is 1 to 30,000 bytes. Although the ranges are different, cross-system communication is not affected. Just be sure to specify a buffer size within the correct range for the respective system. **Urgent Data** - The HP 3000 supports an "urgent data" option in the *opt* parameter. If this bit is set by the HP 3000 program, it will be ignored by the receiving process on the HP 9000.
`ipcshutdown()`	**Socket Shut Down** - The shutdown procedure for HP 9000 and HP 3000 processes is the same. The HP 9000 supports "shared sockets"; the HP 3000 does not. Shared sockets are not destroyed until only one socket descriptor exists (the last socket descriptor). Therefore, an HP 9000 process may take longer to close the connection than expected.

Note There are many additional differences between remote NetIPC calls for the HP 9000 and HP 3000 systems. However, these differences should not affect the cross-system communication capabilities of your program because they affect the local node only. Refer to the system's corresponding NetIPC documentation to determine all of the differences between NetIPC on the HP 9000 and HP 3000 systems.

NetIPC Error Codes

NetIPC calls with the same names on HP 9000 and HP 3000 systems may return different error codes. Refer to the system's corresponding NetIPC documentation for a complete list of the error codes that are applicable to that NetIPC implementation.

HP 9000 to PC NetIPC

When writing an HP 9000 NetIPC program that will communicate with a PC NetIPC peer process, you must be aware of certain differences in the HP 9000 and PC NetIPC implementations. These differences, and the NetIPC calls that are affected, are listed in Table 2-5.

Table 2-5. Calls That Affect HP 9000 to PC NetIPC

NetIPC Call	Cross-System Considerations
`IPCConnect`	**Checksumming** - With PC NetIPC, the TCP checksum option cannot be turned on. On the HP 9000, the TCP checksum is always on. Therefore, the checksum is in effect on both sides of the connection. **Send and Receive sizes** - The HP 9000 send and receive size range is 1 to 32,767 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. Just be sure to specify a buffer size within the correct range for the respective system; otherwise, an error will occur. For example, if a PC sends a 60,000 byte buffer, the HP 9000 process may get all the data by posting two `ipcrecv` functions of 30,000 bytes until all the data has been received.
`IPCCreate` `IPCDest`	**TCP Protocol Address** - The HP 9000 and PC implementations of `IPCCreate` support different ranges of permitted TCP protocol addresses that can be specified in the *opt* parameter. However, both implementations recommend that users specify TCP addresses in the range of 30767 to 32767 decimal for cross-system use. The `IPCDest` call uses the TCP protocol address specified in `IPCCreate` on the remote process.

Table 2-5. Calls That Affect HP 9000 to PC NetIPC-con't

NetIPC Call	Cross-System Considerations
IPCRecv	**Receive size** - The HP 9000 enables you to specify the maximum receive size of the data buffer with the *dlen* parameter through an option array. 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 9000 will determine the receive size buffer on the PC.
IPCRecvCn	**Checksumming** - With PC NetIPC, the TCP checksum option cannot be turned on. On the HP 9000, the TCP checksum is always on. Therefore, the checksum is in effect on both sides of the connection. **Send and Receive sizes** - The HP 9000 send and receive size range is 1 to 32,767 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. Just be sure to specify a buffer size within the correct range for the respective system.
IPCSend	**Send size** - The HP 9000 enables you to specify the maximum send size of the data buffer with the *dlen* parameter through an option array. PC NetIPC has no option array defined for IPCConnect. This does not affect cross-system communication. The maximum send size of the data in the buffer on the HP 9000 will determine the send size buffer on the PC.

Note There are many additional differences between remote NetIPC calls for the HP 9000 and PC NetIPC systems. However, these differences should not affect the cross-system communication capabilities of your program because they affect the local node only. Refer to the system's corresponding NetIPC documentation to determine all of the differences between NetIPC on the HP 9000 and PC NetIPC systems.

NetIPC Error Codes

NetIPC calls with the same names on HP 9000 and PC NetIPC systems may return different error codes. Refer to the system's corresponding NetIPC documentation for a complete list of the error codes that are applicable to that NetIPC implementation.

Process Scheduling

NetIPC does not include a call to schedule a peer process. In programs communicating between multiple HP 3000s or multiple HP 1000s, you can use the Remote Process Management (RPM) call RPMCREATE to programmatically schedule program execution. RPM between HP 9000s and HP 1000s or HP 3000s is not currently supported by Hewlett-Packard.

The following sections describe how to start a process at a remote HP 9000, HP 1000 and HP 3000 system.

Remote HP 9000 Process

Remote HP 9000 processes can be started manually or can be scheduled by daemons.

To manually start up a NetIPC program, simply logon to the HP 9000 system and run the NetIPC program.

To start a NetIPC process from a daemon, start the daemon at system start up by invoking it from the /etc/netlinkrc file.

Remote HP 1000 Process

A remote HP 1000 NetIPC process must be ready to execute by being an RTE type 6 file. HP 1000 processes can be started manually or can be started at system start up.

To manually start up a 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. Refer to the *RTE-A User's Manual* for a description of the XQ command.

Remote HP 3000 Process

HP 3000 processes can be started manually or can be started by a job file.

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.

Remote PC NetIPC Process

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 with the mouse on the executable file.

3

NetIPC Calls

This chapter is a reference source for programmers who code applications that utilize NetIPC calls. It is assumed that the reader has read and understands the concepts presented in the "NetIPC Concepts" chapter.

The information contained in this chapter is organized as follows:

- **Programming Languages.** Identifies the programming languages in which NetIPC programs may be written.
- **Include Files and Libraries.** Describes the NetIPC include file that may be used with NetIPC programs written in C and explains how to link and compile NetIPC programs written in each of the supported programming languages.
- **HP 1000 to Series 600/800 Migration.** Lists reference sources for programmers who will be migrating HP 1000 NetIPC programs to the HP 9000 Series 600/800 programming environment.
- **NetIPC Common Parameters.** Describes parameters that are common to most of the NetIPC calls and explains how to use those parameters in each of the supported programming languages.
- **Syntax Conventions.** Explains the syntax conventions used on the NetIPC reference pages.
- **NetIPC Reference Pages.** Provides reference pages, in alphabetical order, for each of the NetIPC calls.

Programming Languages

NetIPC programs may be written in C, Pascal or FORTRAN. For detailed information about programming languages, refer to the appropriate language reference manual. Programming reference manual titles are listed in the preface of this manual.

Include Files and Libraries

A C include file, `/usr/include/sys/ns_ipc.h`, is provided with the NetIPC software and should be included in all NetIPC programs that are written in the C programming language. This include file contains constant definitions for socket types, protocol types, *`flags`* parameter bits, `ipccontrol()` request codes, *`opt`* parameter option codes, and NetIPC error codes. It also contains the type declaration `ns_int_t` that can be used to describe many of the NetIPC call parameters.

Note If you wish to use Pascal or FORTRAN, you must translate the include file `/usr/include/sys/ns_ipc.h` into these programming languages.

A NetIPC library, `/usr/lib/llib-nsipc.ln` is also provided with the NetIPC software for use with the lint program. For example:

```
lint programname -lnsipc
```

For more information on `lint`, refer to the *HP-UX Reference Manual.*

HP 1000 to Series 600/800 Migration

NetIPC programs written in Pascal and FORTRAN for the HP 1000 environment may be transported to HP 9000 Series 600/800 nodes. Refer to the migration manuals listed in the preface of this manual for information on migrating HP 1000 Pascal and FORTRAN programs to the HP 9000 Series 600/800 environment.

If you plan to transport HP 1000 NetIPC programs to the Series 600/800, refer to Appendix E of this manual for HP 1000 to Series 600/800 NetIPC porting information. Refer to Appendix D of this manual for general NS/1000 and DS/1000-IV to LAN/9000 Series 600/800 migration information.

NetIPC Common Parameters

The *flags*, *opt*, *data*, *result*, *socketname*, and *nodename* parameters are common to many NetIPC calls.

The *opt* parameter provides functionality for NetIPC calls and usually has associated data. The *flags* parameter enables or disables certain functions for NetIPC calls. The *result* parameter returns error codes for NetIPC calls. The *socketname* and *nodename* parameters identify sockets and nodes, respectively.

The following paragraphs provide detailed information regarding the meaning, use and structure of each of these parameters.

Flags Parameter

The *flags* parameter is a 32-bit integer that represents various options. By setting bits in the *flags* parameter, you can invoke various services in `ipcrecv()`, `ipcsend()`, `ipccontrol()` and `ipcdest()` calls.

The NetIPC calls `ipcconnect()`, `ipccreate()`, `ipclookup()`, `ipcrecvcn()` and `ipcshutdown()` also include a *flags* parameter, but in these calls the parameter is reserved for future use. The *flags* parameter must be initialized to zero before it is used in these calls. The parameter must also be cleared *after* it is used in these calls if it is to be in a subsequent call that requires that the *flags* parameter be initialized to zero. This precaution should be taken because NetIPC calls that do not use the *flags* parameter on input may return non-zero values to the parameter on output.

The following paragraphs explain how the *flags* parameter is declared and manipulated in the C, Pascal, and FORTRAN programming languages.

Note NetIPC calls assume that the bits in the *flags* parameter are numbered from left to right with the most significant bit considered to be bit zero and the least significant bit considered to be bit 31.

MSB
0 1 2 3 4 5 ... 31 Pascal, C, and NetIPC
MSB
31 30 29 28 ... 0 FORTRAN

The remaining examples in this chapter assume the most significant bit is 0.

Using Flags in a C Program

The C include file /usr/include/sys/ns_ipc.h includes constant definitions that should be used when setting bits in the *flags* parameter. (Refer to the explanations of the ipcsend(), ipcrecv(), ipccontrol(), and ipcdest() call descriptions later in this chapter for the constants that can be used with these calls.)

The *flags* parameter should be declared as type ns_int_t, which is defined in the C include file /usr/include/sys/ns_ipc.h. A flags option is set by assigning one of the constants defined for the particular call to the *flags* parameter. In the following example, the *flags* parameter used in an ipcrecv() call is assigned the constant NSF_PREVIEW. (NSF_PREVIEW sets bit 30 of the *flags* parameter.)

```
flags = NSF_PREVIEW;
```

In the next example, the NSF_PREVIEW and NSF_DATA_WAIT options are selected by using the bitwise inclusive OR (|) operator. (NSF_DATA_WAIT sets bit 20 of the *flags* parameter.)

```
flags = NSF_PREVIEW | NSF_DATA_WAIT;
```

Using Flags in a Pascal Program

In Pascal, the *flags* parameter may be represented as an array of bits:

```
TYPE      flags_type = packed array [0..31] of boolean;

VAR       flags : flags_type;
```

flags [0] refers to the high order bit in the boolean array; flags [31] refers to the low order bit. To set a bit in the array, assign the value TRUE to the desired bit. For example,

```
flags [21] := TRUE;
```

would set bit 21 of the *flags* array. A clear bit would be assigned the value FALSE. If you do not want to set any of the bits in the *flags* array, but you want to be certain that all of the bits are clear, you may make *flags* type INTEGER and assign it the value zero.

Using Flags in a FORTRAN Program

In FORTRAN, the *flags* parameter must be declared as INTEGER*4 (32-bit integer). The simplest way to set a bit in this parameter is to use the FORTRAN library function ibset(*a*, *b*). The *flags* parameter is passed in the first argument (*a*) and the bit position to be set is passed in the second argument (*b*).

The ibset function assumes that bits are numbered from right to left, with the most significant bit considered to be bit 31 and least significant bit considered to be bit 0. NetIPC calls assume that bits are numbered in the opposite direction (i.e., the most significant bit is 0, the least significant bit is 31). Therefore, to set the proper bit in the *flags* parameter using ibset, you must subtract the *flags* value from 31.

In the following example, bit 21 is set in the *flags* parameter:

```
      INTEGER*4 flags
C     The flags value is subtracted from 31 so that the proper
C     bit is set. This maps ibset's bit numbering convention into
C      NetIPC's.
      flags = ibset(flags,(31-21))
```

Multiple bits can be set by repeating the ibset function.

Opt Parameter

The *opt* parameter allows you to request optional services when invoking certain NetIPC calls. It enables calls that include the *opt* parameter to accept an arbitrary number of arguments that are either protocol or operating system specific.

Because the *opt* parameter has a complex structure, NetIPC provides a special set of calls that allow you to manipulate the parameter. Table 3-1 summarizes the *opt* parameter calls. Before you can invoke a NetIPC call that includes an *opt* parameter, you must prepare the parameter by using the following *opt* parameter calls:

- First, `initopt()` must be called to initialize the *opt* parameter. This call allows you to specify how many arguments will be placed in the parameter.
- Next, `addopt()` must be called to add an argument and its associated data to the *opt* parameter. (An `addopt()` call can add only one argument at a time, so you must call it multiple times if you want to add multiple arguments to the *opt* parameter.)

If the *opt* parameter is *not* used in a certain call (no options are defined for that call or you do not choose to select an option), you must assign a value of zero (0) to the *opt* parameter or pass the constant `NSO_NULL` in its place.

In addition to `initopt()` and `addopt()`, two optional *opt* parameter calls are provided: `readopt()` and `optoverhead()`. The `readopt()` call allows you to obtain option code and argument data associated with a certain *opt* parameter. The `optoverhead()` call may be used to determine the number of bytes needed for the *opt* parameter, excluding the data area. To determine the length of the *entire opt* parameter, you must add the result of the `optoverhead()` call to the length of the data to be placed in it and then allocate memory for the parameter by calling `malloc()`. (`malloc()` is documented in the *HP-UX Reference Manual.*)

The following formula can also be used to determine the *opt* parameter length *before* coding your application.

```
total_length_of_opt = 4 + 8 * OPTNUMARGUMENTS + DATA;
```

In this formula, OPTNUMARGUMENTS contains the number of arguments that will be placed in the parameter and DATA contains the length in bytes of the data associated with all of the arguments.

Table 3-1. Special NetIPC Calls

Call	Description
`addopt()`	Adds an argument and its associated data to an *opt* parameter.
`initopt()`	Initializes an *opt* parameter so that arguments can be added.
`optoverhead()`	Returns the amount of space needed for the *opt* parameter in bytes, not including the data portion of the parameter.
`readopt()`	Obtains the option code and argument data associated with an *opt* parameter argument.

A complete description of each *opt* call, including programmatic examples of the `initopt()` and `addopt()` calls, is provided in "Special NetIPC Calls" later in this chapter.

The following paragraphs explain how the *opt* parameter is declared and manipulated in the C, Pascal, and FORTRAN programming languages.

Using Opt in a C Program

The C include file `/usr/include/sys/ns_ipc.h` includes constant definitions that should be used when placing options in the *opt* parameter. (Refer to the explanations of the NetIPC calls that utilize the *opt* parameter for a description of the constants that can be used.)

The *opt* parameter should be declared as an array of short (16-bit) integers. For example:

```
short int opt [optlength];
```

When declared as a array of short integers, the *opt* parameter can be passed directly to the `initopt()` call. For example,

```
initopt (opt, optnumarguments, error);
```

Alternatively, you can declare the *opt* parameter as a pointer to a short (16-bit) integer. For example:

```
short int *opt;
```

However, if you use `*opt`, you must allocate space for the structure before passing it to `initopt()`. This can be done by using `optoverhead()` and the `malloc()` call as described in the *HP-UX Reference Manual.* For example:

```
datalength = 20;
optlength = optoverhead (number_entries, error);
opt = (short*) malloc (datalength + optlength);
```

Note The *opt* data structure must be aligned on a short (16-bit) boundary.

Using Opt in a Pascal Program

In Pascal, the *opt* parameter should be declared as a a packed array of bytes. For example:

```
TYPE
  byte      = 0..255;
  opt_array = packed array [0..optlength] of byte;
VAR
   opt : opt_array;
```

Using Opt in a FORTRAN Program

In FORTRAN, the *opt* parameter should be declared as an array of short (16-bit) integers. For example:

```
SHORT INTEGER opt(optlength)
```

Opt Parameter Structure

Note The following description is provided for information only. The special *opt* parameter calls are provided to mask this information from the user. It is not necessary to understand the *opt* parameter structure in order to use it.

The following diagrams are provided to illustrate the general form of the *opt* parameter after it has been initialized with the special NetIPC call `initopt()`. In Figure 3-1, OPTLENGTH represents the length of the *opt* parameter from the first byte of OPTNUMARGUMENTS to the end of the data segment:

```
OPTLENGTH = 8 * OPTNUMARGUMENTS + DATA
```

OPTNUMARGUMENTS represents the number of arguments or entries placed in the parameter; OPTARGUMENTS is an area containing the arguments themselves; and DATA is where the data associated with the arguments is stored.

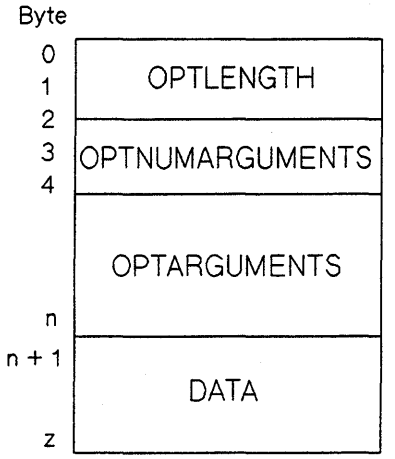

Figure 3-1. Opt Parameter Structure

Figure 3-2 illustrates the structure of an *opt* parameter argument. OPTIONCODE is the option code associated with the argument being added; OFFSET is a byte offset into the *opt* record where any data associated with the argument is located; and DATALENGTH is the length of the data associated with the argument. This information is added to the *opt* parameter with the special NetIPC call `addopt()`. (An example of adding an argument to the *opt* parameter is provided in the discussion of `addopt()` later in this chapter.)

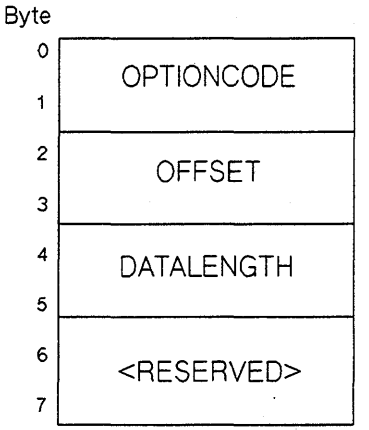

Figure 3-2. OPTARGUMENT Structure

Data Parameter

The *data* parameters present in `ipcsend()`, `ipcrecv()` and `ipccontrol()` may reference data vectors or data buffers.

Unlike a data buffer, which is a structure containing actual data, a data vector is a structure that can *describe* several *data objects*. The description of each object consists of a byte address and a length. The byte address describes where the object is located and the length indicates how much data the object contains. Any kind of data object (arrays, portions of arrays, records, simple variables, etc.) can be described by a data vector.

When a data vector is used to identify data to be sent, it describes where the data is located. This is referred to as a **gathered write**. When a data vector is used to identify data to be received, it describes where the data is to be placed. This is referred to as a **scattered read.**

Using data vectors may be more efficient than using data buffers in certain circumstances. For example, a process that sends data from several different buffers must call `ipcsend()` several times, or copy the data into a packing buffer prior to sending it, if its *data* parameter is a data buffer. However, if its *data* parameter is a data vector, the process may describe all of the buffers in the *data* parameter and transfer it using one `ipcsend()` call.

Note Since the data location descriptors contain machine-specific information, code using the vectored option may not be portable to other machines.

Figure 3-3 is an example of a data vector and the data objects that it represents. The data vector describes the characters "**HERE IS THE DATA.**"

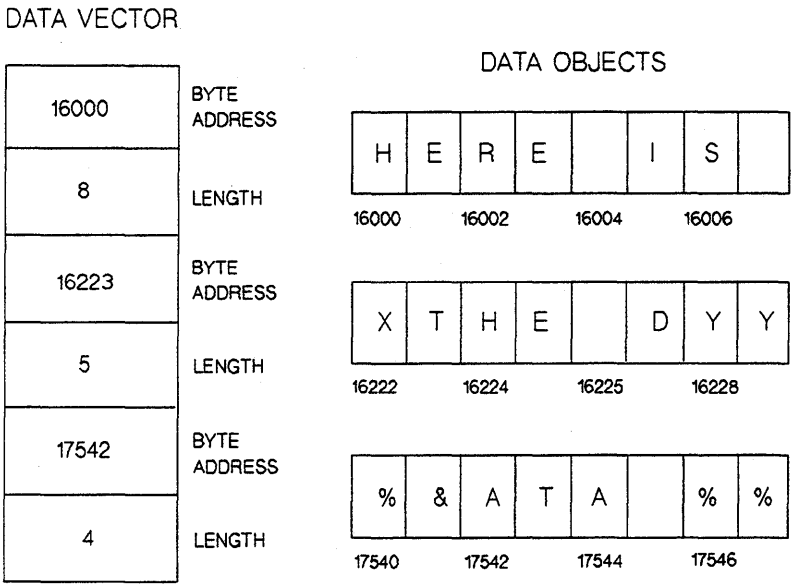

Figure 3-3. Vectored Data

When a *data* parameter refers to a data vector, the length of the data parameter (usually called *dlen*) refers to the *length of the structure containing the vector.*

For example, if an `ipcsend()` call were to reference the data vector in Figure 3-3 above, its *dlen* parameter would be 24 bytes. (Each byte address and length totals 4 bytes; each pointer to a data object is also 4 bytes long. There are three sets of byte addresses, lengths and pointers. Therefore, 8 * 3 = 24.)

Each length in a data vector must be greater than or equal to zero. The format for vectors, and the maximum number of vectors that may be specified, are defined in `/usr/include/sys/uio.h`.

Result Parameter

Every NetIPC call has a *result* parameter. When a NetIPC call encounters an error, an error code is returned via this parameter.

Using Result in a C Program

Because the *result* parameter is provided for error return, you should declare NetIPC calls to be type `void` in C programs. In addition, a C include file called `/usr/include/sys/ns_ipc.h` is provided which contains constant definitions that can be used to refer to errors in your C programs. For example, the following C program fragment checks for the error `NSR_REMOTE_ABORT` (code 64) after an `ipcshutdown()` call:

```
ipcshutdown(descriptor, &flags, opt, &result);
  if (result == NSR_REMOTE_ABORT)
    goto return_error;
```

The *result* parameter should be declared as a pointer to type `ns_int_t`, which is defined in the C include file `/usr/include/sys/ns_ipc.h`.

Note Passing an invalid or out-of-bounds pointer to the actual *result* argument in a NetIPC call will cause the program to core dump due to a memory fault/bus error. A pointer is, in general, considered "bad" if it points outside of the user's memory space.

Using Result in a Pascal Program

In Pascal, the *result* parameter should be declared as type `INTEGER`.

Using Result in a FORTRAN Program

In FORTRAN, the *result* parameter should be declared as type `INTEGER*4`.

Socket Name Parameter

The NetIPC calls `ipcname()`, `ipcnamerase()`, `ipclookup()` and `ipcdest()` require the use of names to identify either sockets or nodes. A **socket name** (the *socketname* parameter) may be a maximum of 16 characters long and may consist of any ASCII character. Upper and lower case characters are not considered distinct (for example, the socket names "john" and "JOHN" are equivalent).

Node Name Parameter

A **node name** (the *nodename* parameter) refers to a node and has a hierarchical structure as follows:

node[.*domain*[.*organization*]]

The *domain* and *organization* may be useful for grouping nodes and collections of nodes, but they currently have no special meaning regarding the structure of the network within the LAN product and are optional. They will default to the local domain and organization if they are omitted. When all three parts of the node name are specified, it is called a **fully-qualified** node name.

Each *node*, *domain*, and *organization* name is a maximum of 16 characters long, and a period (.) separates each name. The maximum total length of a fully-qualified node name is 50 characters. All alphanumeric characters are allowed, including the underscore (_) and dash (-) characters, but the first character of each parameter must be alphabetic. Upper and lower case characters are not considered distinct. For example: `ANIMAL.DCL.IND` would indicate node ANIMAL in the DCL lab (domain) of the IND division (organization).

Syntax Conventions

The the syntax conventions used in this chapter are described below:

- Constant names defined in the C include file `/usr/include/sys/ns_ipc.h` are included in the parameter descriptions for calls that can use them.
- A section titled "Programming Considerations" is included at the end of each NetIPC call reference page. This section consists of a table that lists the type definitions and passing modes that must be used for each call parameter. This table includes information for the C, Pascal, and FORTRAN programming languages.

NetIPC Reference Pages

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

Note Standard HP-UX "manual reference page" versions of the following NetIPC reference pages are also provided on-line and in the *LAN Reference Pages* manual (for the Series 600/800) and the *Network Services Reference Pages* manual (for the Series 300).

---

addopt()

Adds an argument and its associated data to the *opt* parameter.

Syntax

```
addopt(opt,argnum,optioncode,datalength,data,result)
```

Parameters

opt	The *opt* parameter to which you want to add an argument. Refer to "NetIPC Common Parameters" for information on the structure and use of this parameter.
argnum	The number of the argument to be added. The first argument is number zero.
optioncode	The option code or constant definition (C programs only) for the argument to added. These codes are described in each NetIPC call *opt* parameter description.
datalength	The length in bytes of the data to be included. This information is provided in each NetIPC call *opt* parameter description.
data	An array containing the data associated with the argument.
result	The error code returned; zero or NSR_NO_ERROR if no error.

Description

The addopt() call adds an argument and its associated data to an option buffer. The parameter must be initialized by initopt() before arguments can be added.

The following C program fragment illustrates the use of initopt() and addopt() to initialize and add two arguments to the option parameter of an ipcconnect() call.

In this example, the *opt* parameter is used to specify a maximum send size and maximum receive size of 1000 bytes. (Maximum send size indicates the maximum number of bytes that you expect to send with a single `ipcsend();addopt()` call and maximum receive size indicates the maximum number of bytes you expect to receive with a single `ipcrecv()` call.) The *opt* parameter is assumed to be previously defined as an array of short integers.

Note In the following example, it is assumed that the *opt* and *data* parameters were previously declared as arrays of short (16-bit) integers. Refer to "Opt Parameter Structure" earlier in this chapter for more information about the *opt* parameter.

---

addopt Example

```
/* initopt initializes the opt parameter to contain two    */
/* arguments   one for the maximum send size and one for  */
/* the maximum receive size.                               */
     optnumarguments = 2;
     initopt (opt, optnumarguments, &error);
        .
        .
     /* perform error checking here */
        .
        .
/* addopt is called to add the maximum send size.  The data  */
/* parameter contains the value 1000.  The data parameter    */
/* was previously declared as an array of short integers. Note*/
/* that the first argument is number zero.                    */
    argnum = 0;
    optioncode = NSO_MAX_SEND_SIZE;
    datalength = 2;
    data[0] = 1000;
    addopt (opt, argnum, optioncode, datalength, data, &error);
/* addopt is called once more to add the maximum receive size */

/* Note that the data and datalength parameters are unchanged.*/

argnum = 1;    optioncode = NSO_MAX_RECV_SIZE;
addopt (opt, argnum, optioncode, datalength, data, &error);
        .
        .
/* perform error checking here */
/* ipcconnect can now be called with the opt parameter.    */
```

Programming Considerations

The following is a list of the type definitions and passing modes for the `addopt()` call parameters in C, Pascal, and FORTRAN.

Parameter	C	PASCAL	FORTRAN
opt	short int opt []	array of bytes by reference	array of integers by reference
argnum	short int argnum	int16* by value	integer by value
optioncode	short int optioncode	int16* by value	integer by value
datalength	short int datalength	int16* by value	integer by value
data	short int data[]	int16* by reference	array of integers by reference

*int16 is a user-defined Pascal type for a 16-bit integer.

initopt()

Initializes the *opt* parameter so that arguments can be added.

Syntax

```
initopt(opt,optnumarguments,result)
```

Parameters

opt	The *opt* parameter to be initialized. Refer to "NetIPC Common Parameters" for information on the structure and use of this parameter.
optnumarguments	The number of arguments that will be placed in the *opt* parameter. If this parameter is zero, the *opt* parameter will be initialized to contain zero arguments.
result	The error code returned; zero or NSR_NO_ERROR if no error.

The `initopt()` call must be used to initialize the *opt* parameter prior to adding arguments to it with `addopt()`. The *optnumarguments* parameter specifies how many arguments can be placed in the *opt* parameter. For example, if zero is specified, no arguments can be added to the *opt* parameter; if three is specified, three arguments must be added.

In the following C program fragment, the same *opt* parameter is prepared for use in two different `ipcconnect()` calls. The first call will request a connection with the default maximum send and receive sizes (100 bytes), so its option parameter is initialized to contain zero arguments. The second `ipcconnect()` call will request a connection with a maximum send and receive size of 1000 bytes. Thus, its option parameter must be initialized to contain two arguments, the first to contain the maximum send size, and the second to contain the maximum receive size.

Note In the following example, it is assumed that the ***opt*** and ***data*** parameters have been previously declared as arrays of short (16-bit) integers. Refer to the section titled "Opt Parameter" earlier in this chapter for more information about the ***opt*** parameter.

```
/* initopt initializes the opt parameter to be used in an */
/* ipcconnect call to contain zero entries.  This will cause  */
/* the maximum send and receive sizes to default to 100 bytes.*/

optnumarguments = 0;

initopt (opt, optnumarguments, &error);

        .
        .

/* perform error checking here */

        .
        .

/*initopt reinitializes the opt parameter to be used in another*/
/*ipcconnect call.  This call specifies the maximum  */
/*send and receive sizes, so it must be initialized to contain*/
/*two arguments.                                      */

optnumarguments = 2;
initopt (opt, optnumarguments, &error);

          .
          .

/* perform error checking here */

          .
          .
```

```
/*The addopt call is used to add the maximum send size argument*/
/*as the first argument to the opt parameter.  The maximum*/
/*send size has an option code of 3.  The data parameter has been */
/*previously declared as an array of short integers and contains*/
/*the value 1000.  Note that the first argument is number zero.*/

argnum = 0;
optioncode = NSO_MAX_SEND_SIZE;
datalength = 2;
data[0] = 1000;

addopt (opt, argunm, optioncode, datalength, data, &error);

        .
        .

/* perform error checking here */

        .
        .

/*addopt is used again to add the maximum receive size as the*/
/*second argument to the opt parameter.  The maximum receive */
/*size has an option code of 4.  The data parameter contains */
/* the value 1000.                                           */

argnum = 1;
optioncode = NSO_MAX_RECV_SIZE;

addopt (opt, argnum, optioncode, datalength, data, &error);

          .
          .

/* perform error checking here */
/* ipcconnect can now be called using the opt parameter.   */
```

Programming Considerations

The following is a list of the type definitions and passing modes for the `initopt()` call parameters in C, Pascal, and FORTRAN.

Parameter	C	PASCAL	FORTRAN
opt	short int opt[]	array of bytes by reference	array of integers by reference
optnumarguments	short int optnumarguments	int16* by value	integer by value
error	short int *error	int16* by reference	integer by reference

*int16 is a user-defined Pascal type for a 16-bit integer.

ipcconnect()

Requests a connection to another process.

Syntax

```
ipcconnect(calldesc,destdesc,flags,opt,vcdesc,result)
```

Parameters

calldesc — Call socket descriptor. Refers to a call socket owned by the calling process. This parameter is optional; -1 or NS_NULL-DESC is allowed.

destdesc — Destination descriptor. Refers to a structure that indicates the location of the destination call socket (this is the call socket to which the connection request will be sent). A destination descriptor can be obtained by calling ipclookup() or ipcdest().

flags — This parameter must be 0 or a pointer to 0. All other values are reserved for future use. (Refer to "Flags Parameter" for more information on the structure of this parameter.)

opt — Refer to "Opt Parameter" for information on the structure and use of this parameter. The following options are defined for this call:

- *optioncode* = NSO_MAX_SEND_SIZE (code 3), *datalength* = 2. A two-byte integer that specifies the maximum number of bytes expected to be sent with a single ipcsend() call on this connection. 32,767 bytes can be specified.

Default: The TCP default is 100 bytes. If this option is not specified, `ipcsend()` will return an error if a call attempts to send greater than 100 bytes.

- *optioncode* = NSO_MAX_RECV_SIZE (code 4), datalength = 2. A two-byte integer that specifies the maximum number of bytes expected to be received with a single `ipcrecv()` call on this connection. 32,767 bytes can be specified. **Default:** The TCP default is 100 bytes. If this option is not specified, `ipcrecv()` will return an error if a call attempts to receive greater than 100 bytes.

vcdesc — VC socket descriptor. Refers to a VC socket that is the endpoint of the virtual circuit connection at this node. May be used in subsequent NetIPC calls to reference the connection.

result — The error code returned; zero or NSR_NO_ERROR if no error.

Description

The `ipcconnect()` call is used to initiate a virtual circuit on which data may be sent and received. `ipcconnect()` reports only whether a virtual circuit has been *initiated*, not whether it was successfully established. A successful return only indicates that a connection request was sent without error. If the connection is successfully initiated, `ipcconnect()` will return a VC socket descriptor in its *vcdesc* parameter. This VC socket descriptor refers to a VC socket that is the endpoint of the virtual circuit at the local node.

Actively establishing a virtual circuit with NetIPC calls is a three-step process:

- First, `ipcconnect()` is called to request a connection on the client.
- Second, `ipcrecvcn()` is called to receive the connection request on the server.
- Third, `ipcrecv()` is called to find out if the virtual circuit connection initiated with `ipcconnect()` can be successfully established by the client.

`ipcconnect()`'s *opt* parameter specifies the maximum number of bytes expected to be sent and received on the connection. The default for both sending and receiving is 100 bytes. This information is passed to the underlying protocol. When TCP is the underlying protocol it will be used to limit the number of bytes which can be queued on a socket.

Note When a process calls `ipcconnect()`, TCP checksumming for the connection that will be established is automatically enabled. TCP checksum is performed in addition to data link checksum.

Programming Considerations

The following is a list of the type definitions and passing modes for the `ipcconnect()` call parameters in C, Pascal, and FORTRAN.

Parameter	C	PASCAL	FORTRAN
calldesc	ns_int_t calldesc	integer by value	integer*4 by value
destdesc	ns_int_t destdesc	integer by value	integer*4 by value
flags	ns_int_t *flags	boolean array by reference	integer*4 by reference
opt	short int opt[]	packed array of bytes by reference	array of 16-bit integers by reference
vcdesc	ns_int_t *vcdesc	integer by reference	integer*4 by reference
result	ns_int_t *result	integer by reference	integer*4 by reference

Cross-System Considerations

Checksumming - When the `ipcconnect()` call is executed on the HP 9000 node, checksumming is enabled for the HP 9000-to-HP 1000 connection, HP 9000-to-HP 3000 connection, or HP 9000-to-PC connection.

Send and Receive sizes - The HP 1000 send and receive size range is 1 to 8,000 bytes; the HP 9000 send and receive size range is 1 to 32,767 bytes; the HP 3000 send and receive size range is 1 to 30,000 bytes; and the PC range is 1 to 65,535 bytes. Although the ranges are different, specify a buffer size within the correct range for the respective system.

ipccontrol()

Performs special operations on sockets.

Syntax

```
ipccontrol(descriptor,request,wrtdata,wlen,readdata,rlen,
   flags,result)
```

Parameters

descriptor — The descriptor that refers to the socket to be manipulated. The descriptor is either the *vcdesc* parameter returned from the ipcconnect or ipcrecvcn calls or the call descriptor returned from ipccreate. If *request* is set to NSC_GET_NODE_NAME, you must specify NS_NULL_DESC or -1 in this parameter.

request — Request code. Defines which operation is to be performed. May be one of the following:

- NSC_NBIO_ENABLE (code 1). Place the socket referenced in the *descriptor* parameter in asynchronous mode. (Refer to "Synchronous and Asynchronous Socket Modes" in the "NetIPC Concepts" chapter for more information on asynchronous I/O.)
- NSC_NBIO_DISABLE (code 2). Place the socket referenced in the *descriptor* parameter in synchronous mode. (Refer to "Synchronous and Asynchronous Socket Modes" in the "NetIPC Concepts" chapter for information on synchronous I/O.)
- NSC_TIMEOUT_RESET (code 3). Change the referenced socket's synchronous time-out. The default time-out

value is 60 seconds. The time-out value is given in tenths of seconds. (For example, a value of 1200 would indicate 120 seconds.) The new time-out value is treated as a 16-bit signed integer and must be placed in the first two bytes of the *wrtdata* parameter. The time-out value must be in the range of zero to 32767. Negative values have no meaning and will result in error. A value of zero sets the time-out to infinity. The time-out will not be reset if the referenced socket is switched to asynchronous mode and then back to synchronous mode.

- NSC_TIMEOUT_GET (code 4). Return the synchronous time-out value for the socket referenced in the *descriptor* parameter. The time-out value is treated as a 16-bit signed integer and is returned in the *readdata* parameter.

- NSC_RECV_THRESH_RESET (code 1000). Change the read threshold of the VC socket referenced in *descriptor* parameter. (Read thresholds are one byte by default.) The *descriptor* parameter must reference a VC socket descriptor. The new read threshold value must be placed in the first two bytes of the *wrtdata* parameter. Refer to "Asynchronous and Synchronous Socket Modes" in the "NetIPC Concepts" chapter for more information on read thresholds.

- NSC_SEND_THRESH_RESET (code 1001). Change the write threshold of the VC socket referenced in the *descriptor* parameter. (Write thresholds are one byte by default.) The *descriptor* parameter must reference a VC socket descriptor. The new write threshold value must be placed in the first two bytes of the *wrtdata* parameter. Refer to "Asynchronous and Synchronous Socket Modes" in the "NetIPC Concepts" chapter for more information on write thresholds.

- NSC_RECV_THRESH_GET (code 1002). Return the current write threshold for the VC socket referenced in the *descriptor* parameter. The *descriptor* parameter must reference a VC socket descriptor. The write threshold is treated as a 16-bit signed integer and is returned in the *readdata* parameter.

- NSC_SEND_THRESH_GET (code 1003). Return the current read threshold for the VC socket referenced in the *descriptor* parameter. The *descriptor* parameter must reference a VC socket descriptor. The read threshold is treated as a 16-bit signed integer and is returned in the *readdata* parameter.

- NSC_GET_NODE_NAME (code 9008). Return the fully-qualified local node name. The node name is returned in the *readdata* parameter.

wrtdata — A data buffer or data vector used to pass time-out and threshold information. (Refer to "Data Parameter" for information on the structure of this parameter.)

wlen — Length in bytes of the *wrtdata* data buffer.

readdata — A data buffer or data vector used to contain any data returned by the call. (Refer to "Data Parameter" for information on the structure of this parameter.)

rlen (input/output) — The length in bytes of the *readdata* data buffer. On output, this parameter will contain the total number of bytes returned to the process.

flags — Refer to "Flags Parameter" for more information on the structure and use of this parameter. This parameter must be zero or a pointer to zero. All other values are reserved for future use.

result — The error code returned; zero or NSR_NO_ERROR if no error.

Description

The `ipccontrol()` call is used to manipulate sockets in special ways. The type of request is specified by placing a certain request code in the *request* parameter. Although all of the request types require the *descriptor*, *request* and *result* parameters, some of the parameters are meaningless for certain requests. If NSC_TIMEOUT_RESET, NSC_RECV_THRESH_RESET or NSC_SEND_THRESH_RESET is specified, the *wrtdata* and *wlen* parameters are used. If NSC_TIMEOUT_GET, NSC_RECV_THRESH_GET, NSC_SEND_THRESH_GET or NSC_GET_NODE_NAME is specified, the *readdata* and *rlen* parameters are used.

Sockets with Multiple Descriptors

Because the `ipccontrol()` requests operate on sockets, all processes that own descriptors to a particular socket will be affected by `ipccontrol()` operations performed on that socket.

For example, one process can change a socket's read or write threshold, synchronous time-out interval or synchronous/asynchronous mode while another process is reading, writing or selecting on that socket. Exactly when the process that is sharing the socket will be affected by these operations cannot be reliably predicted. Reads, writes and selects in progress may complete using either previous, new or a combination of the previous and new values.

Programming Considerations

When using the NSC_TIMEOUT_RESET or NSC_RECV_THRESH_RESET request, you must be sure to place the time-out value or write threshold value in the *first two bytes* of the ***wrtdata*** parameter. The following C program fragment demonstrates how this can be achieved:

```
char wrtdata[128]
*(short*)&wrtdata = 600;
ipccontrol(descriptor,NSC_TIMEOUT_RESET,&wrtdata,2,0,0,&flags,&result
);
```

The following is a list of the type definitions and passing modes for the `ipccontrol()` call parameters in C, Pascal, and FORTRAN.

Parameter	C	PASCAL	FORTRAN
descriptor	ns_int_t descriptor	integer by value	integer*4 by value
request	ns_int_t request	integer by value	integer*4 by value
wrtdata	char *wrtdata	packed array of characters by reference	array of characters by reference
wlen	ns_int_t wlen	integer by value	integer*4 by value
readdata	char *readdata	packed array of characters by reference	array of characters by reference
rlen	ns_int_t *rlen	integer by reference	integer*4 by reference
flags	ns_int_t *flags	boolean array by reference	integer*4 by reference
result	ns_int_t *result	integer by reference	integer*4 by reference

ipccreate()

Creates a call socket.

Syntax

```
ipccreate(socketkind,protocol,flags,opt,calldesc,result)
```

Parameters

socketkind — Indicates the type of socket to be created. Must be NS_CALL or 3 to indicate a call socket. Other values are reserved for future use.

protocol — Indicates the protocol module that the calling process wishes to access. If specified, can be NSP_TCP or 4 to indicate Transmission Control Protocol (TCP). If zero (0) is specified, TCP will always be chosen for call sockets. Other values are reserved for future use.

flags — This parameter is reserved for future use. All bits must be clear (not set). (Refer to "Flags Parameter" section of this chapter for more information on the structure of this parameter.)

opt — Refer to the "Opt Parameter" section of this chapter for more information on the structure and use of this parameter. The following options are defined for this call:

- *optioncode* = NSO_MAX_CONN_REQ_BACK (code 6), *datalength* = 2. A two-byte integer that specifies the maximum number of unreceived connection requests that may be queued to a call socket. If this value is not specified, the default maximum will be used. **Default:** One request. **Range:** 1-20. (NOTE: A queue limit of one may be too few if many processes attempt to initiate

connections to the call socket simultaneously. If this occurs, some connection requests may be automatically rejected.)

- *optioncode* = NSO_PROTOCOL_ADDRESS (code 128), *datalength* = 2. A two-byte integer that specifies a TCP protocol address to be used by the newly-created call socket. If this option is not specified, or if zero is specified, NetIPC will dynamically allocate an address. You must be have superuser capability to request protocol addresses less than 1024. **Recommended Range For Cross-System Applications**: 30767 to 32767.

calldesc — Call socket descriptor. Refers to the newly-created call socket.

result — The returned error code; zero or NSR_NO_ERROR if no error.

Description

ipccreate() is used to create a call socket which will be used by subsequent NetIPC calls to establish a virtual circuit connection between two or more processes. When invoked successfully, ipccreate() returns a call socket descriptor that refers to the newly-created call socket. A process may own a maximum of 2048 descriptors. ipccreate() will return an error if a process attempts to exceed this limit. This limit includes files as well as socket descriptors and destination descriptors. These descriptors may reference sockets and/or files inherited by or otherwise opened by the process.

The NSO_PROTOCOL_ADDRESS option (code 128) can be used to create a call socket with a specific protocol address. If this protocol address is known to the process's peer, the peer process can call ipcdest() with this address (in ipcdest()'s *protoaddr* parameter) so that it may obtain a destination descriptor that references this call socket. Refer to the section titled "Connection Establishment Summary" in the "NetIPC Concepts" chapter for more information.

Programming Considerations

The following is a list of the type definitions and passing modes for the `ipccreate()` call parameters in C, Pascal, and FORTRAN.

Parameter	C	PASCAL	FORTRAN
socketkind	ns_int_t socketkind	integer by value	integer*4 by value
protocol	ns_int_t protocol	integer by value	integer*4 by value
flags	ns_int_t *flags	boolean array by reference	integer*4 by reference
opt	short int opt[]	packed array of bytes by reference	array of integers by reference
calldesc	ns_int_t *calldesc	integer by reference	integer*4 by reference
result	ns_int_t *result	integer by reference	integer*4 by reference

Cross-System Considerations

TCP Protocol Address - The HP 9000, HP 1000, HP 3000, and PC implementations of `ipccreate()` support different ranges of permitted TCP protocol addresses that can be specified in the *`opt`* parameter. All systems should specify a TCP protocol address within the range 30767 to 32767 decimal for cross-system use.

ipcdest()

Creates a destination descriptor.

Syntax

```
ipcdest(socketkind,nodename,nodelen,protocol,protoaddr,
        protolen,flags,opt,destdesc,result)
```

Parameters

socketkind

Defines the type of socket. Must be NS_CALL or 3 to specify a call socket. Other values are reserved for future use.

nodename

The ASCII-coded name that identifies the node where the call socket that uses *protoaddr* resides.

Default: You may omit the organization, organization and domain, or all parts of the node name. When organization or organization and domain are omitted, they will default to the local organization and/or domain. If the *nodelen* parameter is set to zero, *nodename* is ignored and the node name defaults to the local node.

nodelen

The length in bytes of *nodename*. If this parameter is set to zero (0), the *nodename* parameter is ignored and the node name defaults to the local node. A fully-qualified node name may be 50 bytes long.

protocol

Defines the Transport Layer protocol to be used. Must be NSP_TCP or 4 to indicate the Transmission Control Protocol (TCP). Other values are reserved for future use.

protoaddr

A data buffer containing the TCP protocol address specified in the remote process's ipccreate() call.

protolen	The length in bytes of the protocol address. TCP protocol addresses are two bytes long.
flags	This parameter is reserved for future use. Refer to the "Flags Parameter" section of this chapter for information on the structure of this parameter.
opt	No options are defined for this call. Refer to the "Opt Parameter" section of this chapter for information on the structure and use of this parameter.
destdesc	Destination descriptor. Describes the destination call socket. May be used in a subsequent `ipcconnect()` call to establish a connection to another process.
result	The error code returned; zero or `NSR_NO_ERROR` if no error.

Description

The `ipcdest()` call creates a destination descriptor that the calling process can use to establish a connection to another process.

This call is similar in function to `ipclookup()` because it returns a destination descriptor. However, because `ipcdest()` allows you to specify a protocol address, it allows you to obtain a destination descriptor for a call socket with a *particular protocol address*. A call socket can be created with a particular protocol address by using the `ipccreate()` call with the `NSO_PROTOCOL_ADDRESS` option.

The `ipcdest()` call does not verify that the remote endpoint described by the input parameters exists. This evaluation is delayed until the destination descriptor is used in a subsequent `ipcconnect()` call. Refer to Chapter One for more information on using `ipcdest()` to establish a connection.

Programming Considerations

The following is a list of the type definitions and passing modes for the `ipcdest()` call parameters in C, Pascal, and FORTRAN.

Parameter	C	PASCAL	FORTRAN
socketkind	ns_int_t socketkind	integer by value	integer*4 by value
nodename	char *nodename	packed array of characters by reference	array of characters by reference
nodelen	ns_int_t nodelen	integer by value	integer*4 by value
protocol	ns_int_t protocol	integer by value	integer*4 by value
protoaddr	short int *protoaddr	packed array of int16* by reference	array of integers by reference
protolen	ns_int_t protolen	integer by value	integer*4 by value
flags	ns_int_t *flags	boolean array by reference	integer*4 by reference
opt	short int opt[]	packed array of bytes by reference	array of integers by reference
destdesc	ns_int_t *destdesc	integer by reference	integer*4 by reference
result	ns_int_t *result	integer by reference	integer*4 by reference

*int16 is a user-defined Pascal type for a 16-bit integer.

ipcerrmsg()

Provides text describing NetIPC error.

Syntax

```
ipcerrmsg (error, buffer, len, result)
```

Parameters

error	The number of the NetIPC error being described.
buffer	The data buffer that will hold the description.
len	A pointer to the buffer length. On output, it will contain the length of the description.
result	The error code returned; zero or NSR_NO_ERROR

Description

ipcerrmsg copies an error message for a NetIPC error into a supplied buffer. It will copy **len-1** bytes into the buffer. The result will be NULL terminated. If the error number passed in is not a recognized NetIPC error number, then NSR_ERRNUM (value 85) is returned.

ipcerrmsg() Example

```
#define BUFLEN 80
char buffer [BUFLEN];
ipcsend (    ,&result)
if (result != NSR_NO_ERROR)
ipcerrmsg (result,buffer,BUFLEN,result2);
printf ("NetIPC error %od = %os\n", result, buffer);
```

Programming Considerations

The following is a list of the type definitions and passing modes for the `ipcerrmsg()` call parameters in C, Pascal, and FORTRAN.

Parameter	C	PASCAL	FORTRAN
error	short int *error	int16* by reference	integer by reference
buffer	char *buffer	packed array of character by reference	array of characters by reference
len	ns_int_t len	integer by value	integer *4 by value
result	ns_int_t *result	integer by reference	integer *4 by reference

*int16 is a user-defined Pascal type for a 16-bit integer.

ipcerrstr()

Provides text describing NetIPC error numbers.

Syntax

```
ipcerrstr (error)
```

Parameters

error	The error code returned from a NetIPC system call; zero or NSR_NO_ERROR if no error.

Description

ipcerrstr takes a NetIPC error number as input and returns a pointer to a NULL terminated string describing the error.

ipcerrstr() Example

```
ipcsend (    ,&result);
printf ("NetIPC error %d = %s\n", result, ipcerrstr
       (result));
```

Programming Considerations

The following is a list of the type definitions and passing modes for the `ipcerrstr()` call parameters in C, Pascal, and FORTRAN.

Parameter	C	PASCAL	FORTRAN
error	short int *error	int16* by reference	integer by reference

*int16 is a user-defined Pascal type for a 16-bit integer.

ipcgetnodename()

Obtains NetIPC node name of current host.

Syntax

```
ipcgetnodename (nodename,size,result)
```

Parameters

nodename	The pointer to the character array in which the ASCII-coded NetIPC node name is to be returned.
size	The length in bytes of the *nodename* array on input and the length of the returned NetIPC node name on output.
result	The error code returned; zero or NSR_NO_ERROR if no error.

Description

The ipcgetnodename() call is used to obtain the NetIPC node name for the current processor as set by setnodename(2). The name is returned in the array to which the *nodename* parameter points.

ipclookup()

Obtains a destination descriptor.

Syntax

```
ipclookup(socketname,nlen,nodename,nodelen,flags,
          destdesc,protocol,socketkind,result)
```

Parameters

socketname

The name of the call socket to be "looked up." Upper and lower case characters are not considered distinct. Refer to "Socket Name Parameter" for a detailed discussion of naming.

nlen

The length of the socket name in characters. Maximum length is 16 characters.

nodename

The ASCII-coded name that identifies the node where the socket specified in the *socketname* parameter resides. (Refer to "Node Name Parameter" for the syntax of this parameter.)

Default: You may omit the organization, organization and domain, or all parts of the node name. When organization or organization and domain are omitted, they will default to the local organization and/or domain. If *nodelen* is set to zero, *nodename* is ignored and the node name defaults to the local node.

nodelen

The length in bytes of the *nodename* parameter. If zero (0) is specified, NetIPC will search the local node's socket registry. (See the *nodename* parameter above for more information.)

flags	This parameter is reserved for future use. All bits must be clear (not set). (Refer to the "Flags Parameter" section of this chapter for more information on the structure of this parameter.)
destdesc	Destination descriptor. Refers to the descriptor which indicates the location of the named call socket. May be used in subsequent `ipcconnect` and `ipcname` NetIPC calls.
protocol	This parameter is reserved for future use. Zero (0) is always returned to this parameter.
socketkind	Identifies the socket's type. Will always be 3 to indicate a call socket.
result	The error code returned; zero or NSR_NO_ERROR if no error.

Description

The `ipclookup()` call is used to obtain a destination descriptor for a named call socket. When supplied with valid socket and node names, it looks up the call socket in the socket registry at the node specified in the ***nodename*** parameter and returns a destination descriptor that can be used by subsequent NetIPC calls to locate the call socket. A destination descriptor is required by the `ipcconnect` call to provide the information necessary to direct a connection request to the proper node and call socket and thus initiate a connection.

Timing Problems

When a process attempts to look up a socket name in the appropriate socket registry, the name must be there or a NSR_NAME_NOT_FOUND (code 37) error will be returned to the calling process. When two processes are running concurrently, it may be difficult to ensure that a socket name is placed in the socket registry prior to being "looked up" by another process.

In order to avoid a timing problem:

- The process that calls `ipclookup()` can test for a `NSR_NAME_NOT_FOUND` (code 37) error in the call's *`result`* parameter. If this error is returned, the process can try again by entering a loop and repeating the `ipclookup()` call for a specified number of times.
- The process could also call `sleep()` to suspend execution for an interval and then repeat the `ipclookup()` call. (Refer to the *HP-UX Reference Manual* for more information on *sleep(3c)*).
- The process that calls `ipcname()` can name its call socket and then schedule the process that calls `ipclookup()`.

Note On the Series 600/800 only, `ipclookup()` implementations between HP-UX software versions 1.0 and later software versions are incompatible. If you must use NetIPC to communicate between HP-UX software versions 1.0 and later software versions, utilize the `ipcdest()` system call, not `ipclookup()`.

Programming Considerations

The following is a list of the type definitions and passing modes for the `ipclookup()` call parameters in C, Pascal, and FORTRAN.

Parameter	C	PASCAL	FORTRAN
socketname	ns_int_t socketname	packed array of characters by reference	array of characters by reference
nlen	ns_int_t nlen	integer by value	integer*4f by value
nodename	char *nodename	packed array of characters by reference	array of characters by reference
nodelen	ns_int_t nodelen	integer by value	integer*4 by value
flags	ns_int_t *flags	boolean array by reference	integer*4 by reference
destdesc	ns_int_t *destdesc	integer by reference	integer*4 by reference
protocol	ns_int_t *protocol	integer by reference	integer*4 by reference
socketkind	ns_int_t *socketkind	integer by reference	integer*4 by reference
result	ns_int_t *result	integer by reference	integer*4 by reference

ipcname()

Associates a name with a call socket or destination descriptor.

Syntax

```
ipcname(descriptor,socketname,nlen,result)
```

Parameters

descriptor	The descriptor that references the call socket to be named. May be a call socket descriptor or a destination descriptor.
socketname (input/output)	The ASCII-coded name to be associated with the descriptor. Upper and lower case characters are considered equivalent. NetIPC can also return a randomly-generated name in this parameter (see *nlen*). Refer to "Socket Name Parameter" for a detailed discussion of naming.
nlen	The length in characters of *socketname*. Maximum length is 16 characters. If zero is specified, NetIPC will return a random, eight-byte name in the *socketname* parameter. The eight-byte length is not returned in the *nlen* parameter.
result	The error code returned; zero or NSR_NO_ERROR if no error.

Description

ipcname() associates a name with a call socket and adds this information to the local node's socket registry.

The name a process associates with a call socket must be known to its peer process so that the peer process may look up the name with an ipclookup() call. This may be accomplished by hard-coding the name into both processes or by passing the name from one process to another.

The name associated with a call socket can be user-defined or randomly generated by NetIPC and must be unique to your node (i.e., it cannot be simultaneously associated with two descriptors.) For example, if a call socket is assigned the name "Liz" with a call to `ipcname()`, a subsequent call with "Liz" will result in an error. You can ensure that the name you assign to a call socket is unique by using the random name generating feature of `ipcname()`. A name can be reused only if it is not currently being used, but a call socket may be listed under multiple names.

Under most circumstances, `ipcname()` should be called with a name and the call socket descriptor that refers to a call socket owned by the calling process. If the call completes successfully, the call socket will be listed in the socket registry at the local node. `ipclookup()` can be called from another process to "look up" the socket name in the local node's socket registry.

`ipcname()` always enters its listings into the local node's socket registry. `ipclookup()`, by contrast, can look up socket names at both the local node and at a remote node. Because "long distance" look-ups take longer than local look-ups, it may be helpful to use `ipcname()` to name destination descriptors. When a process names a destination descriptor, the name of the descriptor is placed in the local socket registry (the socket registry at the node where the calling process resides). This allows other processes to look up the name in the local socket registry rather than calling `ipclookup()` to look up the name in a socket registry at a remote node.

Using `ipcname()` to name a destination descriptor is less reliable than looking up the socket name at the remote node because destination descriptors, like telephone numbers, can become outdated. As a precaution, you should periodically refresh locally stored destination descriptors.

Note You cannot use `ipcname()` to name VC sockets.

Programming Considerations

The following is a list of the type definitions and passing modes for the `ipcname()` call parameters in C, Pascal, and FORTRAN.

Parameter	C	PASCAL	FORTRAN
descriptor	ns_int_t descriptor	integer by value	integer*4 by value
socketname	char *socketname	packed array of characters by reference	array of characters by reference
nlen	ns_int_t nlen	integer by value	integer*4 by value
result	ns_int_t *result	integer by reference	integer*4 by reference

ipcnamerase()

Deletes a name associated with a call socket or destination call socket.

Syntax

```
ipcnamerase(socketname,nlen,result)
```

Parameters

socketname	The ASCII-coded name that was previously associated with a call socket descriptor or destination descriptor via `ipcname()`. Upper and lower case characters are considered equivalent. Refer to "Socket Name Parameter" for a detailed description of naming.
nlen	The length in bytes of the specified name. Maximum length is 16 bytes.
result	The error code returned; zero or `NSR_NO_ERROR` if no error.

Description

`ipcnamerase()` can be called to remove listings from the local node's socket registry. Only the owner of a call socket or destination call socket may remove the socket's name from the local socket registry. (Refer to "Socket Ownership" in the "NetIPC Concepts" chapter for the definition of a socket owner.) A process that attempts to erase the name of a socket it does not own will receive an `NSR_NO_OWNERSHIP` (code 38) error.

If a call socket descriptor or destination descriptor is destroyed via `ipcshutdown()`, or if its sole owner terminates, then any listings for it that exist at the local socket registry are automatically purged.

When multiple processes have descriptors for the same socket, the first `ipcnamerase()` call will succeed, but subsequent calls will fail.

Programming Considerations

The following is a list of the type definitions and passing modes for the `ipcnamerase()` call parameters in C, Pascal, and FORTRAN.

Parameter	C	PASCAL	FORTRAN
socketname	char *socketname	packed array of characters by reference	array of characters by reference
nlen	ns_int_t nlen	integer by value	integer*4 by value
result	ns_int_t *result	integer by reference	integer*4 by reference

ipcrecv()

Checks the status of a connection or receives data on an established connection.

Syntax

```
ipcrecv(vcdesc,data,dlen,flags,opt,result)
```

Parameters

vcdesc

VC socket descriptor. Refers to a VC socket that: (1) is the endpoint of a virtual circuit connection that has not yet been established, or (2) is the endpoint of an established virtual circuit on which data will be received.

data

A data buffer that will hold the received data, or a data vector describing the location where the data is to be placed. Refer to "Data Parameter" for information on the structure and use of this parameter.

dlen
(input/output)

If the *data* parameter is a data buffer, *dlen* is the maximum number of bytes you are willing to receive. If the NSF_DATA_WAIT flag is set, the amount of data should be 75% if the maximum amount receivable. If the *data* parameter is a data vector, *dlen* refers to the length of the data vector in bytes. As a return parameter, *dlen* indicates how many bytes were actually received. If ipcrecv() is used to check the status of a connection (not to receive data), the *dlen* parameter is meaningless on input and a value of zero (0) is returned on output.

flags
(input/output)

Refer to the "Flags Parameter" section of this chapter for more information on the structure and use of this parameter. Although *flags* may be set on the first `ipcrecv()` call, they won't take effect until subsequent `ipcrecv()` calls over the established connection. The following bits are defined for this call:

- bit 20 — NSF_DATA_WAIT (input). When this bit is set, `ipcrecv()` will never successfully complete receiving less data than it requested in the *dlen* parameter. If this bit is not set, `ipcrecv()` may complete receiving less data than it requested in *dlen*. Refer to the discussion below for more information on this bit. This bit is only meaningful when `ipcrecv()` is issued against an established connection.

- bit 26 — NSF_MORE_DATA (output). When set, this bit indicates that the data received was not delimited by an end-of-message marker. Since user processes always employ stream mode, this bit will always be set. (The Transmission Control Protocol decides how much data to transmit, but it does not delimit the data transmitted in the form of an individual message.)

- bit 30 — NSF_PREVIEW (input). When set, this bit allows you to preview the data queued on the connection. Data is placed in the *data* parameter but not dequeued from the connection. Because the data is not dequeued, the next `ipcrecv()` call will read the same data. This bit is only valid when `ipcrecv()` is issued against an established connection.

- bit 31 — NSF_VECTORED (input). When set, this bit indicates that the *data* parameter is a data vector and not a data buffer. This bit is only valid when `ipcrecv()` is issued against an established connection.

opt — An array of options and associated information. Refer to "NetIPC Common Parameters" for information on the structure and use of this parameter. The following option is defined for this call:

- optioncode = NSO_DATA_OFFSET (code 8), *datalength = 2*. A two-byte integer that defines a byte offset from the beginning of a data buffer where NetIPC is to begin placing the data. This option is valid only if the *data* parameter is a data buffer and not a data vector.

result — The error code returned; zero or NSR_NO_ERROR if no error.

Description

`ipcrecv()` has two functions:

- Check the status of a connection that was initiated with `ipcconnect()`.
- Receive data on a previously established virtual circuit connection.

Checking the Status of a Connection

When `ipcrecv()` is called to check the status of a connection, a zero returned in the *result* parameter indicates that the call was successful and that a virtual circuit connection has been established. If a non-zero value is returned in the *result* parameter, the call was not successful.

An `ipcrecv()` call can be unsuccessful for the following reasons:

- **NSR_SOCKET_TIMEOUT Error Received.** The synchronous timer expired before a "successful" connection status could be obtained. The connection is still pending and `ipcrecv()` should be called again.

- **NSR_WOULD_BLOCK Error Received.** The VC socket referenced by `ipcrecv()` is in asynchronous mode and the call could not be satisfied. A connection is still pending and `ipcrecv()` should be called again. You can perform an exception select on the referenced socket to determine if a successful status can be obtained prior to calling `ipcrecv()`. (Refer to the discussion of `ipcselect()` later in this chapter for more information.)
- **NSR_SIGNAL_INDICATION Error Received.** A signal indication was received. For more information on signals, refer to the discussion of signals in the "NetIPC Concepts" chapter. Signals are also described in the *HP-UX Reference Manual.*
- **Other Errors.** If `ipcrecv()` was unsuccessful for a reason other than those listed above, the referenced VC socket should be shut down by calling `ipcshutdown()`.

Receiving Data

When `ipcrecv()` is called to receive data queued on a previously established virtual circuit connection, several different alternatives are available:

- **Normal reading.** Data is dequeued from the connection and placed into the user's buffer.
- **Preview reading.** This alternative is specified by setting the `NSF_PREVIEW` bit (bit 30) of the *flags* parameter. When this bit is set, data is placed in the process's buffer, but not dequeued from the connection. Consequently, the next `ipcrecv()` call will read the same data. Because `NSF_PREVIEW` enables a process to determine what a data buffer contains before actually reading it, it is especially useful to set this bit when the receiving process must assemble messages from the byte streams that it receives. For example, if the sending process places the length of its "message" in the first two bytes of its send buffer, the receiving process can use `NSF_PREVIEW` to extract the length information from the data received. When the buffer is received again with a subsequent `ipcrecv()` call, the receiving process can specify this length information in the *dlen* parameter and thus reassemble the "message."
- **Vectored or "scattered" reading.** The calling process may pass a data vector argument that describes one or more locations. Received data will be placed into these locations. This alternative can be used with both the normal and preview reads described above and is specified by setting the `NSF_VECTORED` bit (bit 31) of the *flags* parameter. For vectored reads, an `iovec` structure contains the data vector. An `iovec` structure can be defined in C as:

```
struct iovec {
  char  *iov_base;
  unsigned  iov_len;
};
```

and the normal type for the data argument can be replaced by `struct iovec *data`. Each `iovec` entry specifies the base address and length of an area in memory where the data should be placed.

The `ipcrecv()` call functions differently depending on whether the socket referenced is in synchronous or asynchronous mode, and whether or not the `NSF_DATA_WAIT` bit (bit 20) is set in the *`flags`* parameter.

Caution The NetIPC `NSF_DATA_WAIT` flag can cause a program to block for an extreme period of time (for example, eight minutes for 8 bytes). It is recommended that NetIPC programs implement a loop instead. Refer to the next paragraph for more specific loop information.

A loop such as the following, instead of the NetIPC `NSF_DATA_WAIT` flag, should be implemented to prevent a NetIPC program to block for a long period of time:

```
/* loop to receive 1000 bytes */
char data_array[1000];
char *copy_data_to;
int  bytes_needed = 1000;
int  bytes_received = 0;

/* stuff missing; eg ipccreate or ipcconnect, etc*/
copy_data_to = data_array;
while (bytes_received < bytes_needed) {
        byte_count = bytes_needed - bytes_received;
        /* NOTE: NSF_DATA_WAIT not set */
        ipcrecv(...., start, byte_count,...);
        bytes_received += byte_count;
copy_data_to += byte_count;
}
```

The following paragraphs describe how the ipcrecv() call functions depending on whether or not the socket referenced is in synchronous or asynchronous mode, and whether or not the NSF_DATA_WAIT flag is set. (When a socket is created, it is placed in synchronous mode by default. You can place a socket in asynchronous mode by calling ipccontrol(). Refer to the discussion of ipccontrol() earlier in this chapter for more information.)

Note The "amount requested" by an ipcrecv() call refers to the number of bytes specified by the *dlen* parameter or the amount specified in the data vector if the NSF_VECTORED flag is set.

- **Synchronous I/O, NSF_DATA_WAIT set.** If the socket referenced by ipcrecv() is in synchronous mode and the NSF_DATA_WAIT bit (bit 20) is set, the calling process will block until (1) the amount of data queued on the connection is greater than or equal to the amount requested, (2) the call times out, or (3) a signal is received. If the data queued on the connection is less than *dlen* bytes, ipcrecv() will suspend the calling process and the synchronous timer will be set. If the timer expires before enough data arrives to satisfy the request, the calling process will resume and an NSR_SOCKET_TIMEOUT error (code 59) will be returned indicating that a time-out occurred. (The synchronous time-out can be adjusted by calling ipccontrol(). Refer to the discussion of ipccontrol() for more information.)

- **Synchronous I/O, NSF_DATA_WAIT** not set. If the socket referenced by ipcrecv() is in synchronous mode and the NSF_DATA_WAIT bit (bit 20) is not set, the the calling process will block until (1) *some amount of data* is queued on the connection (the amount of data queued may or may not be the amount requested, and may be as little as one byte), (2) the call times out, (3) a signal is received, or (4) the connection goes down. If no data is queued on the connection within the synchronous time-out period, the calling process will resume and an NSR_SOCKET_TIMEOUT error (code 59) will be returned indicating that a time-out occurred.

- **Asynchronous I/O, NSF_DATA_WAIT set.** If the socket referenced by ipcrecv() is in asynchronous mode and the NSF_DATA_WAIT bit is set, an NSR_WOULD_BLOCK (code 56) error is returned to the calling process if the amount of data queued on the connection is less than the amount requested. The calling process is *not* suspended awaiting the arrival of data. You can perform a read select on the referenced socket by invoking ipcselect(). ipcselect() determines whether or not a socket is readable prior to calling ipcrecv() to receive data. (Refer to the discussion of ipcselect() later in this chapter or more information.)

- **Asynchronous I/O, NSF_DATA_WAIT not set.** If the socket referenced by `ipcrecv()` is in asynchronous mode and the `NSF_DATA_WAIT` bit is not set, as little as one byte of data will satisfy the `ipcrecv()` request. However, if *no* data is queued to the connection, an `NSR_WOULD_BLOCK` error is returned.

For a detailed discussion of asynchronous and synchronous I/O, refer to "Synchronous and Asynchronous Socket Modes" in the "NetIPC Concepts" chapter.

Programming Considerations

The following is a list of the type definitions and passing modes for the `ipcrecv()` call parameters in C, Pascal, and FORTRAN.

Parameter	C	PASCAL	FORTRAN
vcdesc	ns_int_t vcdesc	integer by value	integer*4 by value
data	char *data	packed array of characters by reference	array of characters by reference
dlen	ns_int_t *dlen	integer by reference	integer*4 by reference
flags	ns_int_t *flags	boolean array by reference	integer*4 by reference
opt	short int opt[]	array of bytes by reference	array of integers by reference
result	ns_int_t *result	integer by reference	integer*4 by reference

Cross-System Considerations

Receive size - The HP 1000 receive size range is 1 to 8,000 bytes, the HP 3000 is 1 to 30,000 bytes, and the HP 9000 is 1 to 32,767 bytes. The maximum receive size of the data buffer determines 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 — Socket descriptor. Refers to a call socket owned by the calling process.

vcdesc — VC socket descriptor. Refers to a VC socket that is the endpoint of the newly-established virtual circuit connection.

flags — Refer to "NetIPC Common Parameters" for more information on the structure of this parameter. No flags are defined for this call.

opt — Refer to "NetIPC Common Parameters" for information on the structure and use of this parameter. The following options are defined for this call:

- *optioncode* = NSO_MAX_SEND_SIZE (code 3), *datalength* = 2. A two-byte integer that specifies the maximum number of bytes you expect to send with a single call to ipcsend() on this connection. **Default:** TCP default is 100 bytes. If this option is not specified, ipcsend() will return an error if a call attempts to send greater than 100 bytes.

- *optioncode* = NSO_MAX_RECV_SIZE (code 4), *datalength* = 2. A two-byte integer that specifies the maximum number of bytes you expect to receive with a single call to ipcrecv() on this connection. **Default:** TCP default is 100 bytes. If this option is not specified, ipcrecv() will return an error if a call attempts to receive greater than 100 bytes.

result The error code returned; zero or NSR_NO_ERROR if no error.

Description

When ipcrecvcn() is invoked successfully against a call socket that has queued connection requests, it returns a VC socket descriptor to the calling process. This VC socket descriptor can be used to specify the virtual circuit connection a process intends to send on, receive on, or shut down with subsequent NetIPC calls.

Synchronous vs. Asynchronous I/O

ipcrecvcn() functions differently depending on whether the call socket referenced is in synchronous or asynchronous mode. (When a socket is created, it is placed in synchronous mode by default. You can place a socket in asynchronous mode by calling ipccontrol(). Refer to the discussion of ipccontrol() earlier in this chapter for more information.) The following paragraphs describe these differences:

- **Synchronous I/O.** ipcrecvcn() will block when invoked against a call socket that has no queued connection requests if the socket is in synchronous mode. The calling process will resume execution when a connection request arrives, or after the synchronous time-out interval has expired. An ipcrecvcn() call will not block forever unless the synchronous time-out interval has been set to zero with an ipccontrol() call.

- **Asynchronous I/O.** `ipcrecvcn()` will never block against sockets in asynchronous mode. When `ipcrecvcn()` is invoked against an asynchronous call socket that has no queued connection requests, a `NSR_WOULD_BLOCK` (code 56) is returned to the calling process. When `ipcrecvcn()` is used in this way, the calling process does not wait to receive a connection request. In order to determine when connection requests are present, a process can perform an exception select on the referenced call socket by calling `ipcselect()`. (Refer to the discussion of `ipcselect()` for more information.)

For a detailed discussion of synchronous and asynchronous I/O, refer to "Synchronous and Asynchronous Socket Modes" in the "NetIPC Concepts" chapter.

Note When a process calls `ipcrecvcn()`, TCP checksumming for the connection that will be established is automatically enabled. TCP checksum is performed in addition to data link checksum.

Programming Considerations

The following is a list of the type definitions and passing modes for the `ipcrecvcn()` call parameters in C, Pascal, and FORTRAN.

Parameter	C	PASCAL	FORTRAN
calldesc	ns_int_t calldesc	integer by value	integer*4 by value
vcdesc	ns_int_t vcdesc	integer by reference	integer*4 by reference
flags	ns_int_t *flags	boolean array by reference	integer*4 by reference
opt	short int opt[]	packed array of bytes by reference	array of integers by reference
result	ns_int_t *result	integer by reference	integer*4 by reference

Cross-System Considerations

Checksumming - When the `ipcrecvcn()` call is executed on the HP 9000 node, then checksumming is always enabled for the HP 9000-to-HP 1000 connection, HP 9000-to-3000 connection, or HP 9000-to-PC connection.

Send and Receive sizes - The HP 1000 send and receive size range is 1 to 8,000 bytes; the HP 9000 send and receive size range is 1 to 32,767 bytes; the HP 3000 send and receive size range is 1 to 30,000 bytes; and the PC range is 1 to 65,535 bytes. Although the ranges are different, specify a buffer size within the correct range for the respective system.

ipcselect()

Determines the status of a call socket or VC socket.

Syntax

```
ipcselect(sdbound,readmap,writemap,exceptionmap,timeout,
          result)
```

Parameters

sdbound
(input/output)

Specifies the upper ordinal bound on the range of descriptors specified in the *readmap*, *writemap* and *exceptionmap* parameters. An `ipcselect()` call will be most efficient if this parameter is set to the maximum ordinal value of the sockets specified in these parameters. Because a NetIPC process may have concurrent access to a maximum of 2048 descriptors (descriptors 0 through 2047), *sdbound* may be given a maximum value of 2047. As an output parameter, *sdbound* contains the upper ordinal boundary of all of the descriptors that met the select criteria.

readmap
(input/output)

A bit map indexed with NetIPC socket descriptors. On input, this parameter specifies the socket descriptors to be examined for readability. If zero is passed, no sockets will be examined. On output, *readmap* describes all readable sockets.

writemap (input/output)	A bit map indexed with NetIPC socket descriptors. On input, this parameter specifies the socket descriptors to be examined for writeability. If zero is passed, no sockets will be examined. On output, *writemap* describes all writeable sockets.
exceptionmap (input/output)	A bit map indexed with NetIPC socket descriptors. On input, this parameter specifies the socket descriptors to be examined for exceptions. If zero is passed, no sockets will be examined. On output, *exceptionmap* describes all of the sockets that are exceptions.
timeout	The number of tenths of seconds the calling process will wait for some event to occur which would cause `ipcselect()`'s report to change. This timeout is put into effect only when none of the sockets referenced can immediately satisfy the select criteria (i.e., none are readable, writeable or exceptional). Valid values are zero, -1, or any positive integer. If *timeout* is set to zero, the call will not return until some event occurs. NOTE: Do not set *timeout* to -1 if no bits are set in any of the bit maps as `ipselect()` will block indefinitely.
result	The error code returned; zero or NSR_NO_ERROR if no error.

Description

`ipcselect()` permits a process to detect, and/or wait for, the occurrence of any of several events across any of several sockets. Compared to the telephone system, invoking `ipcselect()` is analogous to performing powerful "switchboard-like" operations because it enables a process to act as a "switchboard operator" by monitoring the sockets, or "telephones," that it owns. A process should call `ipcselect()` with map elements set for descriptors that it owns. If a process attempts to perform a select on a closed or invalid descriptor, an error will be returned. Performing a select on a destination descriptor is meaningless.

`ipcselect()` reports three types of information:

- Whether any of the referenced VC sockets are readable. A VC socket is considered readable if it can immediately satisfy an `ipcrecv()` request for a number of bytes greater than or equal to its read threshold. Each VC socket has an associated read threshold which, when the socket is first created, is set to one byte. This value can be modified by calling `ipccontrol()`. (For more information on setting read thresholds, refer to "Synchronous and Asynchronous Socket Modes" in the "NetIPC Concepts" chapter.) Read selecting on call sockets has no meaning. Although doing so will not produce an error, this practice should be avoided.
- Whether any of the referenced call or VC sockets are writeable. A VC socket is considered writeable if it can immediately accommodate an `ipcsend()` request that involves a number of bytes greater than or equal to the socket's write threshold. Each VC socket has an associated write threshold which, when the socket is first created, is set to one byte. This value may be modified by calling `ipccontrol()`. (For more information on setting write thresholds, refer to "Synchronous and Asynchronous Socket Modes" in the "NetIPC Concepts" chapter.)
- Whether any of the referenced call or VC sockets are exceptional. A VC socket is considered exceptional if it has a problem associated with it (for example, the connection it references was aborted). A call socket is considered exceptional if it has a connection request queued on it or if it can no longer be supported by NetIPC.

When a socket is shared (i.e., more than one process has a descriptor for the same socket), an `ipcsend()` call may return an `NSR_WOULD_BLOCK` error (code 56) even if a previous `ipcselect()` call indicated that the socket was writeable. For example, this would occur if another process (with a descriptor for the same socket) called `ipcsend()` after the original process called `ipcselect()` and before it called `ipcsend()`.

The following are examples of read selecting, write selecting, and exception selecting using `ipcselect()`.

Examples

Detecting Connection Requests

By setting bits in the *exceptionmap* parameter, a process can determine if incoming connection requests are queued to certain call sockets. Consider the following example: Process A must determine whether certain call sockets have received connection requests. To do this, Process A calls `ipcselect()`with the *exceptionmap* map elements set to correspond to these sockets. Assuming that the time-out interval is long enough (set by the *timeout* parameter), ipcselect() will complete after at least one connection request has arrived and has been queued on one of the sockets specified in *exceptionmap*. When the call completes, only those elements that correspond to sockets that have queued connection requests remain set; the other elements will have been cleared.

Performing a Read Select

By setting elements in the *readmap* parameter, a process can determine whether certain VC sockets are readable. Consider the following example: Process A must determine which of its VC sockets have data queued to them. To do this Process A performs a read select on those sockets by setting elements in the *readmap* parameter to correspond with the desired VC sockets. Upon completion of the call, only the elements that represent readable sockets will remain set; the other elements will have been cleared. Process A can call `ipcselect()` with a zero-length time-out to determine the status of a socket immediately, or with a non-zero timeout if it is willing to wait for some data to arrive.

Performing a Write Select

By setting bits in the *writemap* parameter, a process can determine whether certain VC sockets are writeable. Consider the following example: Process A must determine which of its VC sockets can accommodate a new `ipcsend()` request, and which of its call sockets can accommodate a new `ipcconnect();ipcselect()` request. To do this, it can perform a write select on these sockets by setting elements in the *writemap* parameter to correspond with the desired VC and call sockets. Upon completion of the call, only the elements that represent writeable sockets will remain set; the other elements will have been cleared. Process A can call `ipcselect()` with a zero-length timeout to determine the status of a socket immediately, or with a non-zero timeout if it is willing to wait before sending data on the connection.

Exception Selecting

By setting bits in the *exceptionmap* parameter, a process can determine whether certain connections have been aborted. VC sockets that reference aborted connections will always exception select as "true" (their elements will be set when the call completes). Exception selecting on VC sockets can also be useful when the connection associated with the socket is not fully established. Consider the following example: Process B has successfully created a VC socket descriptor via a call to `ipcconnect()`, but will not know whether or not the connection was established until it calls `ipcrecv()`. If Process B calls `ipcrecv()` before the connection is established, or before it becomes known that a connection cannot be established, it will block (if the VC socket is in synchronous mode), or return an `NSR_WOULD_BLOCK` error (if the VC socket is in asynchronous mode). Process B can avoid blocking or polling by performing an exception select on the new VC socket. The socket will select as true if the connection has been established (a call to `ipcrecv()` will be successful) or if there is a problem associated with it (a call to `ipcrecv()` will be unsuccessful.)

Programming Considerations

The following paragraphs explain how the *readmap*, *writemap*, and *exceptionmap* parameters are declared and manipulated in the C, Pascal, and FORTRAN programming languages.

C Programming Language

In the C programming language, the *readmap*, *writemap*, and *exceptionmap* parameters can be declared as integer arrays. For example:

```
int read_map [64];
int write_map [64];
int exception_map [64];
```

This statement defines 2048 bits which can be set to correspond to specific call or VC socket descriptors. The following algorithm can be used to set bits in the array. (The socket descriptor is represented by the variable *vcdesc*.)

```
read_map [vcdesc/32] |= ((unsigned int) 0x80000000 >> (vcdesc %
32));
```

The next algorithm can be used after an `ipcselect()` call completes to check whether or not a certain bit is set:

```
read_map [vcdesc/32] & ((unsigned int) 0x80000000 >> (vcdesc %
32));
```

Pascal Programming Language

In Pascal, the *readmap*, *writemap* and *exceptionmap* parameters can be declared to be type `map_type`. This type is defined as follows:

```
TYPE
  map_type = packed array [0..2047] of boolean;
VAR
  read_map : map_type;
```

To set a bit in any of these parameters to correspond to a specific call socket or VC socket, use the appropriate *calldesc* or *vcdesc* value as a subscript and assign the value TRUE. For example:

```
read_map [vcdesc]     := TRUE;
write_map [calldesc]     := TRUE;
exception_map [vcdesc]     := TRUE;
```

FORTRAN Programming Language

In FORTRAN, the *readmap*, *writemap* and *exceptionmap* parameters may be declared as arrays of 64 32-bit integers (`INTEGER*4`). For example:

```
INTEGER*4 read_map(64), write_map(64), exception_map(64)
```

The first element of the array, `readmap(1)`, contains map bits 0 through 31; the second element of the array, `readmap(2)`, contains bits 32 through 63, etc.

When setting a bit in the array, you must first determine whether your *vcdesc* or *calldesc* parameter is greater than, less than, or equal to 31. If it is less than or equal to 31, you must set a bit in the first element of the array; if it is is greater than 31, you must set a bit in the second element of the array, and so on.

The simplest way to set a bit in one of these parameters is to use the FORTRAN library function `ibset(a,b)`. The *readmap*, *writemap* or *exceptionmap* parameter is passed in the first argument (`a`) and the bit position you want to set is passed in the second argument (`b`).

The `ibset` function assumes that bits are numbered from right to left, with the most significant bit considered to be bit 31 and least significant bit considered to be bit 0. NetIPC calls assume that bits are numbered in the opposite direction (i.e., the most significant bit is 0, the least significant bit is 31). Therefore, to set the proper bit using `ibset`, you must subtract the descriptor value from 31.

In the following example, the *vcdesc* parameter is greater than 31 so the corresponding bit is set in the second element of the *readmap* parameter. Note that the *vcdesc* value must be subtracted from 63 so that the proper bit is set. This maps `ibset`'s bit numbering convention (which is from right to left) into NetIPC's (which is from left to right).

```
read_map = ibset (read_map(2),(63-vcdesc))
```

In the next example, *vcdesc* is equal to 31 so the corresponding bit is set in the first element of the *readmap* parameter. Note that the *vcdesc* value must be subtracted from 31 so that the proper bit is set. Again, this maps `ibset`'s bit numbering convention (which is from right to left) into NetIPC's (which is from left to right).

```
read_map = ibset (read_map(1),(31-vcdesc))
```

The following is a list of the type definitions and passing modes for the `ipcselect()` call parameters in C, Pascal, and FORTRAN.

Parameter	C	PASCAL	FORTRAN
sdbound	ns_int_t *sdbound	integer by reference	integer*4 by reference
readmap	int readmap[64]	packed array of boolean by reference	array of integer*4 by reference
writemap	int writemap[64]	packed array of boolean by reference	array of integer*4 by reference
exceptionmap	int exceptionmap[64]	packed array of boolean by reference	array of integer*4 by reference
timeout	ns_int_t timeout	integer by value	integer*4 by reference
result	ns_int_t *result	integer by reference	integer*4 by reference

ipcsend()

Sends data on a virtual circuit connection.

Syntax

```
ipcsend(vcdesc,data,dlen,flags,opt,result)
```

Parameters

vcdesc — VC socket descriptor. Refers to the VC socket endpoint of the connection through which the data will be sent. A VC socket descriptor can be obtained by calling `ipcconnect()` and `ipcrecvcn()`.

data — A buffer that will hold the data to be sent, or a data vector describing where the data to be sent is located. Refer to "Data Parameter" for more information on the structure and use of this parameter.

dlen — If *data* is a data buffer, *dlen* is the length in bytes of the data in the buffer. If *data* is a data vector, *dlen* is the length in bytes of the data vector.

flags — Refer to "Data Parameter" for more information on the structure and use of this parameter. The following bits are defined for this call:

- `bit26 NSF_MORE_DATA` (input). When this bit is set, TCP may delay sending data. Refer to the "Description" below for more information.
- `bit31 NSF_VECTORED` (input). Indicates that the *data* parameter refers to a data vector and not to a data buffer.

opt An array of options and associated information. Refer to "NetIPC Common Parameters" for more information on the structure and use of this parameter. The following option is defined for this call:

- optioncode = NSO_DATA_OFFSET (code 8), *datalength* = 2). A two-byte integer that indicates a byte offset from the beginning of the data buffer where the data to be sent actually begins. Only valid if the *data* parameter is a data buffer.

result The error code returned; zero or NSR_NO_ERROR if no error.

Description

The ipcsend() call is used to send data on an established connection. The data may be sent as a single contiguous buffer or as a scattered data vector. If the data is vectored, NetIPC will gather all the referenced data before sending it. For vectored writes, an iovec structure contains the data vector. An iovec structure can be defined in C as:

```
struct iovec {
     char          *iov_base;
     unsigned      iov_len;
};
```

and the normal type for the data argument can be replaced by struct iovec *data. Each iovec entry specifies the base address and length of an area in memory where the data should be placed.

If the NSF_MORE_DATA bit (bit 26) of the *flags* parameter is set, the Transmission Control Protocol (TCP) may not immediately transmit the data indicated by the *data* parameter. Instead, it may wait until it has received an amount of data that can be transmitted with the greatest efficiency. Several transmissions of small amounts of data consume more resources than one large transmission. If NSF_MORE_DATA is not set, TCP will attempt to transmit the data immediately, regardless of efficiency considerations. If your process will be sending large amounts of data, HP recommends that you set NSF_MORE_DATA. If NSF_MORE_DATA is set and you submit only a small amount of data (less than a few hundred bytes), then TCP may hold onto the data for a considerable period of time before transmitting it.

Synchronous vs. Asynchronous I/O

`ipcsend()` functions differently depending on whether the VC socket referenced is in synchronous or asynchronous mode. The following paragraphs describe these differences:

- **Synchronous I/O.** Send requests issued against VC sockets in synchronous mode may block. `ipcsend()` will block if it can not immediately obtain the buffer space needed to accommodate the data. The call will resume after the required buffer space becomes available, or if the synchronous timer expires. Timeouts usually occur when the process on the receiving end of the connection stops receiving the data sent to it. (The length of the synchronous time-out interval can be adjusted via `ipccontrol()`. Refer to the discussion of this call for more information.)
- **Asynchronous I/O.** Send requests issued against sockets in asynchronous mode will never block. If the buffer space needed to accommodate the data is not immediately available, a `NSR_WOULD_BLOCK` error (code 56) is returned. After receiving this error, the process can try the call again later, or determine when the socket is writeable by calling `ipcselect()`. (Refer to the discussion of `ipcselect()` for more information on writeable sockets.)

For a detailed discussion of synchronous and asynchronous I/O, refer to "Synchronous and Asynchronous Socket Modes" in the "NetIPC Concepts" chapter.

Programming Considerations

The following is a list of the type definitions and passing modes for the `ipcsend()` call parameters in C, Pascal, and FORTRAN.

Parameter	C	PASCAL	FORTRAN
vcdesc	ns_int_t vcdesc	integer by value	integer*4 by value
data	char *data	packed array of characters by reference	array of characters by reference
dlen	ns_int_t dlen	integer by value	integer*4 by value
flags	ns_int_t *flags	boolean array by reference	integer*4 by reference
opt	short int opt[]	array of bytes by reference	array of integers by reference
result	ns_int_t *result	integer by reference	integer*4 by reference

ipcsetnodename()

Sets the NetIPC node name of the host CPU.

Syntax

```
ipcsetnodename()(nodename,namelen,result)
```

Parameters

nodename	The ASCII-coded name that is assigned to this host. Default: You may omit the organization or the organization and domain and this field will default to the organization and/or domain previously set by `setnodename`.
namelen	The length in bytes of the *nodename* parameter.
result	The error code returned; zero or *NSR_NO_ERROR* if no error.

Description

The `ipcsetnodename()` call sets the NetIPC node name of the host processor to the node name value.

Super-user capability is required to use this call.

ipcshutdown()

Releases a descriptor.

Syntax

```
ipcshutdown(descriptor,flags,opt,result)
```

Parameters

descriptor	The descriptor to be released. May be a call socket descriptor, VC socket descriptor, or destination descriptor.
flags	Must be 0 or NSF_GRACEFUL_RELEASE. If this flag is set, the underlying network protocol will continue to transmit data after the calling process exits. (Refer to "Flags Parameter" for more information on the structure of this parameter.)
opt	Refer to "Opt Parameter" for more information on the structure and use of this parameter. No options are defined for this call. May be 0 or a pointer to an empty NetIPC option buffer.
result	The error code returned; zero or NSR_NO_ERROR if no error.

Description

The ipcshutdown() call is used to release a descriptor. The descriptor referenced may be a call socket descriptor, VC socket descriptor, or destination descriptor. How ipcshutdown() functions depends on which type of descriptor is being used. If the descriptor is a:

- **Call socket descriptor**, the descriptor is released along with any names associated with it. The process that released the call socket descriptor may no longer use it, and all connection requests queued to that descriptor are aborted. The call socket referenced by the descriptor is destroyed along with the descriptor and names only if the descriptor being released is the last descriptor for that socket. If another process, or processes, have a descriptor for the same socket, these duplicate descriptors are not affected. Since system resources are used when a call socket is created, you may want to release your call socket descriptors when they are no longer needed. A call socket descriptor is needed as long as a process is expecting to receive a connection request on that socket. After the connection request is received via `ipcrecvcn()`, and as long as no other connection requests are expected for that call socket descriptor, the descriptor can be released. Similarly, a process that requests a connection can release its call socket descriptors any time after its call to `ipcconnect()`, as long as it is not expecting to receive a connection request on that descriptor. Using `ipcshutdown()` to release a call socket descriptor does not affect any VC sockets.

- **Destination descriptor**, the descriptor is released along with any names associated with it in the local socket registry. The process that released the destination descriptor may no longer use it. The addressing information stored in conjunction with the descriptor is destroyed along with the descriptor only if the descriptor being released is the sole descriptor for that information. If another process, or processes, have a descriptor for the same information, these duplicate descriptors, and any names associated with the descriptors, are not affected. Because destination descriptors also require system resources, you may want to release them when they are no longer needed.

- **VC socket descriptor**, the VC socket descriptor is released and the referenced connection is aborted and is no longer available for sending or receiving data. The VC socket descriptor is released along with the descriptor only if the descriptor being released is the last descriptor for that socket. If another process, or processes, have a descriptor for the same VC socket, these duplicate descriptors are not affected. Because `ipcshutdown()` takes effect very quickly, all of the data that is in transit on the connection, including any data that has already been queued on the destination VC socket, may be destroyed when the connection is shut down. Shutting down a VC socket does not affect any call sockets.

All of the data that is in transit on a VC socket, including any data that has already been queued on the destination VC socket, may be destroyed when the connection is shut down unless the `NSF_GRACEFUL_RELEASE` flag is set. If a process sends important data to its peer process just prior to shutting that process down, it is recommended that the calling process receive a confirmation from the peer process

before calling **ipcshutdown** or exiting, or use the NSF_GRACEFUL_RELEASE flag to ensure that the data was received.

For more information on ipcshutdown(), refer to "Shutting Down a Connection" in the "NetIPC Concepts" chapter.

Programming Considerations

The following is a list of the type definitions and passing modes for the ipcshutdown() call parameters in C, Pascal, and FORTRAN.

Parameter	C	PASCAL	FORTRAN
descriptor	ns_int_t descriptor	integer by value	integer*4 by value
flags	ns_int_t *flags	boolean array by reference	integer*4 by reference
opt	short int opt[]	array of bytes by reference	array of integers by reference
result	ns_int_t *result	integer by reference	integer*4 by reference

Cross-System Considerations

Socket Shut Down - The shutdown procedure for the HP 1000, HP 9000 and HP 3000 processes is identical except for shared sockets on HP 9000 and the "graceful release" flag on the HP 3000 and 9000. 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. If the graceful release flag is set on the HP 3000, the HP 9000 will respond as though it were a normal shutdown request.

optoverhead()

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

Syntax

```
optlength = optoverhead(eventualentries,result)
```

Parameters

optlength	The number of bytes required for the *opt* parameter, not including the data portion of the parameter.
eventualentries	The number of option entries that will be placed in the *opt* parameter.
result	The error code returned; zero or NSR_NO_ERROR if no error.

Description

The optoverhead() call returns the number of bytes needed for the *opt* parameter, excluding the data area.

Programming Considerations

The following is a list of the type definitions and passing modes for the `optoverhead()` call parameters in C, Pascal, and FORTRAN.

Parameter	C	PASCAL	FORTRAN
optlength	short int (returned value)	int16* (returned value)	integer (returned value)
eventualentries	short int (returned value)	int16* by value	integer by value
error	short int *error	int16* by reference	integer by reference

*int16 is a user-defined Pascal type for a 16-bit integer.

readopt()

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

Syntax

```
readopt(opt,argnum,optioncode,datalength,data,result)
```

Parameters

opt	The *opt* parameter to be read. Refer to "NetIPC Common Parameters" for information on the structure and use of this parameter.
argnum	The number of the argument to be obtained. The first argument is number zero.
optioncode	The option code or constant definition (C programs only) associated with the argument. These codes are described in each NetIPC call *opt* parameter description.
datalength (input/output)	The length of the array into which the argument should be read. If the array is not large enough to accommodate the argument 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	An array which will contain the data read from the argument.
result	The error code returned; zero or NSR_NO_ERROR if no error.

Programming Considerations

The following is a list of the type definitions and passing modes for the `readopt()` call parameters in C, Pascal, and FORTRAN.

Parameter	C	PASCAL	FORTRAN
opt	short int opt[]	array of bytes by reference	array of integers by reference
argnum	short int argnum	int16* by value	integer by value
optioncode	short int *optioncode	int16* by reference	integer by reference
datalength	short int *datalength	int16* by reference	integer by reference
data	short int data[]	array of int16* by reference	array of integers by reference
error	short int *error	int16* by reference	integer by reference

*int16 is a user-defined Pascal type for a 16-bit integer.

A

Sample NetIPC Programs

The following are NetIPC program examples. This appendix is organized in two sections: HP 9000 to HP 9000 examples and cross-system NetIPC examples.

HP 9000 to HP 9000 Examples

The following program examples were developed for HP 9000 to HP 9000 communication:

- Example 1: Server Program in C.
- Example 2: Client Program in C.
- Example 3: Server Program in FORTRAN.
- Example 4: Client Program in FORTRAN.

These programs are included in `/usr/netdemo/nsipc`.

Cross-System NetIPC Examples

The following programs were tested with equivalent client/server programs running on the HP 1000 and HP 3000 (MPE-V and MPE/iX). HP 1000 and HP 3000 NetIPC program examples are contained in the NetIPC documentation provided for those systems.

- Example 5: Cross-System Server in C.
- Example 6: Cross-System Client in C.
- Example 7: Cross-System Server in FORTRAN.
- Example 8: Cross-System Client in FORTRAN.
- Example 9: Cross-System Server in PASCAL.
- Example 10: Cross-System Client in PASCAL.

Make File for Sample Programs

```
#!/bin/sh
#                          #
#                                                    #
# Make file for building the sample NetIPC tests.    #
#                                                    #
#                          #
#
#
# Compile NetIPC sample programs(2) in C.
#
all : req_c serv_c req_f serv_f

serv_c : serv.c
        cc -o serv_c serv.c - lnsipc

req_c : req.c
        cc -o req_c req.c - lnsipc

serv_f : serv.f
        fc -o serv_f serv.f - lnsipc

req_f : reg.f
        fc -o req_f req.f - lnsipc
```

Example 1: Server in C

```
/*
 *  This is a server program which executes in background on an
 *  840 machine.  It creates a call socket and names it  ABCDEFGH .
 *  The server waits indefinitely for a connection request. After
 *  a request is received and connection established by the
 *  ipcrecvcn() call, the server forks a child to handle all data
 *  exchange with the requester.  The server then loops back to wait
 *  for new connection requests.
 *
 *  The child receives and logs all messages sent by the requester.
 *  When a shut down message is received, it sends the shut down
 *  message back to the requester.  The next ipcrecv() call will
 *  return a 64 error, signifying that the requester has disconnected.
 *  The child process then calls ipcshutdown() and terminates.
 *
 *  Although the program executes in background, you do not have to
 *  invoke it with a  & .  It automatically puts itself in the background.
 *  This server program is the peer program to the requester
 *  program written in C (Example 2) and FORTRAN (Example 4).
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ns_ipc.h>
#include <sys/time.h>
#include <sys/fcntl.h>
#include <signal.h>
#include <errno.h>
#include <sys/uio.h>
#include <string.h>


#define MSG_SIZE 20

ns_int_t   result;
short  opt[100];
short  opterr;
ns_int_t flags;
char  logbuf[100];
char           *sd_msg =  I want to shut down. ;
char  *logfile =  ./ipc.log ;
int  logf;

main (argc, argv)
int argc;
char **argv;
{
 ns_int_t  vc_desc, call_desc;
 char      *socketname;
 short   short_timeout;
```

```
init_logging ();

/* forks in order to get into the background and detaches
        * process from tty
 */

if (fork())
 exit (0);
setpgrp ();


/* ignore signals */

signal(SIGCLD, SIG_IGN);


/* create call socket */

flags = 0;
initopt (opt, 0, &opterr);  /* initialize opt to zero opt */
ipccreate (NS_CALL, NSP_TCP, &flags, opt, &call_desc, &result);
sprintf (logbuf,  ipccreate: %d\n , result);
log (logbuf);
if (result) goto fatal_error;


/* name the call socket */

socketname= ABCDEFGH ;
ipcname (call_desc, socketname, 8, &result);
sprintf (logbuf,  ipcname %s: %d\n , socketname, result);
log (logbuf);
if (result) goto fatal_error;


/* set call socket timeout to infinite, then wait for
      connection requests */

flags = 0;
short_timeout = 0;
ipccontrol (call_desc, NSC_TIMEOUT_RESET, &short_timeout, 2, 0, 0,
     &flags, &result);
sprintf (logbuf,  ipccontrol NSC_TIMEOUT_RESET: %d\n , result);
log (logbuf);
if (result) goto fatal_error;


while (!result)
{
 flags = 0;
 ipcrecvcn (call_desc, &vc_desc, &flags, opt, &result);
 sprintf (logbuf,  ipcrecvcn: %d\n , result);
 log (logbuf);
 if (result) goto fatal_error;
```

```
  /* fork a child and pass it the newly established
     connection  */

  if (!fork())
  {   /* child */
   rec_data (vc_desc);
   if (!result || result == NSR_REMOTE_ABORT)
     exit (0);
    else
     exit (result);
  }
  /* parent */
  flags = 0;
  ipcshutdown (vc_desc, &flags, opt, &result);
 }

fatal_error: sprintf (logbuf,  fatal_error: %d\n , result);
      log (logbuf);
      exit (result);
}

rec_data (vc_desc)
ns_int_t  vc_desc;
{

 char   buf[MSG_SIZE + 1];
 ns_int_t   msg_len;
 int  shut_down = 0;

 for(;;)
 {
  flags = 0;
  msg_len = MSG_SIZE;
  ipcrecv (vc_desc, buf, &msg_len, &flags, opt, &result);
  sprintf (logbuf,  ipcrecv: %d\n , result);
  log (logbuf);
  if (result) goto return_error;

  if(!strncmp(buf, sd_msg, MSG_SIZE))
  {
   sprintf(logbuf,  Shutdown msg received\n );
   log(logbuf);
   flags = 0;
   msg_len = MSG_SIZE;
   ipcsend (vc_desc, buf, msg_len, &flags, opt, &result);
   sprintf (logbuf,  ipcsend: %d\n , result);
   log (logbuf);
  }
  else
  {
   buf[MSG_SIZE] = (char) 0;
   sprintf(logbuf,  Received: %s\n , buf);
   log(logbuf);
```

```
  }
 }

return_error:   flags = 0;
              ipcshutdown (vc_desc, &flags, opt, &result);
  sprintf (logbuf,  ipcshutdown: %d\n , result);
  log (logbuf);
  return (result);
}

init_logging()
{
 int      flags;

 flags = O_CREAT | O_WRONLY | O_APPEND;
 logf = open (logfile, flags, 0777);
 if (logf < 0) {
  printf ("Couldn't open log file\n");
  exit (-1);
 }
 log ("ipcserver starts\n");
}


log (buf)
char *buf;
{
 int pid = getpid();
 char *ctime(), *time_str;
 struct timeval tv;
 struct timezone tz;
 char  time[25];
 char local_buffer[160];

 gettimeofday (&tv, &tz);
 time_str = ctime (&(tv.tv_sec));
 bcopy (time_str, time, 24);
 time[24] = 0;

 sprintf (local_buffer,  %s: {%d}: %s\n , time, pid, buf);
 write (logf, local_buffer, strlen(local_buffer));
}


bcopy (from_str, to_str, len)
char *from_str;
char  *to_str;
int len;
{
 while (len )
  *to_str++ = *from_str++;
}
```

Example 2: Client in C

```
/*  This program initiates a connection to a remote socket  ABCDEFGH ,
 *  then sends some messages to the server. When this program is ready
 *  to quit, it will send a shut-down message to the server.  After
 *  the server has acknowledged the shut-down message, the shut-down
 *  operation is performed.
 */

#include <stdio.h>
#include <sys/ns_ipc.h>
#include <string.h>

main (argc, argv)
int argc;
char **argv;
{
 ns_int_t  vc_desc, dest_desc;
 ns_int_t  protocol, sock_kind;
 ns_int_t  result;
  short     opt[100];
 short     opterr;
 ns_int_t  flags;
 char   *progname;
 char      *nodename;
 char      *socketname;
 char      buf[64];
 static    char  shut_down[ ] =  I want to shut down. ;
 static    char  msg[ ] =  Message from request ;
 ns_int_t  readlen;
 short   timeout, datalen;

       /* the first argument after the program name indicates the
          node on which  ABCDEFGH  resides  */

 progname = *argv++;
 nodename = *argv;


/* obtain destination descriptor to socket  ABCDEFGH  on
   the remote node that is passed as an argument to the
   program  */

socketname= ABCDEFGH ;
 flags = 0;
 ipclookup (socketname, 8, nodename, strlen(nodename), &flags,
    &dest_desc, &protocol, &sock_kind, &result);
 printf ("{%s} ipclookup: %d\n", progname, result);
 if (result) goto fatal_error;


/* initialize opt structure */
```

```
 initopt (opt, 0, &opterr);  /* initialize opt to zero opt */
 ipcconnect (-1, dest_desc, &flags, opt, &vc_desc, &result);
 printf ("{%s} ipcconnect: %d\n", progname, result);
 if (result) goto fatal_error;


 /* release destination descriptor since it's not needed
    any more */

 ipcshutdown (dest_desc, &flags, opt, &result);
 printf ("{%s} ipcshutdown: %d\n", progname, result);
 if (result) goto fatal_error;

    /* confirm connection */

 flags = 0;
 ipcrecv (vc_desc, buf, &readlen, &flags, opt, &result);
 printf ("{%s} ipcrecv connection: %d\n", progname, result);
 if (result) goto fatal_error;

 flags = 0;
 printf("{%s} Sending requester's message\n", progname);
 ipcsend (vc_desc, msg, 20, &flags, opt, &result);
 if (result) goto fatal_error;

 flags = 0;
 printf("{%s} Sending shutdown message\n", progname);
 ipcsend (vc_desc, shut_down, 20, &flags, opt, &result);
 if (result) goto fatal_error;

 flags = NSF_DATA_WAIT;
 readlen = 20;
 printf("{%s} Waiting to receive shutdown acknowledgement\n", progname);
 ipcrecv (vc_desc, buf, &readlen, &flags, opt, &result);
 printf ("{%s} ipcrecv: %s\n", progname, buf);
 if (result) goto fatal_error;


 if (strncmp (buf, shut_down, 20))
 {
  printf ("strcmp failed %s, %s\n", buf, shut_down);
  goto return_error;
 }

 flags = 0;
 ipcshutdown (vc_desc, &flags, opt, &result);
 printf ("{%s} ipcshutdown: %d\n", progname, result);

return_error: exit(0);

fatal_error: printf ("{%s} fatal error: %d\n", progname, result);
      exit (result);
}
```

Example 3: Server in FORTRAN

```
      PROGRAM serf

C
C       NAME: serf
C     SOURCE: 91790-18237
C      RELOC: 91790-16237
C       PGMR: ZL
C       Modified by KC for the 840

C     This program is the peer process to requester.  It uses sockets in
C     synchronous mode to establish a connection and receive a message
C     from requester.

C     Since FORTRAN values are passed by value by default, the ALIAS
C     statements below are used to indicate which values of the IPC calls
C     should be passed by reference instead.

$ALIAS ipcconect (%val,%val,%ref,%ref,%ref,%ref)
$ALIAS ipccontrol (%val,%val,%ref,%val,%ref,%ref,%ref,%ref)
$ALIAS ipccreate (%val,%val,%ref,%ref,%ref,%ref)
$ALIAS ipcdest (%val,%ref,%val,%val,%ref,%val,%ref,%ref,%ref,%ref)
$ALIAS ipclookup (%ref,%val,%ref,%val,%ref,%ref,%ref,%ref,%ref)
$ALIAS ipcname (%val,%ref,%val,%ref)
$ALIAS ipcnamerase (%ref,%val,%ref)
$ALIAS ipcrecv (%val,%ref,%ref,%ref,%ref,%ref)
$ALIAS ipcrecvcn (%val,%ref,%ref,%ref,%ref)
$ALIAS ipcselect (%ref,%ref,%ref,%ref,%val,%ref)
$ALIAS ipcsend (%val,%ref,%val,%ref,%ref,%ref)
$ALIAS ipcshutdown (%val,%ref,%ref,%ref)
$ALIAS addopt (%ref,%val,%val,%val,%ref,%ref)
$ALIAS initopt (%ref,%val,%ref)
$ALIAS readopt (%ref,%val,%ref,%ref,%ref,%ref)
$ALIAS signal (%val,%val)

      PARAMETER (SIGCLD=1, SIG_IGN=1, NSC_TIMEOUT_RESET=3)
      CHARACTER*20 receive_buffer, shut_down_message
      CHARACTER*8 socket_name

      INTEGER*2 option(2), result, timeout

C     INTEGER SIGCLD, SIG_IGN, request, NSC_TIMEOUT_RESET, fork
      INTEGER request, fork

      INTEGER socket_kind, protocol_kind, call_socket_descriptor,
     >error_return, name_length, VC_socket_descriptor,
     >message_buffer_length, flags_array

      DATA shut_down_message/'I want to shut down.'/

C     Fork Process to get into background and detach from controlling tty
C     and ignore SIGCLD (dealth of child process).
```

```
      if (fork() .NE. 0) STOP
      call setpgrp()
      call signal(SIGCLD, SIG_IGN)

C     The INITOPT call initializes the option parameter used by the
C     IPCCREATE, IPCRECVCN, IPCRECV and IPCSHUTDOWN calls.  By setting
C     the opt_num_arguments parameter to zero, the option parameter is
C     initialized to contain zero entries.  (An example of adding entrie
C     to an option parameter is included in the discussion of ADDOPT in
C     this section.
C
      opt_num_arguments = 0
      CALL INITOPT(option,opt_num_arguments,result)

      here = 1
      IF(result.NE.0) GO TO 99

C     socket_kind is set to 3 and protocol_kind is set to 4 to
C     specify a call socket and the TCP protocol for the following
C     IPCCREATE call.

      socket_kind = 3
      protocol_kind = 4

C     The flags parameter is not used in this program, so flags_array
C     is made a double integer and assigned the value zero to ensure
C     that all the bits are clear.

      flags_array = 0

C     A call socket is created by calling IPCCREATE.  The value returned
C     in the call_socket_descriptor parameter will be used in the following
C     IPCNAME call.

      CALL IPCCREATE(socket_kind,protocol_kind,flags_array,option,
     >call_socket_descriptor,error_return)

      here = 2
      IF(error_return.NE.0) GO TO 99

      flags_array = 0

C     IPCNAME is called to assign a name to the newly-created call
C     socket.  This name is known to the requester.

      socket_name = 'ABCDEFGH'
      name_length = 8

      CALL IPCNAME(call_socket_descriptor,socket_name,name_length,
     >error_return)

      here = 3
      IF(error_return.NE.0) GO TO 99

C     Set call VC socket to infinite.
```

```
      flags_array = 0
      timeout = 0
      request = NSC_TIMEOUT_RESET

      CALL IPCCONTROL(call_socket_descriptor, request, timeout,
     >2, 0, 0, flags_array, error_return)

      here = 4
      IF (error_return .NE. 0) GO TO 99

      flags_array = 0

C     The following IPCRECVCN call will receive the connection request
C     sent by requester and return a VC socket descriptor.  Once this call
C     has completed successfully, you may optionally release the call
C     socket descriptor by calling IPCSHUTDOWN in order to return resources
C     to the system.  Doing so will not affect the newly-created
C     VC socket descriptor.

      CALL IPCRECVCN(call_socket_descriptor,VC_socket_descriptor,
     >flags_array,option,error_return)

      here = 5
      IF(error_return.NE.0) GO TO 99

C     IPCRECV is called to receive a message from requester.

10    flags_array = 0
      message_buffer_length = 20

      CALL IPCRECV(VC_socket_descriptor,receive_buffer,
     >message_buffer_length,flags_array,option,error_return)

C     If error code 64 is received, requester has shut down the connection
C     at its node.  The error processing code at statement 99
C     will call IPCSHUTDOWN to shut down the server's VC socket descriptor.

      here = 6
      IF(error_return.NE.0) GO TO 99

C     The receive buffer is compared to the shut down  message.
C     If the shut down  message  is received, server sends a shut
C     down  message  back to requester so that requester will know that its
C     data has been received.

      IF(receive_buffer .EQ. shut_down_message) THEN
         flags_array = 0
         CALL IPCSEND(VC_socket_descriptor,shut_down_message,
     >        message_buffer_length,flags_array,option,error_return)
         here = 7
         IF(error_return.NE.0) THEN
            GO TO 99
         ELSE
            GO TO 10
```

```
         ENDIF
      ELSE

C     If the shut down  message  was not received, ipc1 will simply rece
C     the data and print it.  It then returns to the previous IPCRECV ca
C     to receive subsequent data until either the shut down  message
C     is received or an error occurs.

C         WRITE(6,'(5A4)')(receive_buffer(index),index = 1,5)
          WRITE(6,*) receive_buffer
          GO TO 10
      ENDIF

99    IF(error_return.EQ.64) THEN
         flags_array = 0
         CALL IPCSHUTDOWN (VC_socket_descriptor,flags_array,option,
     >                     error_return)
         IF(error_return.NE.0) GO TO 99
      ELSE
          WRITE(6,'("error_return error code : _",I4)') error_return
          WRITE(6,'("result error code: _",I4)') result
          WRITE(6,'("Program server at location: _",I4)') here
      ENDIF

100   STOP

      END
```

Example 4: Client in FORTRAN

```
      PROGRAM reqf (location)

C
C       NAME: reqf
C     SOURCE: 91790-18238
C      RELOC: 91790-16238
C       PGMR: ZL
C           Modified by KC to run on a 840
C

C     This program is the peer process to server.  It uses sockets
C     in synchronous mode and sends a message to server.


$ALIAS ipcconnect (%val,%val,%ref,%ref,%ref,%ref)
$ALIAS ipccontrol (%val,%val,%ref,%val,%ref,%ref,%ref,%ref)
$ALIAS ipccreate (%val,%val,%ref,%ref,%ref,%ref)
$ALIAS ipcdest (%val,%ref,%val,%val,%ref,%val,%ref,%ref,%ref,%ref)
$ALIAS ipclookup (%ref,%val,%ref,%val,%ref,%ref,%ref,%ref,%ref)
$ALIAS ipcname (%val,%ref,%val,%ref)
$ALIAS ipcnamerase (%ref,%val,%ref)
$ALIAS ipcrecv (%val,%ref,%ref,%ref,%ref,%ref)
$ALIAS ipcrecvcn (%val,%ref,%ref,%ref,%ref)
$ALIAS ipcselect (%ref,%ref,%ref,%ref,%val,%ref)
$ALIAS ipcsend (%val,%ref,%val,%ref,%ref,%ref)
$ALIAS ipcshutdown (%val,%ref,%ref,%ref)
$ALIAS addopt (%ref,%val,%val,%val,%ref,%ref)
$ALIAS initopt (%ref,%val,%ref)
$ALIAS readopt (%ref,%val,%ref,%ref,%ref,%ref)

      PARAMETER (NSF_DATA_WAIT = 4000B)

      CHARACTER*20 receive_buffer, send_buffer, shut_down_message
      CHARACTER*50 location
      CHARACTER*8 socket_name
      CHARACTER*20 data_buffer

      INTEGER*2 option(2), result, num_arg,
     >opt_num_arguments, counter

      INTEGER*4 socket_kind, protocol_kind, call_socket_descriptor,
     >error_return, name_length, VC_socket_descriptor,
     >message_buffer_length, location_length, data_length,
     >path_report_descriptor, protocol_returned, flags_array

      DATA send_buffer/'Here is the message.'/
      DATA shut_down_message/'I want to shut down.'/

C     INITOPT is called to initialize the option parameter used in the
C     IPCCREATE, IPCLOOKUP, IPCCONNECT, IPCRECV, IPCSEND and
C     IPCSHUTDOWN calls.  By setting opt_num_arguments to zero, the
```

```
C     option parameter is initialized to contain zero entries.
C     (An example of adding options to an option parameter is included
C     in the discussion of ADDOPT in this section.

      DO  i = 1,50
 IF (location(i:i).EQ." " ) THEN
     location_length = i - 1
     GO TO 125
 ENDIF
      END DO
125   CONTINUE
      IF (location_length .EQ. 0) THEN
        WRITE(6, *) 'reqf : usage  reqf nodename'
        STOP
      ENDIF

      opt_num_arguments = 0
      CALL INITOPT(option,opt_num_arguments,result)

      here = 1
      IF(result.NE.0) GO TO 99

C     socket_kind is set to 3 and protocol_kind is set to 4 to specify
C     a call socket and the TCP protocol for the following IPCCREATE
C     call.

      socket_kind = 3
      protocol_kind = 4

C     The flags_array parameter is not used in this program so flags_array
C     is made a double integer and assigned the value zero to ensure tha
C     all the bits are clear.

      flags_array = 0

C     A call socket is created by calling IPCCREATE.  The value returned
C     in the call_socket_descriptor parameter will be used in the following
C     IPCCONNECT call.

      CALL IPCCREATE(socket_kind,protocol_kind,flags_array,option,
     >call_socket_descriptor,error_return)

      here = 2
      IF(error_return.NE.0) GO TO 99

C     The location parameter indicates the node name of the node where
C     ipc2 resides and location_length indicates the length of this
C     name in bytes.  Note that the organization and domain are defaulted.

      socket_name = 'ABCDEFGH'
      name_length = 8

C     IPCLOOKUP searches the socket registry at node1 for server's
C     socket name.  This call returns a path_report_descriptor that is
C     used in the following IPCCONNECT call to request a connection
```

```
C     with server.  Because it is possible for IPCLOOKUP to search for
C     the socket name before server places it in its node's socket
C     registry, server will try to look up the name several times before
C     aborting.

      counter = 0
      flags_array = 0

21    CALL IPCLOOKUP(socket_name,name_length,location,location_length,
     >flags_array,path_report_descriptor,protocol_returned,socket_kind,
     >error_return)

      counter = counter + 1
      here = 4

      IF (error_return.EQ.0) GO TO 28
      IF (error_return.NE.37) GO TO 99
      IF (counter.LE.10) THEN
            GO TO 21
      ELSE
            GO TO 99
      ENDIF

      flags_array = 0

C     The call_socket_descriptor returned by IPCCREATE and the
C     path_report_descriptor returned by IPCLOOKUP are used in
C     IPCCONNECT to request a connection with server.  The
C     VC_socket_descriptor returned by IPCCONNECT is used in subsequent
C     calls to reference the connection.  Once this call has completed
C     successfully, you may optionally release the call socket descriptor
C     by calling IPCSHUTDOWN in order to return resources to the system.
C     Doing so will not affect the newly-created VC socket descriptor.


28    CALL IPCCONNECT(call_socket_descriptor,path_report_descriptor,
     >flags_array,option,VC_socket_descriptor,error_return)

      here = 5
      IF(error_return.NE.0) GO TO 99

      flags_array = 0
      data_length = 20

C     IPCRECV is called to determine if the connection has been
C     established.

      CALL IPCRECV(VC_socket_descriptor,data_buffer,data_length,
     >flags_array,option,error_return)

      here = 6
      IF(error_return.NE.0) GO TO 99

      flags_array = 0
      message_buffer_length = 20
```

```
C     Data is sent to server on the newly established connection.

      CALL IPCSEND(VC_socket_descriptor,send_buffer,
     >message_buffer_length,flags_array,option,error_return)

      here = 7
      IF(error_return.NE.0) GO TO 99

      flags_array = 0

C     After the data is sent, requester initiates the shut down dialogue
C     by sending a shut down  message  to server.

      CALL IPCSEND(VC_socket_descriptor,shut_down_message,
     >message_buffer_length,flags_array,option,error_return)

      here = 8
      IF(error_return.NE.0) GO TO 99

C     After it receives the shut down  message,  server will send its
C     own shut down  message  to requester.  IPCRECV is called to receive
C     this data.

      flags_array = 0

30    CALL IPCRECV(VC_socket_descriptor,receive_buffer,
     >message_buffer_length, flags_array,option,error_return)

          here = 9
          IF(error_return.NE.0) GO TO 99

C     If the receive_buffer contains the shut down  message,  requester will
C     call IPCSHUTDOWN to shut down its VC socket descriptor and termina
C     the connection.

      IF(receive_buffer.EQ.shut_down_message) THEN

      flags_array = 0

      CALL IPCSHUTDOWN (VC_socket_descriptor,flags_array,option,
     >error_return)

          here = 10
          IF(error_return.NE.0) GO TO 99

      GO TO 100

C     Since the only data requester receives from server is a shut down  messag
C     it should never branch to the following ELSE statement.  If this
C     process were the recipient of several IPCSEND calls, it should
C     call IPCRECV again.

      ELSE

C        WRITE(6,'(10A2)')(receive_buffer(index),index=1,10)
         WRITE(6, *) receiver_buffer
```

```
        GO TO 30

      ENDIF

99    WRITE (6, '("result error code: -  ,I4)') result
      WRITE (6, '( error_return error code: _",I4)') error_return
      WRITE (6, '("Program requester at location: _",I4)') here

100   STOP

      END
```

Example 5: Cross-System Server in C

```
/*  NETIPC C-SERVER EXAMPLE
 *
 *  This program similates a local database system which waits
 *  for remote information requests. It will looks for a regular
 *  80 column text file called 'datafile' for information. 'datafile'
 *  must conform with the following format:  The first 20 chars
 *  store a person's name, and the rest of the line stores the
 *  information of that person.
 *
 *  The program creates a call socket at TCP port 31767, then waits
 *  indefinitely for connection requests.  It calls ipcselect() to
 *  test whether the call socket has a connection pending, and calls
 *  ipcrecvcn to accept the connection.  After a connection is
 *  established, the client will send in a person's name, with which
 *  the server will search the database file for information for that
 *  person.  If found, the information will be returned.  This process
 *  continues until the virtual socket becomes exceptional; in which
 *  case, ipcshutdown is called to shutdown that particular socket.
 */

#include <stdio.h>
#include <string.h>
#include <sys/ns_ipc.h>

#define BUFFERLEN20
#define INFOBUFLEN60
#define CALL_SOCKET3
#define INFINITE_SELECT 1
#define MAX_SOCKETS60
#define MAX_BACKLOG5
#define TCP_PORT31767


int   call_sd;
int   call_sd_mask[2];
int   rmap[2], xmap[2];
int   curr_rmap[2], curr_wmap[2], curr_xmap[2];
short   offset;
short   control_value;
ns_int_t  result;
FILE   *datafile;
short   opt[40];
short   opt_data;
short   opterr;
short   timeout;
ns_int_t  flags;
short    opt_num_arguments;
ns_int_t  sbound ;
int   soc_count;
```

```
extern void addopt();
extern void initopt();
extern void ipccontrol();
extern void ipccreate();
extern void ipcrecvcn();
extern void ipcrecv();
extern void ipcselect();
extern void ipcsend();
extern void ipcshutdown();

void Error_Routine();
void HandleNewRequest();
void Initialize_Option();
void ProcessRead();
int  ReadData();
void SetUp();
void ShutdownVC();

/*****************************************************************************/
main()
{
 int  i;

 SetUp();

 /* loop forever to serve clients.  If any new client requests
  * service, the exception map will be set on the call socket.
  * If a client asks for information, the read map will be set
  * on the vc socket for that client.  When the server detects
  * an exceptional condition on an existing vc socket, it means
  * that the corresponding client has shutdown.  In which case,
  * both rmap and xmap are adjusted for the next ipcselect() call.
  *
  * If any other error situation occurs, both the name of the
  * previous ipc call and the error code is printed and the
  * process is terminated by an exit() call.
  */
   curr_rmap[0] = curr_rmap[1] = 0;
   curr_xmap[0] = curr_xmap[1] = 0;
 for(;;)
 {
  for (i = 0; i < 2; i++ )
  {
   curr_rmap[i] = rmap[i];
   curr_xmap[i] = xmap[i];
  }

  timeout = -1;
  sdbound = MAX_SOCKETS;

  ipcselect( &sbound, curr_rmap, curr_wmap, curr_xmap,
      timeout, &result);
  if (result)
  {
```

```
  Error_Routine("ipcselect", result, call_sd);
  }

  /* Check for read condition.
   */
  if ( (curr_rmap[0]) || (curr_rmap[1]) )
  {
   for ( offset = 0; offset < sbound; offset++)
   {
    if (curr_rmap[offset/32] &
        ((unsigned int)0x80000000 >>(offset %32)))
     ProcessRead( offset );
   }
  }

  /* Check for new connection request. The bit in curr_xmap
   * for the call socket is clear, so that the call socket
   * will not be interpreted as a vc shutdown.
   */
  if (curr_xmap[call_sd/32] &
      ((unsigned int)0x80000000 >> (call_sd % 32)) )
  {
   HandleNewRequest();
   curr_xmap[0] &= call_sd_mask[0];
   curr_xmap[1] &= call_sd_mask[1];
  }

  /* Check for vc shutdown.
   */
  if ( (curr_xmap[0]) || (curr_xmap[1]) )
  {
   for ( offset = 0; offset < sbound; offset++)
   {
    if (curr_xmap[offset/32] &
        ((unsigned int)0x80000000 >>(offset %32)))
     ShutdownVC( offset );
   }

  }
 }
}

/*************************************************************************/
void Error_Routine(where, what, sd)
char  *where;
int what;
int sd;
{
 printf("Server: Error occured in %s call.\n", where);
 printf("Server: The error code is: %5d. The local descriptor is:\
 %d \n", what, sd);
 exit();
}
```

```
/*************************************************************************/

void HandleNewRequest()
{
    int vc_sd;


 /* Establish a connection for a new client.  Adjust the xmap
  * and the rmap parameters of the ipcselect() call to reflect
  * the new connection.
  */
    Initialize_Option( opt );
    flags = 0;
 ipcrecvcn( call_sd, &vc_sd, &flags, opt, &result );
 if (result)
 {
  Error_Routine(  ipcrecvcn , result, call_sd );
 }

 /* set rmap and xmap for the new socket for subsequent ipcselect()
        * call.
        */
    rmap[vc_sd/32] |= ((unsigned int) 0x80000000 >> (vc_sd %32));
    xmap[vc_sd/32] |= ((unsigned int) 0x80000000 >> (vc_sd %32));

    /* Set the timeout to infinity with ipccontrol for later calls
  */
    flags = 0;
    control_value = 0;
    ipccontrol( vc_sd, NSC_TIMEOUT_RESET, &control_value, 2, 0,0,
      &flags, &result );
 if (result)
 {
  Error_Routine(  ipccontrol , result, call_sd );
 }

    /*
     * Check if we have reached the maximum number of sockets.
     * If so, disallow any new requests by clearing the exception
     * map for the call socket.
     */
    if (++soc_count >= MAX_SOCKETS )
 {
  xmap[0] = call_sd_mask[0];
  xmap[1] = call_sd_mask[1];
 }
}

/*************************************************************************/

void Initialize_Option()
{
   int opt_num_arguments;
 short opt_err;
```

```
   opt_num_arguments = 0;
   initopt( opt, opt_num_arguments, &opt_err);
 if ( opt_err )
 {
  Error_Routine(  initopt , opt_err, 0);
 }
}

/*****************************************************************************/

void ProcessRead( offset )
short  offset;
{
    int buffer_len;
    char client_buf[BUFFERLEN + 1];
    char data_buf[INFOBUFLEN];
    int vc_sd;


 /* The client with socket discriptor 'offset' has sent in a name.
  * The server will recieve that name and search for the information
  * in the database file. If found, the information will be sent
  * back to the client, otherwise, a 'not found' message will be
  * sent.
  */
    Initialize_Option( opt );
    vc_sd = offset;
    buffer_len = BUFFERLEN;
    flags = 0;
    ipcrecv( vc_sd, client_buf, &buffer_len, &flags, opt, &result );
 if (result)
 {
       Error_Routine(  ipcrecv , result, vc_sd);
 }

 client_buf[BUFFERLEN] = 0;
    if (!ReadData( client_buf, data_buf ))
 {
       printf("Server: %s not in file.\n", client_buf);
  sprintf(data_buf,  SERVER did not find the requested name \
in the datafile.\n );
 }

    buffer_len = INFOBUFLEN;
    flags = 0;
    ipcsend( vc_sd, data_buf, buffer_len, &flags, opt, &result);
 if ( result )
 {
  Error_Routine ("ipcsend", result, vc_sd);
 }
}

/*****************************************************************************/
```

```
int ReadData (client_buf, output_buf)
char *client_buf;
char *output_buf;
{
 chart_buf[80];


 /* Sequentially read the database file until the name is found
  * or EOF is reached. Return 1 if the name is located, 0
  * otherwise.
  */
 rewind( datafile );
 for(;;)
 {
  if (fgets(t_buf, 80, datafile))
  {
   if (!strncmp(client_buf, t_buf, BUFFERLEN))
   {
    strncpy(output_buf, &(t_buf[BUFFERLEN]),
     INFOBUFLEN);
    printf("Server: %s information found.\n",
     client_buf);
    return(1);
   }
  }
  else
  {
   return(0);
  }
 }
}

/*****************************************************************************/

void SetUp()
{
 /* Open the database file for reading.
  */
 if ((datafile = fopen("datafile",  r )) == NULL)
 {
  Error_Routine(  fopen , 0, 0);
 }

    /* Set up the opt array for the two parms we will use
  */
    opt_num_arguments = 2;
    initopt( opt, opt_num_arguments, &opterr);
 if (opterr)
 {
  Error_Routine(  initopt , opterr, call_sd );
 }


 /* Set TCP port address
```

```
 */
opt_data = TCP_PORT;
addopt( opt, 0, NSO_PROTOCOL_ADDRESS, 2, &opt_data, &opterr);
if (opterr)
{
 Error_Routine(  addopt , opterr, call_sd);
}


/* Set maximum number of connection requested can be pend at
 * one time.
 */
opt_data = MAX_BACKLOG;
addopt( opt, 1, NSO_MAX_CONN_REQ_BACK, 2, &opt_data, &opterr);
if (opterr)
{
 Error_Routine(  addopt , opterr, call_sd);
}


/* Create new call socket.
 */
flags = 0;
ipccreate( NS_CALL, NSP_TCP, &flags, opt, &call_sd, &result);
if (result)
{
 Error_Routine(  ipccreate , result, call_sd);
}


/* Set the time out value for subsequent ipcrecvcn() call to
 * infinity. The program will suspend indefinitely on an
 * ipcrecvcn() call.
 */
flags = 0;
control_value = 0;
timeout = 0;
ipccontrol( call_sd, NSC_TIMEOUT_RESET, &timeout, 2, 0, 0,
     &flags, &result);
if (result)
{
 Error_Routine(  ipccontrol , result, call_sd);
}


/* Update soc_count to the number of socket descriptor used so
 * far. Set the xmap bit for the newly created call socket for
 * the next ipcselect() call. Save the one's compliment of xmap
 * for clearing the xmap bit for this call socket later.
 */
soc_count++;
xmap [call_sd/32] |= (((unsigned int) 0x80000000) >> (call_sd % 32));
call_sd_mask[0] = ~xmap[0];
```

```
 call_sd_mask[1] = ~xmap[1];
}

/*****************************************************************************/

void ShutdownVC(offset)
short offset;
{
 int vc_sd;

 flags = 0;
 Initialize_Option( opt );
 vc_sd = offset;
 ipcshutdown( vc_sd, &flags, opt, &result );
soc_count ;
 if ( offset < 32 )
 {
     rmap[offset/32] &= ~((unsigned int) 0x80000000 >> (offset % 32));
     xmap[offset/32] &= ~((unsigned int) 0x80000000 >> (offset % 32));
 }
 xmap[call_sd/32] |= ((unsigned int) 0x80000000 >> (call_sd % 32));
}
```

Example 6: Cross-System Client in C

```
/*  NETIPC C-REQUESTER EXAMPLE
 *
 *  This program initiates a connection to a remote well known
 *  socket at TCP port 31767.  After the connection is established,
 *  The program will prompt the user to input a person's name from
 *  the terminal.  The name will be sent to the server process.  In
 *  return, the server will send back the associate information about
 *  that person if it exists in the database file.  This process
 *  repeats until the user inputs an 'EOT' message. In which case,
 *  the program calls ipcshutdown() to terminate the process.
 *
 */


#include <stdio.h>
#include <string.h>
#include <sys/ns_ipc.h>
#define OPT_SIZE   40
#define NAMELEN    20
#define BUFLEN     80

main ()
{
 ns_int_t  vc_desc, dest_desc;
 ns_int_t  result;
  short     opt[OPT_SIZE];
 short     opterr;
 ns_int_t  flags;
 char      nodename[NAMELEN];
 char   namebuf[BUFLEN];
 char      readbuf[BUFLEN];
 ns_int_t  readlen;
 short   timeout;
 static char  EOTbuf[ ] =  EOT                ;
 short     TCP_port;
 int       i;
 int   shutdown = 0;


 /* Obtain the nodename from the user in which the well know
   * port 31767 is located.
  */

 printf("Client: Enter the remote node name:  );
 gets(nodename);

 initopt (opt, 0, &opterr);
       flags = NSF_DUP_DEST;
 TCP_port = 31767;
 ipcdest (NS_CALL, nodename, strlen(nodename), NSP_TCP,
```

```
  &TCP_port, 2, &flags, opt, &dest_desc, &result);
if (result) goto fatal_error;


/* initialize connection request to server */

ipcconnect (-1, dest_desc, &flags, opt, &vc_desc, &result);
if (result) goto fatal_error;

/* release destination descriptor since it's not needed
   any more */

ipcshutdown (dest_desc, &flags, opt, &result);
if (result) goto fatal_error;

   /* set vc socket timeout to infinite, then confirm connection */

timeout = 0;
ipccontrol (vc_desc, NSC_TIMEOUT_RESET, timeout, 2, readbuf,
     & readlen, &flags, &result);
if (result) goto fatal_error;

ipcrecv (vc_desc, readbuf, &readlen, &flags, opt, &result);
if (result) goto fatal_error;

while(!shutdown)
{
 /* get name from standard input */

 printf ("Client: Enter name for data retrieval:  );
 gets (namebuf);

 for (i = strlen(namebuf); i < NAMELEN; i++)
 {
  namebuf[i] = ' ';
 }

 namebuf[NAMELEN] = (char) 0;
 if (!strncmp(namebuf, EOTbuf, NAMELEN))
 {
  flags = 0;
  ipcshutdown (vc_desc, &flags, opt, &result);
  shutdown = 1;
  exit(0);
 }

 flags = 0;
 ipcsend (vc_desc, namebuf, 20, &flags, opt, &result);
 if (result) goto return_error;

 flags = 0;
 readlen = 60;
 ipcrecv (vc_desc, readbuf, &readlen, &flags, opt, &result);
```

```
  readbuf[readlen] = (char) 0;
  printf ("Client data is: %s\n", readbuf);
  if (result) goto fatal_error;

 }
return_error: exit(0);

fatal_error: printf ("Client:  fatal error: %d\n", result);
     exit (result);
}
```

Example 7: Cross-System Server in FORTRAN

Header File

```
$ALIAS ipcconnect (%val,%val,%ref,%ref,%ref,%ref)
$ALIAS ipccontrol (%val,%val,%ref,%val,%ref,%ref,%ref,%ref)
$ALIAS ipccreate (%val,%val,%ref,%ref,%ref,%ref)
$ALIAS ipcdest (%val,%ref,%val,%val,%ref,%val,%ref,%ref,%ref,%ref)
$ALIAS ipclookup (%ref,%val,%ref,%val,%ref,%ref,%ref,%ref,%ref)
$ALIAS ipcname (%val,%ref,%val,%ref)
$ALIAS ipcnamerase (%ref,%val,%ref)
$ALIAS ipcrecv (%val,%ref,%ref,%ref,%ref,%ref)
$ALIAS ipcrecvcn (%val,%ref,%ref,%ref,%ref)
$ALIAS ipcselect (%ref,%ref,%ref,%ref,%val,%ref)
$ALIAS ipcsend (%val,%ref,%val,%ref,%ref,%ref)
$ALIAS ipcshutdown (%val,%ref,%ref,%ref)
$ALIAS addopt (%ref,%val,%val,%val,%ref,%ref)
$ALIAS initopt (%ref,%val,%ref)
$ALIAS readopt (%ref,%val,%ref,%ref,%ref,%ref)
$ALIAS istrlen = 'strlen' (%ref)
$ALIAS OPEN = 'open' (%ref,%val)
```

COMMONS File

```
      Integer*4 MAX_DESC

      Integer*2 here, active_VC, option(14), result

      Inter*4 call_socket_des, VC_socket_descriptor, flags_array,
     >        error_return,  VC_DES
      Integer*4 sdbound, current_rmap(2), readmap(2), writemap(2),
     >        exceptionmap(2)
      LOGICAL bit_test

      common MAX_DESC

      common here, active_VC, option, result

      common call_socket_des, VC_socket_descriptor, flags_array,
     >        error_return, VC_DES
      common sdbound, current_rmap, readmap, writemap,
     >        exceptionmap

      program server

      Include header

C     This program is the peer process to requester.  It uses sockets in
C     synchronous mode to received a connection and message

      Implicit none
```

```
      Integer*2 SIGCLD, SIG_IGN
      Integer*2 Itime, readdata, backlog, address, opt_num_arguments
      Integer*2 opt_num, opt_code, len, OPEN, oflag
      Integer*2 TCP, MAX_BACKLOG, SYNCH_TIMEOUT, CALL_SOCKET, INFINITE

      Integer*4 socket_kind, protocol_kind, timeout, request, rlen,
     >          wln, fork, filenum, oldnum

      Character filename*16

      Include commons

      PARAMETER (SIGCLD = 1, SIG_IGN = 1)
      PARAMETER (filename = 'datafile' // char(0))
      PARAMETER (oflag = 0)   ! read only

      DATA MAX_BACKLOG/5/, SYNCH_TIMEOUT/3/, TCP/4/, CALL_SOCKET/3/,
     >     INFINITE/0/

      MAX_DESC = 63

      if(fork() .ne. 0) stop
      call setpgrp
      call signal (SIGCLD, SIG_IGN)

C     Open database file 'datafile' needed to service clients.

      filenum = OPEN (filename, oflag)
      call FSET (5, filenum, oldnum)

C     Initialize options to contain 2 parameters.

      opt_num_arguments = 2
      call initopt(option,opt_num_arguments,result)
      here = 1
      IF(result.NE.0) call CLEANUP

C     The Addopt was added to the Server to assign a TCP address
C     during the Ipccreate call.  '128' is the option code equivalent
C     to the predefined constant 'NSO_PROTOCOL_ADDRESS' in C

      opt_num = 0
      opt_code = 128
      len = 2
      Address = 31767
      call addopt(option, opt_num, opt_code, len, Address,result)
      here = 2
      If(result .ne. 0) call CLEANUP

C     Set max backlog of pending connection request to 10

      opt_num = 1
      opt_code = 6
      len = 2
      backlog = MAX_BACKLOG
```

```
      call addopt(option,opt_num, opt_code, len, backlog,result)
      here = 3
      IF(result .NE. 0) call CLEANUP

C     socket_kind is set to 3 and protocol_kind is set to 4 to
C     specify a call Socket and the TCP protocol for the following
C     IPCCREATE call.
C     The flags parameter is not used in this program, so flags_array
C     is made a double integer and assigned the value zero to ensure
C     that all the bits are clear.

      socket_kind = CALL_SOCKET
      protocol_kind = TCP
      flags_array = 0

C     A call Socket is created by calling IPCCREATE.  The value returned
C     in the call_socket_descriptor parameter will be referenced by sub-
C     sequent IPC calls.

      call ipccreate(socket_kind,protocol_kind,flags_array,option,
     >call_socket_des,error_return)
      here = 4
      IF(error_return.NE.0) call CLEANUP

C     IPCCONTROL is used to set synchronous timeout to infinity.

      flags_array = 0
      wln = 2
      request = SYNCH_TIMEOUT
      Itime = INFINITE
      call ipccontrol(call_socket_des,request,Itime,wln,readdata,rlen,
     >flags_array,error_return)
      Here = 5
      IF (error_return .NE. 0) call CLEANUP

C     check call Socket descriptors to check which ones are exceptional
C     (connection request pending) and which ones are readable.

      timeout = -1    ! infinity timeout
      sdbound = MAX_DESC
      writemap(1) = 0
      writemap(2) = 0
      exceptionmap(1) = 0
      exceptionmap(2) = 0
      current_rmap(1) = 0
      current_rmap(2) = 0
      active_VC = 1

C     First time through, set bit mask to recieve connection(s) on
C     newly allocated call socket.

      call bit_set(exceptionmap, call_socket_des)

      DO WHILE (.TRUE.)
        call ipcselect(sdbound, readmap, writemap, exceptionmap,
```

```
     >  timeout, error_return)
        here = 7
        IF (error_return .NE. 0) call CLEANUP
        IF ((readmap(1) .NE. 0) .OR. (readmap(2) .NE. 0)) call get_data
        IF ((exceptionmap(1) .NE. 0) .OR.
     >      (exceptionmap(2) .NE. 0)) THEN
            call process_xmap
        ENDIF
        sdbound = MAX_DESC
        writemap(1) = 0
        writemap(2) = 0
        readmap(1) = current_rmap(1)
        readmap(2) = current_rmap(2)
        exceptionmap(1) = current_rmap(1)
        exceptionmap(2) = current_rmap(2)
        IF (active_VC .LT. MAX_DESC)
     >     call bit_set(exceptionmap, call_socket_des)
       END DO
       END

       subroutine process_xmap

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
C    Subroutine process_xmap receives new connections
C      or shutsdown aborted VC connections.
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

       Include header

       Implicit none

       Integer*2 opt_num_arguments

       Include commons

C      Reset the opt array to 0 so IPCRecvCn and IPCShutdown don't yell at us.

       flags_array = 0
       opt_num_arguments = 0
       CALL INITOPT(option,opt_num_arguments,result)

C      get a new VC_socket_descriptor for the new connection.
C      Set appropriate bit of readmap used later by IPCSELECT.

       IF (bit_test(exceptionmap, call_socket_des)) THEN
            call IPCRECVCN(call_socket_des,VC_socket_descriptor,
     >         flags_array,option,error_return)
            here = 8
            IF (error_return .NE. 0) call CLEANUP
       call bit_set(current_rmap, VC_socket_descriptor)
       END IF
```

```
C     Check to see if VC sockets are exceptional conditions (aborted).
C     If so, shutdown socket and clear update readmap mask for next
C     IPCSELECT call.

      VC_DES = 0
      call bit_clear(exceptionmap, call_socket_des)
      DO WHILE ((exceptionmap(1) .NE. 0) .OR. (exceptionmap(2) .NE. 0))
      IF(bit_test(exceptionmap, VC_DES)) THEN
            flags_array = 0
            call IPCSHUTDOWN(VC_DES, flags_array, option, error_return)
            IF (result .NE. 0) call CLEANUP
            active_VC = active_VC - 1
call bit_clear(exceptionmap, VC_DES)
call bit_clear(current_rmap, VC_DES)
          END IF
          VC_DES = VC_DES + 1
      END DO

      END

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
C   Subroutine recv_data receives data from VC
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

      subroutine get_data

      Include header

      Implicit none

      Integer*2 index

      Integer*4 message_buffer_length

      Character name_requested*20, name*20, eof_message*60,
     >          send_buffer*60

      Include commons

      Data eof_message/'does not appear in datafile'/


C     IPCRECV is called to receive a request from client (requester).
C     First, received the name of item needed.

      VC_DES = 0
      DO WHILE ((readmap(1) .NE. 0) .OR. (readmap(2) .NE. 0))
      IF(bit_test(readmap, VC_DES)) THEN
               flags_array = 0
               message_buffer_length = 20
               VC_socket_descriptor = VC_DES
               CALL IPCRECV(VC_socket_descriptor,name_requested,
     >           message_buffer_length,flags_array,option,error_return)
```

```
            here = 9
            IF (result .NE. 0) call CLEANUP
            call bit_clear(readmap, VC_DES)
         END IF
         VC_DES = VC_DES + 1
      END DO

C     The data file (datafile) is read to locate the corresponding entry.
C     If found, return the information. Otherwise, notify the client.
C
C     ****************************************************************
C     *    An EOF record must exists at the end of 'datafile' to     *
C     *    terminate the sequential search. Otherwise the program    *
C     *    will hang when the name is not found                      *
C     ****************************************************************

      message_buffer_length = 60
      REWIND (5)

      DO WHILE (.TRUE.)
         read(5, '(A20, A60)', end = 98) name, send_buffer
         IF (name_requested .EQ. name) THEN
            flags_array = 0
            CALL IPCSEND(VC_socket_descriptor,send_buffer,
     >        message_buffer_length,flags_array,option,error_return)
            here = 10
            IF(error_return.NE.0) call CLEANUP
            RETURN
         END IF
      END DO
      STOP

98    message_buffer_length = 60
      CALL IPCSEND(VC_socket_descriptor,eof_message,
     >     message_buffer_length,flags_array,option,error_return)
      here = 11
      IF (error_return .NE. 0) call CLEANUP
      RETURN
      END


CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
C      Routine cleanup
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

      subroutine cleanup

      Implicit none

      Include commons
```

```
      WRITE(6,'("error_return error code : _",I4)') error_return
      WRITE(6,'("result error code: _",I4)') result
      WRITE(6,'("Program server at location: _",I4)') here
      STOP
      END

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
C     routine bit_set
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

      subroutine bit_set(map, bit)

      Implicit none

      integer*4 map(2), bit
      integer*4 offset, MAX_DESC

      common MAX_DESC

      offset = 31
      IF (bit .LE. offset) THEN
       map(1) = ibset(map(1), (offset - bit))
      ELSE
       map(2) = ibset(map(2), (MAX_DESC - bit))
      END IF
      RETURN
      END

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
C       routine bit_clear
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

      subroutine bit_clear(map, bit)

      Implicit none

      integer*4 map(2), bit
      integer*4 offset, MAX_DESC
      common MAX_DESC

      offset = 31
      IF (bit .LE. offset) THEN
       map(1) = ibclr(map(1), (offset - bit))
      ELSE
       map(2) = ibclr(map(2), (MAX_DESC  - bit))
      END IF
      RETURN
      END
```

```
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
C         routine bit_test
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

      logical function bit_test(map, bit)

      integer*4 map(2), bit
      integer*4 offset, MAX_DESC

      common MAX_DESC

      offset = 31
      IF (bit .LE. offset) THEN
       IF (btest(map(1), (offset - bit))) THEN
        bit_test = .TRUE.
       ELSE
        bit_test = .FALSE.
       ENDIF
      ELSE
       IF (btest(map(2), (MAX_DESC - bit))) THEN
        bit_test = .TRUE.
       ELSE
        bit_test = .FALSE.
       ENDIF
      ENDIF
      END
```

Example 8: Cross-System Client in FORTRAN

```
      PROGRAM client

C     This program is the peer process to server.  It uses sockets
C     in synchronous mode and sends a message to server.

$ALIAS ipcconnect (%val,%val,%ref,%ref,%ref,%ref)
$ALIAS ipccontrol (%val,%val,%ref,%val,%ref,%ref,%ref,%ref)
$ALIAS ipccreate (%val,%val,%ref,%ref,%ref,%ref)
$ALIAS ipcdest (%val,%ref,%val,%val,%ref,%val,%ref,%ref,%ref,%ref)
$ALIAS ipclookup (%ref,%val,%ref,%val,%ref,%ref,%ref,%ref,%ref)
$ALIAS ipcname (%val,%ref,%val,%ref)
$ALIAS ipcnamerase (%ref,%val,%ref)
$ALIAS ipcrecv (%val,%ref,%ref,%ref,%ref,%ref)
$ALIAS ipcrecvcn (%val,%ref,%ref,%ref,%ref)
$ALIAS ipcselect (%ref,%ref,%ref,%ref,%val,%ref)
$ALIAS ipcsend (%val,%ref,%val,%ref,%ref,%ref)
$ALIAS ipcshutdown (%val,%ref,%ref,%ref)
$ALIAS addopt (%ref,%val,%val,%val,%ref,%ref)
$ALIAS initopt (%ref,%val,%ref)
$ALIAS readopt (%ref,%val,%ref,%ref,%ref,%ref)

      implicit none

      INTEGER*2 option(14), result, opt_num_arguments, counter,
     >          protocol_addr

      INTEGER*4 socket_kind, protocol_kind, call_socket_descriptor,
     >error_return, VC_socket_descriptor, protocol_length,
     >message_buffer_length, location_length, data_length,
     >path_report_descriptor, protocol_returned, flags_array,
     >request, vlen, p1

      INTEGER*2 here, I, J, p2, timeout

      CHARACTER BLANK*1, EOT*3
      CHARACTER receive_buffer*60, send_buffer*20
      CHARACTER location*50, socket_name*8

      DATA  EOT/'EOT'/, BLANK/' '/

C     INITOPT is called to initialize the option parameter used in the
C     IPCCREATE, IPCDEST, IPCCONNECT, IPCRECV, IPCSEND and
C     IPCSHUTDOWN calls.  By setting opt_num_arguments to zero, the
C     option parameter is initialized to contain zero entries.


      opt_num_arguments = 0

      CALL INITOPT(option,opt_num_arguments,result)
      here = 1
      IF(result.NE.0) GO TO 99
```

```
C     socket_kind is set to 3 and protocol_kind is set to 4 to specify
C     a call socket and the TCP protocol for the following IPCCREATE
C     call.
C     The flags_array parameter is not used in this program so flags_array
C     is made a double integer and assigned the value zero to ensure that
C     all the bits are clear.

      socket_kind = 3
      protocol_kind = 4
      flags_array = 0

C     A call socket is created by calling IPCCREATE.  The value returned
C     in the call_socket_descriptor parameter will be used in the following
C     IPCCONNECT call.

      CALL IPCCREATE(socket_kind,protocol_kind,flags_array,option,
     >call_socket_descriptor,error_return)
      here = 2
      IF(error_return.NE.0) GO TO 99

      write(6,*) 'Client: Enter the remote node name:'
      read (5, '(A50)') location
      DO  i = 1,50
IF (location(i:i).EQ." ) THEN
    location_length = i - 1
    GO TO 10
ENDIF
      END DO
10    CONTINUE

      flags_array = 0
      protocol_addr = 31767
      protocol_length = 2

20    call ipcdest(socket_kind, location, location_length,
     >             protocol_kind, protocol_addr, protocol_length,
     >             flags_array, option, path_report_descriptor,
     >             error_return)
      here = 3
      IF (error_return .NE. 0) GO TO 99

      counter = counter + 1
      here = 4

      IF (error_return.EQ.0) GO TO 30
      IF (error_return.NE.37) GO TO 99
      IF (counter.LE.10) THEN
            GO TO 20
      ELSE
            GO TO 99
      ENDIF

      flags_array = 0
```

```
C     The call_socket_descriptor returned by IPCCREATE and the
C     path_report_descriptor returned by IPCDEST are used in
C     IPCCONNECT to request a connection with server.  The
C     VC_socket_descriptor returned by IPCCONNECT is used in subsequent
C     calls to reference the connection.  Once this call has completed
C     successfully, you may optionally release the call socket descriptor
C     by calling IPCSHUTDOWN in order to return resources to the system.
C     Doing so will not affect the newly-created VC socket descriptor.


30    CALL IPCCONNECT(call_socket_descriptor,path_report_descriptor,
     >flags_array,option,VC_socket_descriptor,error_return)

      here = 5
      IF(error_return.NE.0) GO TO 99

      flags_array = 0
      request = 3    ! timeout
      timeout = 0    ! infinite
      vlen = 2


      CALL IPCCONTROL (VC_socket_descriptor, request, timeout, vlen, p1,
     >                 p2, flags_array, error_return)
      here = 9
      IF (error_return .NE. 0) GO TO 99

C     IPCRECV is called to determine if the connection has been
C     established.

      flags_array = 0
      data_length = 60

      CALL IPCRECV(VC_socket_descriptor,receive_buffer,data_length,
     >flags_array,option,error_return)
      here = 6
      IF(error_return.NE.0) GO TO 99

C     Loop forever till user types in 'EOT' in response.
C     Client will then terminate itself and let the networking code
C     clean up which will notify server via the exceptional condition
C     on the appropriate VC socket.

      DO WHILE (.TRUE.)
40        write(6,*) 'Client: Enter name for data retrieval:'
          read (5, '(A20)') send_buffer
          IF (send_buffer .EQ. EOT) STOP
          IF (send_buffer .EQ. BLANK) THEN
              write(6,*) 'Type EOT to terminate.'
              go to 40
          END IF

C     Data is sent to server on the newly established connection.
```

```
      flags_array = 0
      message_buffer_length = 20

      CALL IPCSEND(VC_socket_descriptor,send_buffer,
     >   message_buffer_length,flags_array,option,error_return)
      here = 7
      IF(error_return.NE.0) GO TO 99

C   receives data from server

      message_buffer_length = 60

      CALL IPCRECV(VC_socket_descriptor,receive_buffer,
     >    message_buffer_length, flags_array,option,error_return)
      here = 8
      IF (error_return .NE. 0) go to 99
      write(6,'(A20, $)') send_buffer
      write(6,'(A60)') receive_buffer
    END DO

99  WRITE (6, '("result error code: -  ,I4)') result
    WRITE (6, '( error_return error code: _",I4)') error_return
    WRITE (6, '("Program requester at location: _",I4)') here

100 STOP

    END
```

Example 9: Cross-System Server in PASCAL

```
PROGRAM server( input, output );

{                                   }
{                                                                   }
{   SERVER: IPCSelect Server Sample Program                         }
{                                              Revision: <870610.1338> }
{                                   }
{
{
{                                   }
{  COPYRIGHT (C) 1987 HEWLETT-PACKARD COMPANY.
{  All rights reserved.  No part of this program may be photocopied,
{  reproduced or translated into another programming language without
{  the prior written consent of the Hewlett-Packard Company.
{                                   }
{
{                                   }
{    Name : Server
{  Source : 91790-18###
{   Reloc : 91790-16###
{   Prgmr : <<lms>>
{    Date : <870610.1338>
{                                   }
{
{
{ PURPOSE:
{    To show the operation of the IpcSelect() call.
{
{ REVISION HISTORY
{}
LABEL
   99;

CONST

   BUFFERLEN = 20;
   CALL_SOCKET = 3;
   CHANGE_BACKLOG = 6;
   CHANGE_TIMEOUT = 3;
   FOREVER = TRUE;
   INFINITE_SELECT = -1;
   INFOBUFLEN = 60;
   INT16_LEN = 2;
   LENGTH_OF_DATA = 20;
   MAX_BACKLOG = 5;
   MAX_BUFF_SIZE = 1000;
   MAX_RCV_SIZE = 4;
   MAX_SEND_SIZE = 3;
   MAX_SOCKETS = 32;
   PROTO_ADDR = 128;
```

```
    TCP = 4;
    ZERO = 0;


TYPE

  BitMapType = RECORD
      CASE Integer OF
      1: ( bits     : PACKED ARRAY[0..63] OF Boolean );
      2: ( longint  : Packed Array[1..2] OF Integer );
      3: ( ints     : ARRAY[1..4] OF ShortInt );
      END;

  byte = 0..255;
  byte_array_type = packed array [1..40] of byte;
  buffer_type = packed array [1..BUFFERLEN] of char;
  InfoBufType = packed array [1..INFOBUFLEN] of char;
  name_of_call_array_type = packed array [1..10] of char;
  name_array_type = packed array [1..7] of char;

VAR
  call_name                         : name_of_call_array_type;
  call_sd                           : integer;
  control_value                     : ShortInt;
  curr_rmap                         : BitMapType;
  curr_wmap                         : BitMapType;
  curr_xmap                         : BitMapType;
  dummy_parm                        : Integer;
  dummy_len                         : Integer;
  error_return                      : Integer;
  flags_array                       : integer;
  map_offset                        : ShortInt;
  opt_data                          : ShortInt;
  opt_num_arguments                 : ShortInt;
  option                            : byte_array_type;
  protocol_kind                     : Integer;
  rmap                              : BitMapType;
  sbound                            : Integer;
  short_error                       : ShortInt;
  socket_kind                       : Integer;
  timeout                           : Integer;
  timeout_len                       : Integer;
  vc_count                          : Integer;
  xmap                              : BitMapType;

$TITLE 'IPC Procedures', PAGE $
PROCEDURE ADDOPT
  (VAR opt       : byte_array_type;
       argnum    : ShortInt;
       optcode   : ShortInt;
       data_len  : ShortInt;
   VAR data      : ShortInt;
   VAR error     : ShortInt);
   EXTERNAL;
```

```
PROCEDURE INITOPT
  (VAR opt       : byte_array_type;
       num_args  : ShortInt;
   VAR error     : ShortInt);
   EXTERNAL;

PROCEDURE READOPT
  (VAR opt       : byte_array_type;
       argnum    : ShortInt;
   VAR optcode   : ShortInt;
   VAR data_len  : ShortInt;
   VAR data      : Integer;
   VAR error     : ShortInt);
   EXTERNAL;

PROCEDURE IPCControl
  (    socket    : integer;
       request   : integer;
   VAR wrtdata   : ShortInt;
       wrtlen    : Integer;
   VAR data      : Integer;
   VAR datalen   : Integer;
   VAR flags     : Integer;
   VAR result    : Integer );
   EXTERNAL;

PROCEDURE IPCCREATE
  (    socket    : integer;
       protocol  : integer;
   VAR flags     : integer;
   VAR opt       : byte_array_type;
   VAR csd       : integer;
   VAR result    : integer);
   EXTERNAL;

PROCEDURE IPCNAME
  (    descriptor : integer;
   VAR name       : name_array_type;
       nlen       : integer;
   VAR result     : integer);
   EXTERNAL;

PROCEDURE IPCRECVCN
   (    csd    : integer;
    VAR vcsd   : integer;
    VAR flags  : integer;
    VAR opt    : byte_array_type;
    VAR result : integer);
   EXTERNAL;

PROCEDURE IPCRECV
  (    csd    :integer;
   VAR data   : buffer_type;
   VAR dlen   : integer;
```

```
   VAR flags : integer;
   VAR opt    : byte_array_type;
   VAR result : integer);
   EXTERNAL;

PROCEDURE IPCSelect
  (VAR sbound : Integer;
   VAR rmap   : BitMapType;
   VAR wmap   : BitMapType;
   VAR xmap   : BitMapType;
       timeout: Integer;
   VAR result : Integer );
   EXTERNAL;

PROCEDURE IPCSEND
  (    vcsd   : integer;
   VAR data   : InfoBufType;
       dlen   : integer;
   VAR flags  : integer;
   VAR opt    : byte_array_type;
   VAR result : integer);
   EXTERNAL;

PROCEDURE IPCSHUTDOWN
  (    vcsd   : integer;
   VAR flags  : integer;
   VAR opt    : byte_array_type;
   VAR result : integer);
   EXTERNAL;

$ TITLE 'Internal Procedures', PAGE $

PROCEDURE Error_Routine
  (VAR where  : name_of_call_array_type;
       what   : integer;
       sd     : integer);
   FORWARD;

PROCEDURE HandleNewRequest;
   FORWARD;
   { A new client wants to talk to us, complete the vc establishment }

PROCEDURE Initialize_Option
  (VAR opt_parameter : byte_array_type);
   FORWARD;

PROCEDURE ProcessRead
   (    map_offset   : ShortInt );
   FORWARD;
   { Process the read that is waiting on a particluar vc }

PROCEDURE ReadData
(VAR client_buf    : Buffer_Type;
 VAR output_buf    : InfoBufType );
```

```
   FORWARD;
  { Read the data from the file, prepare for the IPCSend call. }

PROCEDURE SetUp;
   FORWARD;
   { Create a call socket using a well-known address }

PROCEDURE ShutdownVC
   (    map_offset    : ShortInt );
   FORWARD;
   { Shut down a vc that the client no longer needs }


$ TITLE 'Error_Routine', PAGE $
PROCEDURE Error_Routine
  (VAR where  : name_of_call_array_type;
       what   : integer;
       sd     : integer);

   BEGIN    { Error_Routine }

   writeln('Server: Error occurred in ', where,' call.' );
   writeln('Server: The error code is: ', what:5,
                 '.  The local descriptor is: ', sd:4 );

   GOTO 99;

   END;    { Error_Routine }

$ TITLE 'HandleNewRequest', PAGE $
PROCEDURE HandleNewRequest;
{ A new client wants to talk to us, complete the vc establishment }
VAR
   result           : Integer;
   vc_sd            : Integer;

   BEGIN    { HandleNewRequest }

   Initialize_Option( option );

   { Accept the connection for this new vc. }
   IPCRecvCn( call_sd, vc_sd, flags_array, option, result );
   IF result <> ZERO THEN
    BEGIN      { error on ipcrecvcn }
    call_name := 'IPCRECVCN ';
    Error_Routine(call_name,result, vc_sd );
    END;       { error on ipcrecvcn }

   { Increment the total number of active vcs for the server }
   vc_count := vc_count + 1;

   { Now set the read and exception maps for this new vc }
   rmap.bits[vc_sd] := TRUE;
   xmap.bits[vc_sd] := TRUE;
```

```
    { Set the timeout to infinity with IPCControl for later calls }
    flags_array := 0;
    control_value := 0;
    timeout_len := 2;

    IPCControl( vc_sd, CHANGE_TIMEOUT, control_value, timeout_len,
       dummy_parm, dummy_len, flags_array, error_return );

    IF error_return <> ZERO THEN
       BEGIN
       call_name := 'IPCCONTROL';
       Error_Routine( call_name,error_return, vc_sd );
       END;

    {}
    { Check if we have reached the maximum number of sockets.
    { If so, disallow any new requests by clearing the exception
    { map for the call socket.
    {}
    IF vc_count = MAX_SOCKETS -1 THEN
       BEGIN    { reached socket limit }

       xmap.bits[call_sd] := FALSE;
       END;     { reached socket limit }

    END;     { HandleNewRequest }

$ TITLE 'Initialize_Option ', PAGE $

PROCEDURE Initialize_Option
  (VAR opt_parameter : byte_array_type);

VAR
  opt_num_arguments : ShortInt;
  result            : ShortInt;

BEGIN

  opt_num_arguments := 0;
  INITOPT( opt_parameter,opt_num_arguments,result );
  IF result <> ZERO THEN
    BEGIN      { error on initopt }
    call_name := 'INITOPT   ';
    Error_Routine( call_name, result, 0 );
    END;       { error on initopt }

END; {Initialize_Option}

$ TITLE 'ProcessRead', PAGE $
PROCEDURE ProcessRead
   (    map_offset    : ShortInt );
{ Process the read that is waiting on a particluar vc }
VAR
   buffer_len       : Integer;
   client_buf       : Buffer_type;
```

```
    data_buf        : InfoBufType;
    result          : Integer;
    vc_sd           : Integer;

    BEGIN    { ProcessRead }
    { There is a pending read on a vc.  Do an IPCRecv on the vc }
    flags_array := 0;
    Initialize_Option( option );

    vc_sd := map_offset;

    { Get the name this client wants data for }
    buffer_len := BUFFERLEN;

    IPCRecv( vc_sd, client_buf, buffer_len,
       flags_array, option, result );
    IF result <> ZERO THEN
       BEGIN       { error on ipcrecv }
       call_name := 'IPCRECV ';
       Error_Routine( call_name,result,vc_sd );
       END;        { error on ipcrecv }


    { Get the data we need from the file to send to the client }
    ReadData( client_buf, data_buf );
    buffer_len := INFOBUFLEN;

    IPCSend( vc_sd, data_buf, buffer_len, flags_array,
       option, result );
    IF result <> ZERO THEN
       BEGIN       { error on ipcsend }
       call_name := 'IPCSEND ';
       Error_Routine( call_name,result,vc_sd );
       END;        { error on ipcsend }

    END;     { ProcessRead }

$ TITLE 'ReadData', PAGE $
PROCEDURE ReadData
(VAR client_buf     : Buffer_Type;
 VAR output_buf     : InfoBufType );
  { Read the data from the file, prepare for the IPCSend call. }

CONST
    LAST_REC     = 4;

VAR
    current_rec     : ShortInt;
    datafile        : TEXT;
    info_buf        : InfoBufType;
    infofile        : Buffer_Type;
    found           : Boolean;
    name_buf        : Buffer_Type;

    BEGIN    { ReadData }
```

```
{}
{ Open the file named  datafile .  Search until the last record
{ is found, or we match the user name the client wants.
{ If there is a match, retreive the remaining data from the
{ file, and prepare to send it back.
{
{ If there is no match, return  name not found  to the client.
{}

found := FALSE;
current_rec := 1;
infofile := 'datafile';

RESET( datafile, infofile );

WHILE ( NOT found ) AND ( current_rec <= LAST_REC ) DO
   BEGIN    { search the file }

   READLN( datafile, name_buf, info_buf );

   IF client_buf = name_buf THEN
      BEGIN    { found a match }
      {}
      { We found the name the client requested in the file.
      { Set the flag to fall out of the while loop, and
      { get the buffer to be sent to the client.
      {}
      writeln( 'Server: ', client_buf, ' information found.' );

      found := TRUE;
      output_buf := info_buf;

      END;     { found a match }

   { increment to test the next record in the file }
   current_rec := current_rec +1;

   END;     { search the file }


{}
{ We've fallen out of the WHILE loop because there is a match,
{ or we reached the end of the file.  Find out which one it is.
{}

IF NOT found THEN
   BEGIN    { didn't find the requested name }

   {}
   { We didn't find the data in the file.  Put an error
   { message in the data buffer.
   {}
   writeln ('Server: ', client_buf, ' not in file.' );
```

```
      output_buf :=
    'SERVER did not find the requested name in the datafile.    ';

      END;     { didn't find the requested name }


   END;     { ReadData }

$ TITLE 'SetUp', PAGE $
PROCEDURE SetUp;
{ Create a call socket using a well-known address }

   BEGIN    { SetUp }

   { Set up the opt array for the two parms we will use }
   opt_num_arguments := 2;
   InitOpt( option, opt_num_arguments, short_error );
   IF short_error <> ZERO THEN
      BEGIN    { error on initopt }
      call_name := 'InitOpt';
      error_return := short_error;
      Error_Routine( call_name,error_return,call_sd );
      END;     { error on initopt }

   { Now add the option for the well-known address for the IPCCreate Call }
   opt_data := 31767;
   AddOpt( option, 0, PROTO_ADDR, INT16_LEN, opt_data, short_error );
   IF short_error <> ZERO THEN
      BEGIN    { error on AddOpt }
      call_name := 'AddOpt';
      error_return := short_error;
      Error_Routine( call_name,error_return,call_sd );
      END;     { error on AddOpt }


   { Change the backlog queue to the maximum }
   opt_data := MAX_BACKLOG;
   AddOpt( option, 1, CHANGE_BACKLOG, INT16_LEN, opt_data, short_error );
   IF short_error <> ZERO THEN
      BEGIN    { error on AddOpt }
      call_name := 'AddOpt';
      error_return := short_error;
      Error_Routine( call_name,error_return,call_sd );
      END;     { error on AddOpt }

   { Prepare to create a call socket }
   socket_kind := CALL_SOCKET;
   protocol_kind := TCP;

   { clear the flags array }
   flags_array := 0;

   {}
   {A call socket is created by calling IPCCREATE.  The value returned
```

```
    {in the call_sd parameter will be used in the following calls.
    {}

    IPCCREATE( socket_kind, protocol_kind, flags_array, option,
             call_sd, error_return );

    IF error_return <> ZERO THEN
       BEGIN
       call_name := 'IPCCREATE ';
       Error_Routine( call_name,error_return,call_sd );
       END;

{ Set the call_sd timeout to infinity with IPCControl for later calls }
flags_array := 0;
control_value := 0;
timeout_len := 2;

IPCControl( call_sd, CHANGE_TIMEOUT, control_value, timeout_len,
   dummy_parm, dummy_len, flags_array, error_return );

IF error_return <> ZERO THEN
   BEGIN
   call_name := 'IPCCONTROL';
   Error_Routine( call_name,error_return,call_sd );
   END;

   { Now set IPCSelect's bit map for the call socket }
   xmap.bits[call_sd] := TRUE;

   END;     { SetUp }

$ TITLE 'ShutdownVC', PAGE $
PROCEDURE ShutdownVC
   (    map_offset    : ShortInt );
{ Shut down a vc that the client no longer needs }

VAR
   result    :             Integer;
   vc_sd     :             Integer;

   BEGIN    { ShutdownVC }
   {}
   { The client shut down the vc, or it has gone down due to a
   { Networking problem.  Either way, merely accept the shutdown.
   {}
   flags_array := 0;
   Initialize_Option( option );

   vc_sd := map_offset;

   IPCShutdown( vc_sd, flags_array, option, result );
   { Don't worry about errors here, since there isn't much we can do. }

   { Decrement the number of active vcs }
   vc_count := vc_count -1;
```

```
    { Clear the read map and exception map bits for this vc }
    rmap.bits[map_offset] := FALSE;
    xmap.bits[map_offset] := FALSE;

    {}
    { Always set the exception map for the call socket.  That way
    { we'll be sure to re-enable new requests if we were at the
    { limit before this vc was shut down.
    {}
    xmap.bits[call_sd] := TRUE;

    END;      { ShutdownVC }

$TITLE 'Server MAIN', PAGE $
BEGIN { Server }

{ Create a call socket with a well known address for the clients to use. }
SetUp;

{}
{ Loop forever waiting to serve clients.  If any new clients request
{ service, the exception map will be set on the call socket.  If
{ a client asks for information, the read map will be set on the
{ vc socket for that client.  When the client has received the data,
{ it will shut down the vc, and the vc socket will have the exception
{ map set.  Handle each one of these cases in this loop.
{
{ If any other situations occur, exit out of the loop, and let the
{ NS clean up routines de-allocate the sockets for this server.
{}

WHILE FOREVER = TRUE DO
    BEGIN     { Forever Do }

    {}
    { Set the bit masks to check for all the vcs that we own.
    { The rmap & xmap variables are maintained by ProcessNewRequest
    { and ShutdownVC.
    {}
    curr_rmap := rmap;
    curr_xmap := xmap;


    sbound := MAX_SOCKETS;
    timeout := INFINITE_SELECT;

    {}
    { Do an exceptional select on the call socket, and on all vcs
    { we own.     Do a read select on all the vc sockets.
    {}

    IPCSelect( sbound, curr_rmap, curr_wmap, curr_xmap,
       timeout, error_return );
    IF error_return <> ZERO THEN
```

```
   BEGIN       { Select Error }
   call_name := 'IPCSELECT ';
   Error_Routine( call_name,error_return,call_sd );
   END;        { Select Error }

{ See if there are any clients requesting information }
IF ( rmap.longint[1] <> 0 ) OR ( rmap.longint[2] <> 0 ) THEN
   BEGIN    { Process read on VC sockets }

   { We have someone to service.  Find out who it is. }
   FOR map_offset := 1 TO MAX_SOCKETS DO
      BEGIN    { check all vcs }

      IF curr_rmap.bits[map_offset] = TRUE THEN
         BEGIN    { have read on a vc }

         {}
         { We know the client who needs service,
         { Do an IPCRecv, get the necessary data,
         { and do an IPCSend to send it back.
         {}
         ProcessRead( map_offset );

         END;     { have read on a vc }
      END;     { check all vcs }
   END;     { Process read on VC sockets }

{ See if any clients have sent a message to the call socket }
IF curr_xmap.bits[call_sd] = TRUE THEN
   BEGIN    { new request on the call socket }

   {}
   { We have a new client, go do an IPCRecvCn, and set the
   { bit masks to accept reads and exceptions on the new vc.
   {}
   HandleNewRequest;


   { Clear the call socket xmap bit to simplify the test for the vcs }
   curr_xmap.bits[call_sd] := FALSE;

   END;     { new request on the call socket }

{}
{ If we get an exception on a vc socket, shut it down. The client
{ knows to shut down a socket once it has received the data it needs.
{}
IF ( curr_xmap.longint[1] <> 0 ) OR ( curr_xmap.longint[2] <> 0 ) THEN
   BEGIN    { check for errors on vc sockets }

   { One vc had an exception, find out which one }
   FOR map_offset := 1 TO MAX_SOCKETS DO
      BEGIN       { check all vcs }
```

```
        IF curr_xmap.bits[map_offset] = TRUE THEN
           BEGIN    { shut down the vc }

           {}
           { Do an IPCShutdown on the vc, and clear
           { its bit in both the read and exception maps.
           {}

           ShutdownVC( map_offset );

           END;     { shut down the vc }
        END;        { check all vcs }
     END;     { check for errors on vc sockets }

  END;     { Forever Do }

99:

{}
{ We have some problem, the NS cleanup routine will shut down
{ All the sockets we own once the program has terminated.
{}

END.  { Server }
```

Example 10: Cross-System Client in PASCAL

```
PROGRAM Client( input, output );

{                                   }
{                                                                   }
{   Client: IPCSelect Client Sample Program                         }
{                                              Revision: <870610.1327> }
{                                   }
{
{
{                                   }
{  COPYRIGHT (C) 1987 HEWLETT-PACKARD COMPANY.
{  All rights reserved.  No part of this program may be photocopied,
{  reproduced or translated into another programming language without
{  the prior written consent of the Hewlett-Packard Company.
{                                   }
{
{                                   }
{   Reloc : 91790-16###
{   Prgmr : <<lms>>
{    Date : <870610.1327>
{                                   }
{
{
{ PURPOSE:
{    To show the operation of the IpcSelect() call.
{
{ REVISION HISTORY
{}
LABEL
   89,
   99;

CONST

   BUFFERLEN = 20;
   CALL_SOCKET = 3;
   CHANGE_TIMEOUT = 3;
   FOREVER = TRUE;
   INFINITE_SELECT = -1;
   INFOBUFLEN = 60;
   INT16_LEN = 2;
   LENGTH_OF_DATA = 20;
   MAX_BUFF_SIZE = 1000;
   MAX_RCV_SIZE = 4;
   MAX_SEND_SIZE = 3;
   MAX_SOCKETS = 32;
   INTEGER_LEN = 2;
   TCP = 4;
   ZERO = 0;
```

```
TYPE

  BitMapType = RECORD
      CASE Integer OF
      1: ( bits     : PACKED ARRAY[1..32] OF Boolean );
      2: ( longint  : Integer );
      3: ( ints     : ARRAY[1..2] OF ShortInt );
      END;

  byte = 0..255;
  byte_array_type = packed array [1..8] of byte;
  buffer_type = packed array [1..BUFFERLEN] of char;
  InfoBufType = packed array [1..INFOBUFLEN] of char;
  name_of_call_array_type = packed array [1..10] of char;
  name_array_type = packed array [1..7] of char;

VAR
  buffer_len                    : Integer;
  call_name                     : name_of_call_array_type;
  call_sd                       : integer;
  control_value                 : ShortInt;
  data_buf                      : InfoBufType;
  dummy_len                     : Integer;
  dummy_parm                    : Integer;
  error_return                  : Integer;
  flags_array                   : integer;
  node_name                     : Buffer_Type;
  node_name_len                 : Integer;
  opt_data                      : ShortInt;
  opt_num_arguments             : ShortInt;
  option                        : byte_array_type;
  proto_addr                    : ShortInt;
  protocol_kind                 : Integer;
  req_name_len                  : Integer;
  requested_name                : Buffer_Type;
  short_error                   : ShortInt;
  socket_kind                   : Integer;
  temp_position                 : ShortInt;
  timeout                       : Integer;
  timeout_len                   : Integer;
  vc_sd                         : Integer;

$TITLE 'IPC Procedures', PAGE $
PROCEDURE ADDOPT
  (VAR opt       : byte_array_type;
       argnum    : ShortInt;
       optcode   : ShortInt;
       data_len  : ShortInt;
   VAR data      : ShortInt;
   VAR error     : ShortInt);
   EXTERNAL;

PROCEDURE INITOPT
  (VAR opt       : byte_array_type;
```

```
      num_args  : ShortInt;
   VAR error    : ShortInt);
   EXTERNAL;

PROCEDURE IPCConnect
  (    call_sd   : Integer;
       pathdesc  : Integer;
   VAR flags     : Integer;
   VAR opt       : Byte_array_type;
   VAR vc_sd     : Integer;
   VAR error     : Integer);
   EXTERNAL;

PROCEDURE IPCControl
  (    socket    : integer;
       request   : integer;
   VAR wrtdata   : ShortInt;
       wrtlen    : Integer;
   VAR data      : Integer;
   VAR datalen   : Integer;
   VAR flags     : Integer;
   VAR result    : Integer );
   EXTERNAL;

PROCEDURE IPCCREATE
  (    socket    : integer;
       protocol  : integer;
   VAR flags     : integer;
   VAR opt       : byte_array_type;
   VAR csd       : integer;
   VAR result    : integer);
   EXTERNAL;

PROCEDURE IPCNAME
  (    descriptor : integer;
   VAR name       : name_array_type;
       nlen       : integer;
   VAR result     : integer);
   EXTERNAL;

PROCEDURE IPCDEST
   (    sock_kind : Integer;
    VAR node_name : Buffer_Type;
        name_len  : Integer;
        protocol  : Integer;
    VAR protoaddr : ShortInt;
        proto_len : Integer;
    VAR flags     : integer;
    VAR opt       : byte_array_type;
    VAR pathdesc  : Integer;
    VAR result    : Integer);
   EXTERNAL;
```

```
PROCEDURE IPCRECVCN
   (    csd    : integer;
    VAR vcsd   : integer;
    VAR flags  : integer;
    VAR opt    : byte_array_type;
    VAR result : integer);
   EXTERNAL;

PROCEDURE IPCRECV
  (    csd    :integer;
   VAR data   : InfoBufType;
   VAR dlen   : integer;
   VAR flags  : integer;
   VAR opt    : byte_array_type;
   VAR result : integer);
   EXTERNAL;

PROCEDURE IPCSelect
  (VAR sbound : Integer;
   VAR rmap   : BitMapType;
   VAR wmap   : BitMapType;
   VAR xmap   : BitMapType;
       timeout: Integer;
   VAR result : Integer );
   EXTERNAL;

PROCEDURE IPCSEND
  (    vcsd   : integer;
   VAR data   : buffer_type;
       dlen   : integer;
   VAR flags  : integer;
   VAR opt    : byte_array_type;
   VAR result : integer);
   EXTERNAL;

PROCEDURE IPCSHUTDOWN
  (    vcsd   : integer;
   VAR flags  : integer;
   VAR opt    : byte_array_type;
   VAR result : integer);
   EXTERNAL;

$ TITLE 'Internal Procedures', PAGE $

PROCEDURE GetLen
(VAR  buffer      :   Buffer_Type;
 VAR  current_pos :   ShortInt;
 VAR  length      :   Integer );
   FORWARD;
   { Get the length of a string.  Return the next postion }

PROCEDURE Error_Routine
  (VAR where  : name_of_call_array_type;
       what   : integer;
```

```
        sd      : integer);
    FORWARD;

PROCEDURE Initialize_Option
  (VAR opt_parameter : byte_array_type);
    FORWARD;

PROCEDURE SetUp;
    FORWARD;
    { Create a call socket, connect to server using IPCDest }

PROCEDURE ShutdownSockets;
    FORWARD;
    { Shut down the call and vc sockets }


$ TITLE 'Error_Routine', PAGE $
PROCEDURE Error_Routine
  (VAR where  : name_of_call_array_type;
       what   : integer;
       sd     : integer);

    BEGIN    { Error_Routine }

    writeln('Client: Error occurred in ', where,' call.' );
    writeln('Client: The error code is: ', what:5,
                  '.  The local descriptor is: ', sd:4 );

    GOTO 89;

    END;    { Error_Routine }

$ TITLE 'GetLen', PAGE $
PROCEDURE GetLen
(VAR  buffer      :   Buffer_Type;
 VAR  current_pos :   ShortInt;
 VAR  length      :   Integer );

{ Get the length of a string.  Return the next postion }

VAR
    orig_pos :    ShortInt;

    BEGIN    { GetLen }
    {}
    { Find the first blank in the string.  Return the difference
    { between the blank position, and the initial value of current_pos
    {}

    orig_pos := current_pos;

    WHILE buffer[current_pos] <> ' ' DO
       current_pos := current_pos + 1;
```

```
    { set the length value for the caller }
    length := current_pos - orig_pos;

    { increment beyond the space, for the next time }
    current_pos := current_pos + 1;

    END;      { GetLen }
$ TITLE 'Initialize_Option ', PAGE $

PROCEDURE Initialize_Option
  (VAR opt_parameter : byte_array_type);

VAR
  opt_num_arguments : ShortInt;
  result            : ShortInt;

BEGIN   {Initialize_Option}

  opt_num_arguments := 0;
  INITOPT( opt_parameter,opt_num_arguments,result );
  IF result <> ZERO THEN
    BEGIN       { error on initopt }
    call_name := 'INITOPT    ';
    Error_Routine( call_name, result, 0 );
    END;        { error on initopt }

END;    {Initialize_Option}


$ TITLE 'SetUp', PAGE $
PROCEDURE SetUp;
{ Create a call socket using a well-known address }

VAR
   pathdesc     : Integer;

   BEGIN    { SetUp }

   { Prepare to create a call socket }
   socket_kind := CALL_SOCKET;
   protocol_kind := TCP;


   { clear the flags and option arrays }
   flags_array := 0;
   Initialize_Option( option );

   {}
   {A call socket is created by calling IPCCREATE.  The value returned
   {in the call_sd parameter will be used in the following calls.
   {}

   IPCCREATE( socket_kind, protocol_kind, flags_array, option,
            call_sd, error_return );
```

```
IF error_return <> ZERO THEN
   BEGIN
   call_name := 'IPCCREATE ';
   Error_Routine( call_name,error_return, call_sd );
   END;

{}
{ The server is waiting on a well-known address.  Get the path
{ descriptor for the socket from the remote node.
{}
proto_addr := 31767;
flags_array := 0;

IPCDest( socket_kind, node_name, node_name_len, protocol_kind,
   proto_addr, INTEGER_LEN, flags_array, option,
   pathdesc, error_return );
IF error_return <> ZERO THEN
   BEGIN
   call_name := 'IPCDEST   ';
   Error_Routine( call_name,error_return, pathdesc );
   END;

flags_array := 0;

{ Now connect to the server }
IPCConnect( call_sd, pathdesc, flags_array, option,
                        vc_sd, error_return );
IF error_return <> ZERO THEN
   BEGIN
   call_name := 'IPCCONNECT';
   Error_Routine( call_name,error_return, pathdesc );
   END;

{ Set the timeout to infinity with IPCControl for later calls }
flags_array := 0;
control_value := 0;
timeout_len := 2;

IPCControl( vc_sd, CHANGE_TIMEOUT, control_value, timeout_len,
   dummy_parm, dummy_len, flags_array, error_return );

IF error_return <> ZERO THEN
   BEGIN
   call_name := 'IPCCONTROL';
   Error_Routine( call_name,error_return, vc_sd );
   END;

flags_array := 0;
Initialize_Option( option );

{}
{ Verify the server received the connect req.  Wait for the
{ server to do an IPCRecvCn.
{}
```

```
    IPCRecv( vc_sd, data_buf, buffer_len, flags_array,
                                        option, error_return );
    IF error_return <> ZERO THEN
       BEGIN
       call_name := 'IPCRECV   ';
       Error_Routine( call_name,error_return, vc_sd );
       END;

    END;     { SetUp }

$ TITLE 'ShutdownSockets', PAGE $
PROCEDURE ShutdownSockets;

VAR
    result    :             Integer;

    BEGIN    { ShutdownSockets }
    {}
    { We are terminating this program.  Clean up the allocated
    { sockets.
    {}
    flags_array := 0;
    Initialize_Option( option );

    IPCShutdown( vc_sd, flags_array, option, result );
    { Don't worry about errors here, since there isn't much we can do. }

    IPCShutdown( call_sd, flags_array, option, result );
    { Don't worry about errors here, since there isn't much we can do. }

    END;     { ShutdownSockets }

$TITLE 'Client MAIN', PAGE $
BEGIN { Client }

node_name_len := 0;
requested_name := '';

{ Ask the user for the NS node name of the remote node }
Prompt( 'Client: Enter the remote node name: ' );
Readln( node_name );

temp_position := 1;
GetLen( node_name, temp_position, node_name_len );

{ Create a call socket and connect to the server }
SetUp;

WHILE requested_name <> 'EOT' DO
    BEGIN    { loop for name }

    { Ask the user for a name to be retrieved }
    Prompt( 'Client: Enter name for data retrieval: ' );
    Readln( requested_name );
```

```
    req_name_len := BUFFERLEN;
    flags_array := 0;

    IF requested_name <> 'EOT' THEN
       BEGIN    { continue processing }

       { Ask for the name the user requested }
       IPCSend( vc_sd, requested_name, req_name_len, flags_array, option,
             error_return );

       { Block waiting for the response back from the server. }
       buffer_len := INFOBUFLEN;
       flags_array := 0;

       IPCRecv( vc_sd, data_buf, buffer_len,  flags_array, option,
             error_return );
         IF error_return <> ZERO THEN
           BEGIN      { error on initopt }
           call_name := 'IPCRECV   ';
           Error_Routine( call_name, error_return, vc_sd );
           END;       { error on initopt }

       { Print out the data received }
       Writeln( 'Client data is: ', data_buf );

       END;      { continue processing }
    END;     { loop for name }
89:

{ Clean up the call and vc sockets }
ShutDownSockets;

99:


END.  { Client }
```

Error Messages

This appendix lists and describes the messages that can be returned to the *result* and error parameters of NetIPC calls. The ASCII message associated with each code can be used in C programs. These constants are listed in the NS include file `/usr/include/sys/ns_ipc.h`.

0	MESSAGE	**NSR_NO_ERROR**
	CAUSE	The call was successful.
	ACTION	No action is necessary.
3	MESSAGE	**NSR_BOUNDS_VIO**
	CAUSE	Parameter bounds violation.
	ACTION	An address references memory to which the user does not have access rights.
4	MESSAGE	**NSR_NETWORK_DOWN**
	CAUSE	The network is down. The system is not initialized for networked operation.
	ACTION	Use `ifconfig lan0` to see if the "down" flag is displayed. If not, this may be an internal error. Consult your Network Manager.

5	MESSAGE	**NSR_SOCK_KIND**
	CAUSE	Illegal socket kind. The calling process attempted to create a kind of socket that the system does not support.
	ACTION	Check the *socketkind* parameter to make sure it matches one of the socket kind supported by the system.
6	MESSAGE	**NSR_PROTOCOL**
	CAUSE	Illegal protocol. The protocol referenced is not supported by the system.
	ACTION	One or more of the following actions may be taken: (1) Check the *protocol* parameter to make sure it matches one of the protocol types supported by the system; (2) make sure the system supports the referenced protocol; (3) consider defaulting the protocol argument to zero, thus letting the system decide which protocols are best.
7	MESSAGE	**NSR_FLAGS**
	CAUSE	Illegal flags. A `flags` bit was set that is not supported.
	ACTION	Check the *flags* parameter to make sure that the correct bits are set. Some calls may return information through the *flags* parameter and the bits returned may not be valid input on subsequent calls.

8	MESSAGE	**NSR_OPT_OPTION**
	CAUSE	Illegal option. An illegal option was specified in the *opt* parameter.
	ACTION	Check the *opt* parameter to make sure that it was correctly initialized with `initopt()` and that all options added with `addopt()` are defined for the system and system call.
10	MESSAGE	**NSR_KIND_AND_PROTOCOL**
	CAUSE	Protocol type mismatch. A protocol and a socket kind type were specified that are not supported together.
	ACTION	One or more of the following actions may be taken: (1) Check the *socketkind* and *protocol* parameters for the correct values; (2) default the protocol value to zero, thus letting the system decide which protocols best support the referenced socket kind.

11	MESSAGE	**NSR_NO_MEMORY**
	CAUSE	No memory. The system does not have enough memory available to support the request. This error can occur when you attempt to issue an `ipccreate()`, `ipcconnect()`, `ipcrecvcn()`, `ipclookup()`, or `ipcdest()` call.
	ACTION	One or more of the following actions may be taken: (1) release some of the sockets or destination descriptors that are not currently being used; (2) if applicable, reduce the service requirements for the socket being created (eg., by requesting smaller message sizes); (3) determine if some of the other programs running on the system can release some of their networking resources.
14	MESSAGE	**NSR_ADDR_OPT**
	CAUSE	This error is returned to `ipcdest()` if an invalid value is specified in the *`protolen`* parameter. The *`protolen`* parameter indicates the length of a protocol address.
	ACTION	Check the length specified in the *`protolen`* parameter. For TCP protocol addresses, the protocol parameter must be 2.
15	MESSAGE	**NSR_NO_FILE_AVAIL**
	CAUSE	No file table entries are available. Close unnecessary open files and retry.
	ACTION	If the error persists, reconfigure a larger File Table.

18	MESSAGE	**NSR_OPT_SYNTAX**
	CAUSE	An error was detected in the option array syntax.
	ACTION	Check the option array for valid fields.
21	MESSAGE	**NSR_DUP_OPTION**
	CAUSE	Duplicate option. At least one of the options in the *opt* parameter was specified twice.
	ACTION	Check arguments input to `addopt()` when initializing the *opt* parameter.
24	MESSAGE	**NSR_MAX_CONNECTQ**
	CAUSE	Connection queued option error. An error was detected in the arguments regarding the maximum number of connections queued option in the *opt* parameter.
	ACTION	Check the `addopt()` call that was used to put the `NSO_MAX_CONN_REQ` option argument in the *opt* parameter. Must be less than or equal to 20.
28	MESSAGE	**NSR_NLEN**
	CAUSE	Illegal name length. The name length was either too large or too small.
	ACTION	Compare the name length to the acceptable range for this parameter.

29	MESSAGE	**NSR_DESC**
	CAUSE	Illegal descriptor. The referenced descriptor is outside of the acceptable range for socket descriptors. The descriptor might have been a disc file descriptor or a closed socket descriptor.
	ACTION	Determine why the value was not within the acceptable range. One possible reason is that the call to allocate the descriptor failed. Also check for socket descriptors that had been already closed.
30	MESSAGE	**NSR_CANT_NAME_VC**
	CAUSE	Cannot name VC socket. The calling process tried to name a VC socket using `ipcname()`.
	ACTION	`ipcname()` cannot be invoked against VC sockets.
31	MESSAGE	**NSR_DUP_NAME**
	CAUSE	Duplicate name. The name that `ipcname()` tried to assign to a socket was already in use.
	ACTION	One of the following actions may be taken: (1) Pick another name; (2) wait and try again; (3) if several copies of the same process are running, make sure that each process has some way of generating a unique name. `ipcname()` has a random name generation facility that could be used, or the calling process could wait and try again later.

36	MESSAGE	**NSR_NAME_TABLE_FULL**
	CAUSE	Name table full. A process attempted to bind a name to a socket via `ipcname()` when the system had no free name records. A name record must be allocated for each name that is bound to a socket. When the system runs out of name records, all succeeding `ipcname()` requests are rejected.
	ACTION	Release some of the names that are bound to sockets. This may be done using `ipcnamerase()`. Because name records are system-wide resources shared by all NetIPC programs, the name records released by one program may be allocated for use by another.
37	MESSAGE	**NSR_NAME_NOT_FOUND**
	CAUSE	Name not found. A process attempted to obtain a destination descriptor using `ipclookup()`, but the name specified in the call was not registered in the referenced socket registry.
	ACTION	One or more of the following actions may be taken: (1) Make sure that the name specified in the `ipclookup()` call was the one that was intended; (2) consider that the failure could have been due to a race condition (the `ipclookup()` caller could have executed its call before the `ipcname()` caller executed its call).

38	MESSAGE	**NSR_NO_OWNERSHIP**
	CAUSE	No ownership. The caller invoked `ipcnamerase()` specifying a valid name but one bound to a socket that the it does not own. Only the owner of a call socket may purge its name.
	ACTION	Check that the name specified is the one the caller intended to use.
39	MESSAGE	**NSR_NODE_NAME_SYNTAX**
	CAUSE	Illegal node name. The caller invoked `ipclookup()`, `ipcdest()`, or `ipcsetnodename()` passing it a node name having an illegal syntax (for example, too many levels of hierarchy or too many characters in one of the name parts).
	ACTION	Verify that the name passed was the intended one or verify that the length specified for the passed name was correct.
40	MESSAGE	**NSR_NO_NODE**
	CAUSE	Unknown node. The caller invoked `ipclookup()` or `ipcdest()` with the name of a node that was unknown to the local node. A local node resolves a node name by using the PROBE protocol.
	ACTION	One or more of the following actions can be taken: (1) Verify that the name specified was the intended one; (2) check to see if the node is down; (3) verify that the `nodename` command was executed to assign the node name or (4) if the node exists on a remote network, verify that a proxy server exists on the local network and that it has an entry configured for the remote node. If the remote node is non-HP-UX, check that IEEE 802.3 is turned on locally (use the *lanconfig* command).

43	MESSAGE	**NSR_CANT_CONTACT_SERVER**
	CAUSE	Could not send an `ipclookup()` request. Problem may be due to lack of kernal memory or the system may be heavily loaded.
	ACTION	Try again.
44	MESSAGE	**NSR_NO_REG_RESPONSE**
	CAUSE	No socket registry response. A name look up query was sent to the remote socket registry referenced in an `ipclookup()` call, but the registry never responded. The node upon which the socket registry resides might be down, unreachable, or the system may be heavily loaded and not responding.
	ACTION	If the node crashed, is temporarily unreachable or heavily loaded, the caller may wait and try again later. If the remote node is non-HP-UX, use *lanconfig* to verify that IEEE 802.3 is turned on locally.
45	MESSAGE	**NSR_SIGNAL_INDICATION**
	CAUSE	System call aborted due to signal.
	ACTION	Retry if appropriate.

46	MESSAGE	**NSR_PATH_REPORT**
	CAUSE	Could not interpret path. The address information referenced by the specified destination descriptor contained uninterpretable information. When this error occurs, it may be indicative of a system software error. It may also indicate that the destination descriptor was somehow corrupted between the time it was generated and the time it was interpreted.
	ACTION	Assuming the problem is due to corruption of the destination descriptor and not a system software error, try shutting down the referenced destination descriptor and then performing another `ipclookup()`. If the same error is returned when the new destination descriptor is used, this error requires HP notification.
47	MESSAGE	**NSR_BAD_REG_MSG**
	CAUSE	Received corrupted message from socket registry.
	ACTION	Retry. If the problem persists, this error requires HP notification.
50	MESSAGE	**NSR_DLEN**
	CAUSE	Bad length. The data length specified was either too long or too short.
	ACTION	One or more of the following actions may be taken: (1) Verify that the data length specified was the data length intended; (2) Verify that the size specified was not larger or smaller than maximum or minimum permissible receive size of the socket

51	MESSAGE	**NSR_DEST**
	CAUSE	Not a destination descriptor. The descriptor specified in the parameter reserved for destination descriptors did not describe a destination descriptor.
	ACTION	(1) Verify that the descriptor was the one intended; (2) Verify that you meant to execute an `ipcdest()` or `ipclookup()` call.
52	MESSAGE	**NSR_PROTOCOL_MISMATCH**
	CAUSE	Protocol mismatch. The call socket referenced in an `ipccreate()` or `ipcdest()` call is not bound to any of the protocols that the destination descriptor references (i.e., there is no way to use the protocol referenced by the call socket to access the socket referenced by the destination descriptor).
	ACTION	One of the following actions may be taken: (1) Do not specify a particular protocol when creating the call socket. Instead, use the default protocol value of zero in `ipccreate()`'s *`protocol`* parameter; (2) create a new call socket and bind it to a different protocol and try again.
53	MESSAGE	**NSR_SOCKET_MISMATCH**
	CAUSE	Socket type mismatch. The destination descriptor specified in an `ipcconnect()` call does not reference a remote call socket. This error occurs when the remote socket is supported by a system that supports socket kinds other than those supported on the local system.
	ACTION	None, unless the remote application that the calling process wants to connect to can be modified to use call sockets.

54	MESSAGE	**NSR_NOT_CALL_SOCKET**
	CAUSE	Not a call socket descriptor.
	ACTION	(1) Verify it is the one intended; (2) Verify the original `ipccreate()` call.
56	MESSAGE	**NSR_WOULD_BLOCK**
	CAUSE	Would block error. The calling process issued a request that could not be immediately satisfied against a socket that was in asynchronous mode.
	ACTION	This is an informational message so no action is necessary. For more information on asynchronous I/O, refer to the "NetIPC Concepts" chapter.
59	MESSAGE	**NSR_SOCKET_TIMEOUT**
	CAUSE	Timed out. The calling process's request timed out. The request was an `ipcselect()` call or a NetIPC call issued against a socket that was in synchronous mode (the default mode for NetIPC sockets). Time out errors that occur on calls issued against VC sockets do not concern the protocol or connection they reference; protocols use their own timers to determine if a connection is not functioning reasonably. Possible scenarios in which this error could occur include: (1) An `ipcsend()` call could not obtain the buffer space needed to accommodate its data within the synchronous time-out interval; (2) an `ipcrecv()` call's request for data could not be satisfied within the synchronous time-out interval; (3) a connection request was not received by an `ipcrecvcn()` call within the synchronous time-out interval; (4) a process attempted to send or receive data before a virtual circuit connection was established.
	ACTION	Check your programs to make sure that the event the socket is expecting will indeed occur. In scenarios 1 through 3 above, you should also consider modifying the

		socket's associated time out interval. Refer to the discussion of `ipccontrol()` in the "NetIPC Calls" chapter for information on adjusting the synchronous time-out. If scenario 4 has occurred, make sure your programs are synchronized as shown in the "NetIPC Concepts" chapter.
60	MESSAGE	**NSR_NO_DESC_AVAIL**
	CAUSE	The file descriptor limit was exceeded. The calling process attempted to gain access to a new socket descriptor or destination descriptor even though it already owned the maximum permissible number of descriptors (60).
	ACTION	The process must release one of the socket descriptors or destination descriptors that it owns and then retry the request.

62	MESSAGE	**NSR_CNCT_PENDING**
	CAUSE	`ipcrecv()` expected. An attempt was made to manipulate a VC socket whose corresponding connection had been initiated with `ipcconnect()` but whose successful establishment had not been completed via `ipcrecv()`. A user cannot send or receive on a VC socket that was created with `ipcconnect()` without first having called `ipcrecv()` to complete the establishment sequence.
	ACTION	Call `ipcrecv()` to verify that the connection referenced by the VC socket came up before trying to send or receive again.
64	MESSAGE	**NSR_REMOTE_ABORT**
	CAUSE	Connection aborted. The connection underlying a VC socket has been aborted either by the protocol handler running on the local node because it was unable to contact its peer protocol handler at the remote end of the connection, or by the protocol handler on the node at the other end of the connection. This error may be returned when (1) the remote node is down, (2) some network links are malfunctioning, (3) the network is extremely congested, (4) the user of the connection told the remote protocol handler to abort the connection, or (5) the remote process aborted. This error can be used to detect that the remote peer has completed transmission and has shut down the connection.
	ACTION	Consult your Network Manager for assistance in diagnosing the problem.

65	MESSAGE	**NSR_LOCAL_ABORT**
	CAUSE	Connection aborted. The connection underlying a VC socket has been aborted by the protocol handler running on the local node because it was unable to contact its peer protocol handler at the remote end of the connection. This error may be returned when (1) the remote node goes down, (2) some network links are malfunctioning, (3) the network is extremely congested, (4) or the connection could not be established because there is not a common encapsulation method.
	ACTION	Consult your Network Manager for assistance in diagnosing the problem. Use the *lanconfig* command to verify that the local and remote nodes have a common encapsulation method (IEEE or Ethernet).
66	MESSAGE	**NSR_NOT_CONNECTION**
	CAUSE	Not a VC socket. The descriptor specified in the parameter reserved for VC socket descriptors did not describe a VC socket.
	ACTION	One or more of the following actions may be taken: (1) Verify that the descriptor specified was the one that the calling process intended to specify; (2) verify that the original call to create the VC socket succeeded; (3) do not use Berkeley sockets with NetIPC calls.

74	MESSAGE	**NSR_REQUEST**
	CAUSE	Illegal request. The request code passed in an `ipccontrol()` request was not valid. Or, the request is not valid for the kind of socket.
	ACTION	One or more of the following actions may be taken: (1) Verify that the request code specified was the intended one; (2) verify that the request code is supported on the local system (consult the "NetIPC Concepts" chapter); (3) Verify that the request is meaningful for the kind of socket.
76	MESSAGE	**NSR_TIMEOUT_VALUE**
	CAUSE	Illegal time out value. The `ipccontrol()` or `ipcselect()` request invoked by the calling process specified a time out value that was invalid.
	ACTION	One or more of the following actions may be taken: (1) Verify that the time out value specified was the intended value; (2) consult the "NetIPC Concepts" chapter to make sure the value is acceptable.
99	MESSAGE	**NSR_VECT_COUNT**
	CAUSE	Bad vector data length. The calling process specified a data vector argument that contained a negative length field.
	ACTION	Recheck the initialization of the data vector.

100	MESSAGE	**NSR_TOO_MANY_VECTS**
	CAUSE	Too many vectored data descriptors.
	ACTION	Recode your program so that the number of vectored data descriptors is within acceptable limits. Refer to the "NetIPC Concepts" chapter for information on data vectors.
106	MESSAGE	**NSR_DUP_ADDRESS**
	CAUSE	Address in use. The caller process requested that its call socket descriptor be bound to a particular protocol address, but the address was already bound to another call socket descriptor.
	ACTION	One or more of the following actions may be taken: (1) Verify that the address specified was the intended one; or (2) check to make sure there are not duplicate copies of the program running.
109	MESSAGE	**NSR_REMOTE_RELEASED**
	CAUSE	The remote endpoint of the connection has been released. You can continue to send on the local endpoint, but data will not be received.
	ACTION	This is an informational message only. No action is required.

116	MESSAGE	**NSR_DEST_UNREACHABLE**
	CAUSE	No usable paths. The local node's protocol software cannot connect to the remote node described by the destination referenced by a passed destination descriptor. This could occur because the local node does not know where the remote node's network is, or because the remote node does not support the same protocols as the local node.
	ACTION	Obtain a new destination descriptor using `ipclookup()`. If this is not successful, ask the System Manager to verify that the correct routing information is configured locally so that the remote network can be reached. Also, determine which protocols are supported by the remote node.
118	MESSAGE	**NSR_VERSION**
	CAUSE	Version number mismatch.
	ACTION	Make sure that all processes are running on nodes with the same version of the LAN software.
124	MESSAGE	**NSR_OPT_ENTRY_NUM**
	CAUSE	Bad entry number specified.
	ACTION	Check syntax of *opt* structure.
125	MESSAGE	**NSR_OPT_DATA_LEN**
	CAUSE	Bad option data length. The data length specified in the `addopt()` or `readopt()` call was invalid.
	ACTION	Verify that the value passed was the intended value.

126	MESSAGE	**NSR_OPT_TOTAL**
	CAUSE	Bad option total. `initopt()` was invoked specifying that the number of eventual entries to be placed into the *opt* parameter would be either fewer than zero or greater than the maximum possible number of *opt* entries.
	ACTION	One or more of the following actions may be taken: (1) Verify that the value passed for the eventual number of entries argument was the intended value; (2) recalculate the number of entries that will actually be needed.
127	MESSAGE	**NSR_OPT_CANTREAD**
	CAUSE	Cannot read option. The *opt* entry specified in the `readopt()` call was not initialized.
	ACTION	One or more of the following actions may be taken: (1) Verify that the *opt* parameter was properly initialized with `initopt()`; (2) verify that the referenced entry was set up properly with `addopt()`.

1002	MESSAGE	**NSR_THRESH_VALUE**
	CAUSE	Bad threshold value. This error is returned for one of the following reasons: (1) an illegal read threshold was specified, or (2) an illegal write threshold was specified.
	ACTION	Verify that the value passed was the intended value and that it was not negative or zero or greater than the socket's maximum receive size specified when the socket was created.
2003	MESSAGE	**NSR_NOT_ALLOWED**
	CAUSE	User not a super user. The caller attempted to use functionality restricted to super users.
	ACTION	This is an informational message only. No action is required.
2004	MESSAGE	**NSR_MSGSIZE**
	CAUSE	The message size being used is too large for the protocol. This error if a process requests a maximum send or receive size larger than the maximum allowed or if a process attempts to send or receive more than the number of bytes set in the `ipcconnect()` or `ipcrecv()` call.
	ACTION	Make sure your `ipcconnect()` or `ipcrecv()` call does not attempt to set the maximum send and receive sizes to larger than 32,767 bytes. Also, make sure your process does not attempt to send or receive more bytes than specified by the `ipcconnect()` call or `ipcrecv()` call for that connection.

2005	MESSAGE	**NSR_ADDR_NOT_AVAIL**
	CAUSE	Tries to connect to an unavailable protocol address.
	ACTION	(1) Verify the destination descriptor was the one intended to use, or, (2) Verify the protocol address is correct and available for connection.

System Calls and NetIPC Sockets

NetIPC processes make use of sockets via the NetIPC calls to establish connections and exchange data. The Transport Layer's Transmission Control Protocol (TCP) regulates the transmission of data to and from sockets. NetIPC processes reference sockets with socket descriptors. Socket descriptors are returned to processes when certain NetIPC calls are invoked. Socket descriptors are allocated from the same space as file descriptors. Therefore, sockets are accessible through the standard HP-UX file system calls.

Table C-1 describes HP-UX system calls that operate on NetIPC sockets. The NetIPC calls are described in the "NetIPC Calls" chapter. HP-UX system calls are described in the HP-UX Reference Manual.

Table C-1. System Calls and NetIPC Sockets

HP-UX Call	Description
`acct()`	The `ac_io` field in accounting file records will be updated.
`close()`	When `close()` is used on a NetIPC socket, it does not guarantee that any data buffered will actually be sent.
`dup()`	Supported as described in the *HP-UX Reference Manual.*
`exec()` `execve()`	Sockets remain open over `exec()` and/or `execve()`.
`fchown()`	Not supported for NetIPC sockets.
`fork()`	Socket descriptors are inherited by the child process. Refer to the "NetIPC Concepts" chapter for more information on shared socket descriptors.
`fstat()`	The `stat` structure is undefined.
`read()`	Supported as described in the *HP-UX Reference Manual* for VC sockets only.
`select()`	Read and write thresholds for sockets are supported. Read, write and exception conditions for sockets are defined in the "NetIPC Calls" chapter.
`ulimit()`	No limits are currently supported for NetIPC usage.
`write()`	Supported for VC sockets only.

LAN/9000 Series 600/800 Migration

This appendix provides an introduction to LAN/9000 Series 600/800 for users who are current DS/1000-IV, NS/1000 or NS/9000 Series 200 or 500 customers. Because it addresses three different audiences, this appendix is organized into three different sections:

- **LAN/9000 Series 600/800 for DS/1000-IV Users.** This section compares DS/1000-IV and LAN/9000 Series 600/800.
- **NS/1000 to LAN/9000 Series 600/800 Migration.** This section compares NS/1000 and LAN/9000 Series 600/800. For information on porting NS/1000 applications to LAN/9000 Series 600/800, see the "Porting NetIPC Programs" appendix.
- **NS/9000 to LAN/9000 Series 600/800 Migration.** This section compares the NS/9000 product provided on the HP 9000 Series 200 and 500 to LAN/9000 Series 600/800.

Note For information on migrating from DS/1000-IV, NS/1000 or NS/9000 to NS/9000 Series 600/800, refer to the *Using Network Services (NS)/9000 Series 600/800* manual.

LAN/9000 Series 600/800 for DS/1000-IV Users

LAN/9000 Series 600/800 and DS/1000-IV do not share any common user services. As a result, programs written using DS/1000-IV calls are not transportable to LAN/9000 Series 600/800 systems.

Migration Analysis Utility

In order to help customers migrate from DS/1000-IV to LAN/9000 Series 600/800, Hewlett-Packard has developed a utility that reads RTE program source files and flags DS/1000-IV calls. This program can be used as an aid in determining the difficulty of converting a program to use LAN/9000 Series 600/800 calls and in locating calls that must be modified.

For more information about this utility, refer to the *PORT/HP-UX Migration Analysis Utility Manual* (92561-90002).

Feature Comparison

Table D-1 maps the DS/1000-IV services to LAN/9000 Series 600/800 and NS/9000 Series 600/800 services with similar functionality. **There is no one-to-one correspondence between DS/1000-IV and LAN/9000 Series 600/800 or NS/9000 Series 600/800 services.**

Table D-1. DS/1000-IV vs. LAN & NS/9000 Series 800

DS/1000-IV Service	Similar LAN & NS/9000 Series 800 Service
Program-to-Program Communication (PTOP)	Network Interprocess Communication (NetIPC)
REMAT	Network File Transfer (NFT)
RMOTE	Network File Transfer (NFT)

Note Network File Transfer are services provided by NS/9000 Series 600/800. Refer to the *Using Network Services (NS)/9000 Series 600/800* manual for a detailed comparison of DS/1000-IV and NS/9000 Series 600/800 product features.

DS/1000-IV and LAN/9000 Series 600/800 share two similar services: PTOP and NetIPC. The following paragraphs compare these services.

Interprocess Communication

The DS/1000-IV service Program-to-Program Communication (PTOP) allows a **master** program to exchange information with and control the execution of a **slave** program. PTOP calls are divided into two categories, master calls and slave calls. The master and the slave programs may be located at the local node, or one may be at the local node and the other may be at a remote node. DS/1000-IV PTOP programs can communicate with other PTOP programs on remote DS/1000-IV or DS/3000 nodes.

LAN/9000 Series 600/800 provides a service similar to PTOP called Network Interprocess Communication (NetIPC) which is documented in this manual. NetIPC allows two or more **peer** processes to exchange information; one program does not control the execution of another. Because NetIPC operates in a peer-to-peer rather than master-to-slave fashion, any NetIPC process can use any of the NetIPC calls. As a result, the form of interprocess communication offered by NetIPC is more flexible than that provided by PTOP. NetIPC peer processes may be located on the same or different nodes.

NetIPC processes establish connections with other processes via sockets. A NetIPC process can engage in a dialogue that references certain sockets in order to create a connection with another NetIPC process. Several NetIPC calls are provided to allow processes to engage in this dialogue. Once a connection, called a virtual circuit, is established, the processes may exchange data. A LAN/9000 Series 600/800 NetIPC process can communicate with other NetIPC processes on remote LAN/9000 Series 600/800 and NS/1000 nodes only.

Unlike PTOP, NetIPC does not provide a call to schedule a remote process. Remote processes must be manually started or can be daemons that are started at system start up.

Table D-2 maps the DS/1000-IV PTOP calls to similar LAN/9000 Series 600/800 NetIPC calls. Most of the NetIPC calls have no PTOP equivalents; therefore, they are not listed in the table. These calls are not similar to any PTOP calls because they are primarily used to create and establish virtual circuit connections. The NetIPC calls that have no PTOP equivalents include `ipcconnect()`, `ipccreate()`, `ipcdest()`, `ipclookup()`, `ipcname()`, `ipcnamerase()`, `ipcrecvcn()`, `ipcselect()`, `addopt()`, `initopt()`, and `readopt()`.

Table D-2. PTOP Calls vs. NetIPC Calls

PTOP Call	NetIPC Call	Comparison
POPEN	No similar NetIPC call.	Series 600/800 NetIPC does not provide a call to schedule a peer process.
PREAD	`ipcrecv()`	PREAD allows a PTOP master program to receive data contained in the buffer parameter of a slave program's ACEPT call. The master program must call PREAD before the slave program can transmit data via an ACEPT call. `ipcrecv()` allows a NetIPC process to receive data queued on a virtual circuit connection. The data becomes queued on this connection when another NetIPC process calls `ipcsend()`.
PWRIT	`ipcsend()`	PWRIT allows a PTOP master program to transfer data contained in its buffer parameter to the buffer parameter of a slave program's GET call. `ipcsend()` allows a NetIPC process to send data on a virtual circuit connection. The data becomes queued on this connection and may be dequeued by another NetIPC process when that process calls `ipcrecv()`.

Table D-2. PTOP Calls vs. NetIPC Calls-con't

PTOP Call	NetIPC Call	Comparison
PCONT	ipcsend()	PCONT allows a PTOP master program to transfer data contained in its *tag* parameter to the *tag* parameter of a slave program's GET call.
PCLOS	ipcshutdown()	PCLOS allows a PTOP master program to terminate a slave program. If the slave program resides on an HP 1000 node, PCLOS also terminates logical communication with that node. ipcshutdown() may be used to abort a virtual circuit connection. This terminates logical communication with a peer process over that connection. ipcshutdown() can *not* be used to terminate a peer process; NetIPC does not provide a call with this functionality.
PNRPY	ipccontrol()	PNRPY allows a PTOP master program to issue PWRIT, PCONT and PCLOS requests asynchronously. Master programs that use this call will not be suspended when they issue requests to send data to, or to terminate, slave programs. The NetIPC call ipccontrol() may be used to enable asynchronous I/O between NetIPC processes. Unlike PNRPY, ipccontrol() allows processes to both send and receive data asynchronously by placing the sockets shared by the processes in asynchronous mode.

Table D-2. PTOP Calls vs. NetIPC Calls-con't

PTOP Call	NetIPC Call	Comparison
GET	ipcrecv()	The main function of the PTOP slave call GET is to receive master program requests (PWRIT, PREAD, etc.) However, the tag and buffer parameters of the GET call can be used to receive data sent by the master program. The NetIPC call ipcrecv() is similar to GET only in that it allows a process to receive data. (Refer to the discussion of PREAD above for more information on ipcrecv().)
ACEPT	ipcrecv() and ipcsend()	The PTOP slave call ACEPT allows PTOP slave programs to receive data from, and send data back, to PTOP master programs. When a master program sends data via a call to PWRIT, the buffer parameter of the ACEPT call can be used to receive the data. When a master program requests data via a call to PREAD, the ACEPT's buffer parameter contains the data that will be transmitted to the master program. The ACEPT call also contains a tag parameter that can be used to send data to the master program. The ACEPT call's data acceptance and transmission functions are similar to ipcrecv() and ipcsend(), respectively. (Refer to the discussion of PWRIT and PREAD above for more information on these NetIPC calls.)

Table D-2. PTOP Calls vs. NetIPC Calls-con't

PTOP Call	NetIPC Call	Comparison
REJCT	ipcsend()	The main function of the PTOP slave call REJCT is to reject a master request. REJCT also contains a tag field that can be used to transmit data back to the master program. This secondary feature of REJCT is similar to the NetIPC call ipcrecv(). (Refer to the discussion of PREAD above for more information on ipcrecv().)
FINIS	ipcshutdown()	The PTOP slave call FINIS is similar to the PTOP master call PCLOS in that it terminates communication between two programs. The NetIPC call ipcshutdown() terminates logical communication over a certain connection.

NS/1000 to LAN/9000 Series 600/800 Migration

NS/1000 and LAN/9000 Series 600/800 share the same HP AdvanceNet NS user service NetIPC. NS/1000 and NS/9000 Series 600/800 also share the same HP AdvanceNet NS user service Network File Transfer (NFT).

Refer to Appendix E, "Porting NetIPC Programs," in this manual for information regarding transporting NS/1000 NetIPC programs to the LAN/9000 Series 600/800 programming environment. Refer to the *Using Network Services (NS)/9000 Series 600/800* manual for a detailed comparison of the NS/1000 and NS/9000 Series 600/800 NFT implementations.

As shown in the previous section, "LAN/9000 Series 600/800 for DS/1000-IV Users," NS/1000's DS/1000-IV Compatible Services (Remote File Access, Distributed Executive, Program-to-Program Communication, REMAT, RMOTE and Remote File Access) are not supported on LAN/9000 Series 600/800 nodes.

In order to help customers migrate from DS/1000-IV to LAN/9000 Series 600/800, Hewlett-Packard has developed a utility that reads RTE program source files and flags DS/1000-IV calls. This program can be used as an aid in determining the difficulty of converting a program to use LAN/9000 Series 600/800 calls and in locating calls that must be modified. For more information about this utility, refer to the *PORT/HP-UX Migration Analysis Utility Manual.*

Table D-3 maps the NS/1000 services to the same or similar LAN/9000 Series 600/800 and NS/9000 Series 600/800 services.

Table D-3. NS/1000 vs. LAN & NS/9000 Series 800

NS/1000 Service	LAN or NS/9000 Series 800 Service
Network File Transfer (NFT)*	Network File Transfer (NFT)*
Network Interprocess Communication (NetIPC)*	Network Interprocess Communication (NetIPC)*
Program-to-Program Communication (PTOP)	Network Interprocess Communication (NetIPC)
REMAT	Network File Transfer (NFT)
RMOTE	Network File Transfer (NFT)

*Indicates compatible HP AdvanceNet NS user services.

Note Network File Transfer is a service provided by the NS/9000 Series 600/800 product. Refer to the *Using Network Services (NS)/9000 Series 600/800* manual for a detailed comparison of the NS/1000 and NS/9000 Series 600/800 user services.

NS/9000 to LAN/9000 Series 600/800 Migration

LAN/9000 Series 600/800 and the NS/9000 product for the HP 9000 Series 200 and 500 do not share any common services. NS/9000 Series 500 Interprocess Communication ("IPC") and NS/9000 Series 600/800 Network Interprocess Communication (NetIPC) are *not* compatible services.

NS/9000 Series 600/800 and NS/9000 Series 200 and 500, however, do share the Network File Transfer (NFT). Refer to the *Using Network Services* manual for a detailed comparison of the Series 600/800 and NS/9000 user services.

Table D-4 maps the NS/9000 Series 200 and 500 services to the same or similar NS/9000 Series 600/800 and LAN/9000 Series 600/800 services.

Table D-4. NS/9000 vs. NS & LAN/9000 Series 800

NS/9000 Series 200 & 500 Service	NS & LAN/9000 Series 800 Service
Network File Transfer (NFT)*	Network File Transfer (NFT)*
Interprocess Communication (IPC) (Series 500 only)	Network Interprocess Communication (NetIPC)

*Indicates implementation of compatible user service.

The following paragraphs provide a comparison of the NS/9000 Series 600/800 NetIPC service and the NS/9000 Series 500 IPC service.

Interprocess Communication

NS/9000 Series 500 Interprocess Communication ("IPC") and LAN/9000 Series 600/800 Network Interprocess Communication (NetIPC) are not compatible services. However, because the services are somewhat similar, it may be useful to convert an NS/9000 IPC program to use LAN/9000 Series 600/800 NetIPC calls.

Features common to both NS/9000 Series 500 IPC and LAN/9000 Series 600/800 NetIPC include the following:

- Processes communicate with each other by means of sockets. Before a connection can be established between two processes, each process must create a socket. On the Series 500, this socket is called a **source socket**; on the Series 600/800, it is called a call socket.
- Source (or call) sockets may be **named**. A process can gain access to another process's socket by referencing the socket's name. When the name of a socket that belongs to another process is referenced in a "look up" call (`uipclookup()` for the Series 500, `ipclookup()` for the Series 600/800), the calling process receives a **destination descriptor**.
- Processes use destination descriptors in "connection request" calls (`uipcconnect()` for the Series 500, `ipcconnect()` for the Series 600/800). The connection request call returns a VC socket which is the endpoint of a virtual circuit connection.
- Communication between processes takes place over a virtual circuit connection.
- Connections can be set to synchronous or asynchronous communications mode. The default mode is synchronous, which is blocking mode. The communications mode can be reset to asynchronous using a "control" call (`uipccontrol()` for the Series 500, `ipccontrol()` for the Series 600/800).

Porting NetIPC Programs

This appendix summarizes differences and provides information to help you successfully port NetIPC programs between HP 1000 A-Series and HP 9000 Series 600/800 systems. Refer to the *NS/1000 User/Programmer Reference Manual* for NetIPC programming information for HP 1000 A-Series systems.

This appendix does *not* compare the programming language implementations at the different systems. For this information, you should refer to the following language reference manuals:

SYSTEM	LANGUAGE REFERENCE MANUALS
HP 9000 Series 600/800	*HP FORTRAN 77/HP-UX Reference Manual*
	HP C Reference Manual
	HP C/HP-UX Reference Manual Supplement
	HP Pascal Reference Manual
HP 1000 A-Series	*FORTRAN 77 Reference Manual*
	Pascal/1000 Reference Manual

In addition, the following manuals contain information that is useful to programmers porting FORTRAN 77 and Pascal programs from the HP 1000 to the HP 9000 Series 600/800:

- *HP FORTRAN 77/HP-UX Migration Guide.*
- *HP Pascal/HP-UX Migration Guide.*

When you are porting NetIPC programs, the following strategy may help:

1. Make sure that the NetIPC programs are executing correctly between homogeneous systems. That is, the programs should work between HP 1000 A-Series systems first.

2. Port the programs using the language reference manuals. Check carefully for compiler differences such as data types and lengths.

3. Check the differences between NetIPC calls documented in this appendix. Check all the parameters; some are not implemented or have different values.

4. If your ported programs still do not work, consider both programming language and NetIPC differences.

LAN/9000 Series 600/800 and NS/1000

This section describes the differences between the LAN/9000 Series 600/800 and NS/1000 NetIPC implementations.

Path Report and Destination Descriptors

In NS/1000 NetIPC, the descriptor returned by the socket registry software is called a **path report descriptor**; in LAN/9000 Series 600/800, this descriptor is called a **destination descriptor.** Although path report descriptors and destination descriptors have slightly different meanings, their function is the same: both contain addressing information that is used by a NetIPC process to direct requests to a certain call socket at a certain node.

Socket Ownership

An LAN/9000 Series 600/800 NetIPC process may own a maximum of 1024 descriptors. This limit includes call socket, VC socket, and destination descriptors as well as HP-UX file descriptors and NetIPC and/or file descriptors inherited or otherwise opened by the process.

An NS/1000 NetIPC process may own a maximum of 32 socket descriptors. This limit includes call socket, VC socket, and path report descriptors.

NS/1000 and LAN/9000 Series 600/800 NetIPC process creates a call socket by calling `IPCCreate`; they create a VC socket by calling `IPCConnect` or `IPCRecvCn`. An NS/1000 NetIPC process may also gain access to a socket by calling `IPCGive`. Sockets are given away with the `IPCGive` call.

The `IPCGive` and `IPCGet` calls are not part of the LAN/9000 Series 600/800 NetIPC implementation. Instead, LAN/9000 Series 600/800 processes can also acquire access to sockets owned by other NetIPC processes by utilizing socket "sharing." On HP 9000 Series 600/800 systems, NetIPC socket descriptors (call socket, VC socket, and destination), like HP-UX file descriptors, are copied to the "child" process when a process forks. As a result, more than one process can own a descriptor for the same socket. Programmers are responsible for regulating the use of shared sockets on LAN/9000 Series 600/800 systems. An NS/1000 NetIPC process creates a call socket by calling `IPCCreate` or `IPCGet`; it creates a VC socket by calling `IPCConnect` or `IPCRecvCn`. An NS/1000 NetIPC process may also gain access to a socket by calling `IPCGive`. Sockets are given away with the `IPCGive` call.

Socket Shut Down

The IPCShutDown call is used in both NS/1000 and LAN/9000 Series 600/800 NetIPC to release a descriptor and any resources associated with it. The shut down procedure for both NS/1000 and LAN/9000 Series 600/800 processes is identical with the following exception: the operation of the LAN/9000 Series 600/800 implementation of IPCShutDown is affected by socket sharing. The LAN/9000 Series 600/800 supports NSF_GRACEFUL_RELEASE.

When a LAN/9000 Series 600/800 NetIPC process "shuts down" a VC socket descriptor that is shared by other processes, the descriptors owned by the other processes are not affected. The IPCShutDown call does not operate on the VC socket referred to by a VC socket descriptor unless the descriptor is the *last* descriptor for that socket. A VC socket is destroyed along with its VC socket descriptor *only when the descriptor being released is the sole descriptor for that socket.*

When shutting down a shared call socket descriptor, the call socket referred to by the descriptor is destroyed along with the descriptor and names associated with the descriptor *only if the descriptor being released is the last descriptor for that socket.* If another process, or processes, have descriptors for the same socket, these duplicate descriptors, and any names associated with the descriptors, are not affected.

When shutting down a shared destination descriptor, the addressing information stored in conjunction with the descriptor is destroyed along with the descriptor *only if the descriptor being released is the sole descriptor for that information.* If another process, or processes, have descriptors for the same information, these duplicate descriptors, and any names associated with the descriptors, are not affected.

Signals

Unlike NS/1000 NetIPC calls, LAN/9000 Series 600/800 NetIPC calls that would normally block may be interrupted by HP-UX signals. NetIPC calls that are interrupted by signals are optionally restartable. When a call is restarted after a signal, any time-outs (including the synchronous time-out) will be reset. As a result, signals that continuously interrupt/restart a NetIPC call at an interval shorter than the socket time-out will effectively void the time-out. Signals are explained in detail in the *HP-UX Reference Manual.*

TCP Checksum

The NS/1000 IPCConnect and IPCRecvCn calls include a "checksumming" bit in their *flags* parameters. When set, this bit causes TCP to enable checksumming.

Unlike NS/1000 NetIPC, the LAN/9000 Series 600/800 IPCConnect and IPCRecvCn calls do not include "checksumming" bits. When an NS/9000 Series 600/800 NetIPC process calls IPCConnect or IPCRecvCn, TCP checksumming is *automatically enabled.*

TCP checksumming will *always* be performed if one or both NetIPC processes are LAN/9000 Series 600/800 processes. If both processes are NS/1000 NetIPC processes, TCP checksumming will be performed only if one or both processes call IPCConnect or IPCRecvCn with the "checksumming" bit set.

Remote Process Scheduling

NetIPC itself does not include a call to schedule a remote process. The method used to schedule a remote NetIPC process depends on the types of systems involved. For example, an NS/1000 NetIPC process written to schedule an NS/1000 peer process must be modified to utilize another scheduling method when it is ported to a LAN/9000 Series 600/800 system.

Remote NS/1000 Process

In order to schedule a remote NS/1000 NetIPC process from an NS/1000 node, you can use one of the following methods: the Remote Process Management (RPM) call RPMCreate, the Program-to-Program communication (PTOP) POPEN call, one of the DEXEC scheduling calls, the REMAT QU command, or the TELNET virtual terminal service.

You cannot use any of these services to schedule a remote NS/1000 process from a LAN/9000 Series 600/800 node because these services are only NS/1000 services. The "Process Scheduling" section in the "Cross-System NetIPC" chapter describes ways to schedule an NS/1000 NetIPC process from a LAN/9000 Series 600/800 node.

Remote LAN/9000 Series 600/800 Process

Remote LAN/9000 Series 600/800 processes can be manually started or can be scheduled by user-written daemons that are started at system start up. The "Process

Scheduling" section in "Cross-System NetIPC" chapter describes ways to schedule a LAN/9000 Series 600/800 NetIPC process from an NS/1000 node.

Case Sensitivity

Because the HP-UX operating system is case-sensitive, LAN/9000 Series 600/800 NetIPC call names must be typed using lower case characters. For example, the NetIPC call `IPCConnect` must be typed as `ipcconnect` on LAN/9000 Series 600/800 systems.

NS/1000 NetIPC call names are not case sensitive and may be typed using lower case or upper case characters, or a combination of both upper and lower case characters.

NetIPC Calls

For the purposes of the following discussion, the NS/1000 and LAN/9000 Series 600/800 NetIPC calls are divided into four categories:

- Calls that are **unique to NS/1000 NetIPC.**
- Calls that are **unique to LAN/9000 NetIPC.**
- Calls that are **common** to both NS/1000 and LAN/9000 Series 600/800 NetIPC **and are implemented identically** on each system.
- Calls that are **common** to both NS/1000 and LAN/9000 Series 600/800 NetIPC **but are implemented differently** on each system.

Unique NetIPC Calls

The following calls are provided as part of the NS/1000 NetIPC implementation only:

- **AdrOf**. This call obtains the byte address of any byte within a data object.
- **IPCGet**. This call allows a process to obtain ownership of a call socket, path report or VC socket descriptor that was given away by another process with an IPCGive call.
- **IPCGive**. This call allows a process to "give up" a call socket, VC socket or path report descriptor so that another process may obtain it.

The LAN/9000 Series 600/800 NetIPC implementation includes one call that is not provided by NS/1000 NetIPC:

- **OptOverHead**. This call is used to determine the number of bytes needed for the *opt* parameter.

Common NetIPC Calls

The following NetIPC calls are common to both the NS/1000 and LAN/9000 Series 600/800 NetIPC and are implemented identically.

Table E-1. Identical NetIPC Calls

AddOpt	InitOpt
IPCDest	IPCLookUp
IPCSend	ReadOpt

Call Comparison

Table E-2 lists the differences between the NetIPC calls that are common to both the NS/1000 and LAN/9000 Series 600/800 NetIPC implementations but that are implemented differently.

Table E-2. NS/1000 and LAN/9000 Series 800 Call Comparison

NetIPC Call	Differences Between Implementation
IPCConnect	The NS/1000 implementation of IPCConnect defines a *flags* parameter bit that is not defined by the LAN/9000 Series 600/800 implementation of the call: "checksumming" (bit 21). All LAN/9000 Series 600/800 IPCConnect *flags* parameter bits must be clear (not set). NS/1000 NetIPC processes can enable TCP checksumming by setting the "checksumming" bit. If this bit is not set, TCP checksum will not be performed for the connection unless the process's peer process calls IPCRecvCn with that call's "checksumming" bit set, or the peer process is a LAN/9000 Series 600/800 NetIPC process. TCP checksumming is *always* enabled when the LAN/9000 Series 600/800 implementation of IPCConnect is called. Refer to "TCP Checksum" earlier in this appendix for more information. The LAN/9000 Series 600/800 implementation of IPCConnect allows a value of -1 to be assigned to the call's calldesc parameter. This value causes a call socket to be created and then destroyed after the call completes successfully. The NS/1000 implementation of IPCConnect does not allow this value. The NS/1000 and LAN/9000 Series 600/800 implementations of IPCConnect implement different maximum send and receive sizes. The NS/1000 maximum send and receive sizes are 8,000 bytes; the NS/9000 Series 600/800 maximum send and receive sizes are 32,000 bytes. The default size on both implementations is 100 bytes.

Table E-2. NS/1000 and LAN/9000 Series 800 Call Comparison-con't

NetIPC Call	Diifferences Between Implementation
IPCControl	IPCControl includes four request codes that are not provided by the NS/1000 implementation of the call: 4, 1002, 1003 and 9008. When request code 9008 is specified, the LAN/9000 Series 600/800 implementation of IPCControl allows a value of -1 in the call's *descriptor* parameter; this is also not part of the NS/1000 implementation of the call. Refer to the "NetIPC Calls" chapter in this manual for a description of these request codes. Unlike the NS/1000 implementation of IPCControl, the operation of the LAN/9000 Series 600/800 IPCControl call is affected by socket sharing. Refer to "Socket Ownership" earlier in this appendix for more information about socket sharing. Refer to the "NetIPC Calls" chapter in this manual for a complete description of how socket sharing affects the IPCControl call.
IPCCreate	The NS/1000 and LAN/9000 Series 600/800 implementations of IPCCreate support different ranges of permitted TCP protocol addresses that can be specified in the *opt* parameter. However, both implementations recommend that users specify TCP addresses in the range 30767 to 32767 decimal. The NS/1000 and LAN/9000 Series 600/800 implementations of IPCCreate also support different maximum connection request backlog defaults and ranges. The NS/1000 implementation has a default of three connection requests and an allowable range of zero to five; the LAN/9000 Series 600/800 implementation has a connection request default of one and an allowable range of 1 to 20.
IPCName	The LAN/9000 Series 600/800 implementation of IPCName allows for the naming of destination (also known as path) descriptors. The NS/1000 implementation of the call does not.

Table E-2. NS/1000 and LAN/9000 Series 800 Call Comparison-con't

NetIPC Call	Diifferences Between Implementation
IPCNamErase	Unlike the NS/1000 implementation of IPCNamErase, the operation of the LAN/9000 Series 600/800 implementation of IPCNamErase is affected by socket sharing. Refer to "Socket Ownership" earlier in this appendix for more information about socket sharing. Unlike the LAN/9000 Series 600/800 implementation of IPCNamErase, the operation of the NS/1000 implementation of the call does not allow for erasing names assigned to path report (also known as destination) descriptors.
IPCRecv	The LAN/9000 Series 600/800 implementation of IPCRecv defines bit 26 of the call's *flags* parameter as "more data." This bit is not implemented on NS/1000. When this bit is set on a LAN/9000 Series 600/800, it indicates that non-delimited data was received.
IPCRecvCn	The NS/1000 implementation of IPCRecv includes a *flags* parameter bit that is not defined by the LAN/9000 Series 600/800 implementation of the call: "checksumming" (bit 21). All LAN/9000 Series 600/800 IPCRecvCn *flags* parameter bits must be clear (not set). NS/1000 NetIPC processes can enable TCP checksumming by setting the "checksumming" bit. If this bit is not set, TCP checksum will not be performed for the connection unless the process's peer process called IPCConnect with that call's "checksumming" bit set, or the peer process is a LAN/9000 Series 600/800 NetIPC process. TCP checksumming is *always* enabled when the LAN/9000 Series 600/800 implementation of IPCRecvCn is called. Refer to "TCP Checksum" earlier in this appendix for more information.

Table E-2. NS/1000 and LAN/9000 Series 800 Call Comparison-con't

NetIPC Call	Diifferences Between Implementation
IPCSelect	The LAN/9000 Series 600/800 implementation of IPCSelect allows the *sdbound* parameter to have a maximum value of 60. The NS/1000 implementation has an upper limit of 32.
IPCShutDown	Unlike the NS/1000 of IPCShutDown, the operation of the LAN/9000 Series 600/800 implementation of IPCShutDown is affected by socket sharing. Refer to "Socket Ownership" earlier in this appendix for more information. The HP 9000 supports NSF_GRACEFUL_RELEASE.

Index

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

Printing History

New editions are complete revisions of the manual. The dates on the title page change only when a new edition or a new update is published.

Note that many product updates and fixes do not require manual changes and, conversely, manual corrections may be done without accompanying product changes. Therefore, do not expect a one-to-one correspondence between product updates and manual updates.

Edition 1 . . . . . February 1991

Edition 2 . . . . . July 1992

Printed in USA 07/92 English

Customer Order No.
98194-60532

Manufacturing No.
98194-90032
Mfg. number is for HP internal use o

98194-90032