Migration Guide

NetIPC to BSD Sockets and DSCOPY to FTP

FINAL TRIM SIZE : 7.5 in x 9.0 in

FINAL TRIM SIZE : 7.5 in x 9.0 in

Legal Notice

Hewlett-Packard Company
19420 Homestead Road
Cupertino, CA 95014 U.S.A.

FINAL TRIM SIZE : 7.5 in x 9.0 in

Printing History

New editions are complete revisions of the manual. Updates, which are issued between editions, contain additional and replacement pages to be merged into 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 . . . . . October 1993

In This Guide

This guide provides information about how to migrate NetIPC client/server applications to Berkeley Software Distribution (BSD) Sockets and about how to migrate from DSCOPY/9000 to FTP/9000. This guide also describes how to port UNIX BSD Socket applications to the HP 3000 MPE/iX platform. Although both AF_UNIX and AF_INET are address families used with BSD Sockets, this guide only discusses the AF_INET address family.

Chapter 1	**Introduction** This chapter explains the purpose of migration and presents a basic comparison of NetIPC and BSD Sockets.
Chapter 2	**NetIPC to BSD IPC Migration** This chapter provides the details of NetIPC to BSD IPC migration.
Chapter 3	**NetIPC and BSD IPC Communication** This chapter explains how MPE/iX NetIPC applications communicate with BSD IPC applications.
Chapter 4	**BSD IPC Porting** This chapter describes the procedures for porting BSD IPC applications to HP 3000 systems.
Chapter 5	**DSCOPY/9000 to FTP/9000 Migration** This chapter provides information about migrating from DSCOPY to FTP.
Appendix A	**NetIPC Sample Programs** This appendix provides examples of working NetIPC client and server programs.
Appendix B	**BSD IPC Sample Programs** This appendix provides examples of working BSD IPC client and server programs.
Appendix C	**NetIPC Sample Include File** This appendix provides an example of an HP 3000 NetIPC include file.

FINAL TRIM SIZE : 7.5 in x 9.0 in

Contents

Figures

Tables

1

Introduction

Introduction

To prevent unnecessary maintenance of different versions of applications for different vendor's platforms, Hewlett-Packard provides the components, tools, and solutions for application developers to move from proprietary to open client/server solutions. To accomplish this, HP encourages application developers to transition from HP's existing proprietary client/server Application Programmatic Interfaces (APIs) to HP's standards based client/server tools on HP 9000 systems, HP 3000 MPE/iX systems, HP 1000 RTE systems, and PCs.

HP recommends that you migrate all existing NetIPC applications on HP 9000 and PC platforms to BSD Sockets. For the PC windows-based platform, you should migrate NetIPC and WSOCKETS implementations to Microsoft's WINSOCK interface. HP also recommends that all new applications on HP 3000 MPE/iX and HP 1000 platforms use BSD Sockets; however, HP does not recommend migration to BSD Sockets for existing NetIPC applications on HP 3000 MPE/iX and HP 1000 platforms.

By migrating the recommended applications from the proprietary NetIPC API to the BSD Sockets API, you can take advantage of the following benefits.

- Lower development costs.

  BSD Sockets improve application portability. This allows support of a wider range of systems in multi-vendor environments without maintaining several different versions of your applications for each vendor's platform.

- Increased application access.

  Applications based on BSD Sockets can be accessed from anywhere on the network, providing increased accessibility to better meet your needs. The NetIPC `ipclookup()` call uses the proprietary PXP protocol, which is not supported on all routers.

- Reduced configuration requirements.

  Applications based on BSD Sockets fit into your existing multi-vendor environments. NetIPC requires that a proprietary naming scheme be configured and maintained in addition to industry standard domain naming used in multi-vendor environments.

- Reduced support costs.

  BSD applications use standard protocols. This makes troubleshooting easier. Additionally, the BSD Sockets API is an industry standard with reference books and courses available from a variety of different sources.

The intent of this guide is to make the transition to standards based client/server APIs easier for application developers. In some cases, your applications may have to communicate with NetIPC applications running on MPE-based or RTE-based systems. For this reason, this guide includes interoperability information about how NetIPC and BSD IPC applications communicate with each other.

Before You Begin

Before you begin the process of migrating your applications, you may need to review the following sections:

- Additional references.
- NetIPC to BSD Sockets call mapping.
- The Socket Registry.

Additional References

You may need to refer to one or more of the following manuals in addition to the information contained in this guide:

HP 9000 Manuals

- *HP 9000 Networking Berkeley IPC Programmer's Guide*
- *HP 9000 Networking NetIPC Programmer's Guide*

HP 3000 Manuals

- *NetIPC 3000/XL Programmer's Reference Manual*
- *BSD Sockets/iX Programmer's Guide*

HP 1000 Manuals

- *NS-ARPA/1000 User/Programmer Reference Manual*
- *NS-ARPA/1000 BSD IPC Reference Manual*

PC Manuals

- *PC Sockets Programmer's Guide*
- *PC NetIPC Programmer's Guide*

NetIPC to BSD Sockets Call Mapping

NetIPC and BSD Sockets are simply interfaces to the TCP/IP transport protocol; they are not end-to-end protocols in themselves. As a result, BSD applications can communicate with NetIPC applications. While a one-to-one mapping of NetIPC and BSD calls is not possible, both NetIPC and BSD Sockets provide similar functionality.

The following table shows the relationship between NetIPC and BSD Sockets calls. Mapping does not imply that a one-to-one replacement of BSD Sockets calls is sufficient to migrate from a NetIPC application to a BSD application. Rather, the entire program must be examined and may have to be altered depending on the circumstances.

Table 1-1. NetIPC to BSD Sockets Call Mapping

NetIPC Calls		BSD Sockets Calls

`ipccreate()`	maps to	`socket()`
		`bind()`
		`listen()`

`ipcrecvcn()`	maps to	`accept()`

`ipcdest()`
`ipclookup()`
`ipcconnect()`
	maps to	`connect()`

`ipcsend()`	maps to	`send()` `sendto()`
`ipcrecv()`	maps to	`recv()` `recvfrom()`
`ipcshutdown()`	maps to	`shutdown()` `close()`

		`getsockname()`
		`getpeername()`
		`socketpair()`

		File system call
		`fcntl()`
		`ioctl()`

`iowait()`	maps to	`select()`

The Socket Registry

A NetIPC feature that has no direct BSD Sockets equivalent is the facility for named sockets, called the Socket Registry. The Socket Registry enables users to dynamically name sockets and register them, so other users can find the socket by name.

The system calls to access and modify this registry are `IPCLOOKUP()` and `IPCNAME()`. A server program, for instance, could allocate a socket without specifying a port number to get a random port, name the socket, and register it in the Socket Registry by calling `IPCNAME()`. Now the client does not need to know the specific port number of the server. Instead, the client simply calls `IPCLOOKUP()` with the name of the socket and the name of the remote machine.

The closest BSD sockets equivalent is the `getservbyname()` function. `getservbyname()`, combined with `gethostbyname,` provides the functionality of `IPCLOOKUP()`. Because it does not provide the functionality of `IPCNAME()`, however, `getservbyname()` is not dynamic. To name a BSD socket, you must modify the flat ASCII file `/etc/services` (`SERVICES.NET.SYS` on MPE/iX) to include an entry for the socket on the client and server systems. Once these entries are made, the server calls `getservbyname()` to get the specific port number and then calls `bind()` to bind to that port address. The client uses `getservbyname()` to get the specific port and then calls `connect()` to that port.

2

NetIPC to BSD IPC Migration

NetIPC to BSD IPC Migration

Migration Overview

The following is a brief overview that outlines the process of migrating NetIPC client/server applications to BSD Sockets.

- Install and configure ARPA Services on your HP 9000 system or your PC.

  For information on installing ARPA Services/9000 refer to *Installing and Administering ARPA Services*. For information on installing PC ARPA Services, refer to the *PC Sockets Programmer's Guide*.

- Add the server's hostname to the `/etc/hosts` file (`HOSTS.NET.SYS` file for MPE/iX) on the client system or configure the client to use a Domain Name Server (DNS).
- Test connectivity to the server using the *ping* command (*PING.NET.SYS* for MPE/iX).
- If you are using an HP 3000 or HP 1000 NetIPC client or server application, you will need to modify it to use a well-known port.

  Refer to chapter 4 for more information about creating and connecting to well-known ports.

- If the server is an HP 3000 system, configure Ethernet support.
- Convert the client/server application from NetIPC to BSD IPC using the information contained in this chapter.
- Test the BSD client program with the NetIPC or BSD server.

NOTE

For MPE/V-based client/server applications, you must continue to use NetIPC. Existing HP 3000 MPE/iX and HP 1000 applications should also continue using NetIPC. HP recommends that HP 9000, PC, new HP 3000, and new HP 1000 applications move to BSD sockets applications. Interoperability between NetIPC and BSD Sockets applications is supported for backward compatibility.

Configuration Considerations

To allow client/server applications to run on different platforms, you must consistently configure both the client and server systems. The following sections describe the required configuration for converting a NetIPC client/server application to BSD Sockets.

Name to IP Address Resolution

Client applications typically identify systems using a hostname. Prior to communicating with the host, the client must convert the hostname to an IP address before the connection with the remote host can be established. This is accomplished differently in NS and BSD. With NS, the hostname to IP address mapping is accomplished using either Probe, Probe Proxy, or the HP 3000 Network Directory. With BSD Sockets, the hostname to IP address mapping is accomplished by configuring hostnames and IP addresses in the `/etc/hosts` file (`HOSTS.NET.SYS` for MPE/iX).

Alternatively, you can configure the system to use the Domain Name System (DNS). In this case, the `/etc/hosts` file (`HOSTS.NET.SYS` for MPE/iX) will be bypassed and the hostname mapping requests will be sent to a domain name server. The domain name server must have the requested hostname configured to return its IP address. The DNS alternative used is transparent to the BSD application. The BSD client application uses the BSD *gethostbyname()* call to resolve the hostname to IP address regardless of which method is used. The `/etc/hosts` file (`HOSTS.NET.SYS` for MPE/iX) approach is recommended for small networks and the domain name server approach is recommended for medium to large networks.

On an HP 9000 system, the *nodename* command shows the NS nodename. This name is distinct from hostname which is a Internet Domain term. All the BSD services including BSD IPC recognize hostnames and not nodenames. Usually the HP-only nodename is the same as the hostname of a HP host, but this is not necessarily true. You should always check the correctness of the hostname by looking up the name in the `/etc/hosts` file on the HP 9000

system (`HOSTS.NET.SYS` for MPE/iX) or by using the *nslookup* command on an HP 9000 system to determine if a name server is used.

Port Name to Port Address Resolution

Client applications identify a service residing on a remote host using a service name. The service name must be converted to a port address before the request is sent to the host.

With NetIPC, the server application registers a server name by calling *ipcname()*. The NetIPC client applications connect to the server by calling *ipclookup()*, without having to know the port address of the server. The server name-to-address mapping information is stored in the Socket Registry. Alternatively, the NetIPC server application can create a port address (well-known port) using the *ipccreate()* call. The NetIPC client applications connect these well-known ports using the *ipcdest()* call. With BSD Sockets, the server application is assigned a unique service name and port address. The service name and port address must be configured in each client's `/etc/services` file (`SERVICES.NET.SYS` for MPE/iX). The BSD client application calls *getservbyname()* to resolve the service name-to-address mapping. The *getservbyname()* call will use the `/etc/services` file (`SERVICES.NET.SYS` for MPE/iX) on the client system.

The well-known address must be unique. For HP 3000 and HP 1000 systems, the port addresses available to non-privileged users are in the range of 1024 through 32767 (decimal). For HP 9000 systems, the port addresses available for non-super user access are in the range of 1024 through 65535 (decimal). To ensure interoperability between different platforms, you should use addresses in the range of 30767 through 32767 (decimal).

IP to LAN Address Resolution

NetIPC uses the proprietary Probe protocol to resolve IP addresses and may optionally use the industry standard ARP protocol. Begining with MPE/iX Release 4.0, the proprietary Probe protocol can be turned off, allowing only the ARP protocol to be used to resolve IP addresses. BSD Sockets use the industry standard ARP protocol. When NetIPC clients are converted to use BSD Sockets, you must configure the client and server systems to use the ARP protocol. For HP 9000 and HP 3000 systems to support the ARP protocol, you should configure Ethernet. For HP 1000 systems, 802 uses the Probe protocol only, Ethernet uses the ARP protocol only, and LAN uses both Probe and ARP protocols.

Overall Socket Differences

Sockets are a generic term used in NetIPC and BSD IPC to describe an endpoint for network communications. In NetIPC, there is a distinct difference between the connecting **call sockets** (referenced by source and destination descriptors) and the **Virtual Circuit (VC) sockets** (referenced by connection descriptors). In BSD IPC, all the sockets involved are simply **BSD sockets**.

NetIPC has **VC connection descriptors** for referencing local VC sockets and **destination descriptors** for carrying all the addressing information required to connect to a remote call socket. BSD IPC has only one form of descriptors, namely **socket descriptors**.

With NetIPC, establishing the client end of a connection requires a minimum of two socket descriptors: one destination descriptor and one VC socket descriptor. Establishing the server end of a connection with NetIPC requires a minimum of three socket descriptors: one call socket descriptor, one destination descriptor, and one VC socket descriptor. With BSD IPC, setting up the client end of a connection requires only one socket, and setting up the server end of a connection requires only two socket descriptors: one for the *listen()* call and one for each connection established.

The maximum number of file and socket descriptors owned by an HP 9000 process at any given time is 2048 (total). The HP 3000 maximum is 1024 (total), the PC maximum is 21 (total), and the HP 1000 maximum is 31 (total). These maximum numbers include call socket, VC socket, and open file descriptors and are independent of the number of open files.

Include Files

NetIPC and BSD Sockets use different include files. The following lists the include files used for each type of application.

NetIPC Include Files

NetIPC applications use the following include files:

HP 9000
```
#include <sys/ns_ipc.h>
```

HP 3000
```
#include <sys/ns_ipc.h>
```
See Appendix C for an example.

HP 1000 None

PC
```
#include <mscipc.h>
```

BSD IPC Include Files

BSD IPC applications use the following include files:

HP 9000
```
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/fcntl.h>
#include <netinet/in.h>
#include <netdb.h>
```

HP 3000

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <sys/ioctl.h>
#include <sys/file.h>
#include <sys/errno.h>
#include <unistd.h>
#include <fcntl.h>
#include <time.h>
```

HP 1000

```
#include <types.h>
#include <socket.h>
#include <in.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <netdb.h>
#include <string.h>
#include <time.h>
#include <socket.ftni>
#include <socket.pasi>
#include <extcalls.pasi>
```

PC

```
#include <sys\socket.h>
#include <winsock.h>
#include <netinet\in.h>
#include <netdb.h>
#include <sock_err.h>
```

Setting Up Connections

When calling *ipcdest()* or *ipclookup()*, you must use the NS nodename for the location parameter on an HP 9000 system. An HP 3000 MPE/iX system allows you to use an Internet hostname. NetIPC uses the Probe protocol or the HP 3000 Network Directory to resolve hostname to IP address. Probe proxy can also be used to accomplish address resolution. On an HP 3000 running MPE/iX 4.0 or later, you can disable the Probe protocol. This means that all location parameters in the *ipcdest()* and *ipclookup()* calls will be resolved using DNS or `HOST.NET.SYS`.

When creating a BSD IPC socket, you must specify the remote host by its **Internet hostname** or **Internet address** (IP address). If you specify an Internet hostname, the *gethostbymame()* function is called to return the IP address information. The name-to-address resolution is done by looking up the hostname in the `/etc/hosts` file (`HOSTS.NET.SYS` for MPE/iX). When you specify a name server (except on HP 1000 systems), the routine will use the Domain Name System (DNS) protocol to resolve the hostname.

Call Mapping

The following explains how NetIPC and BSD Sockets calls "map" for setting up connections. Here is an overview of the differences.

- The socket kind parameter (`NS_CALL`) of the *ipccreate()* call "maps" to the type parameter (`SOCK_STREAM`) of the BSD *socket()* call.
- The protocol parameter (`NSP_TCP`) of the *ipccreate()* call "maps" to the protocol parameter of the BSD *socket()* call. Since `SOCK_STREAM` implies a TCP connection, the last argument about protocol in *socket()* is supplied with a zero.
- The flags parameter of the *ipccreate()* call is not defined for PC NetIPC and should be replaced by 0 or NULL.

- The opt parameter of the *ipccreate()* call "maps" to the address parameter of the BSD *bind()* call. To ensure the interoperability between different platforms, you should use addresses in the range of 30767 through 32767 (decimal).
- The location parameter (`nodename`) of the *ipcdest()* call "maps" to the name parameter of the BSD *gethostbyname()* call. You can also use the BSD *gethostbyaddr()* call if the IP address is given or explicitly fill in the `sockaddr_in` structure.
- The prot_addr parameter of the *ipcdest()* call "maps" to the service port address of *getservbyname*.
- The destination descriptor parameter of the *ipcdest()* call "maps" to the hostent and servent structures returned by the BSD *gethostbyname()* and *getservbyname()* calls.
- For PC BSD applications, the hostent and servent structures must be defined as far pointers:

```
struct hostent far *hp;
struct servent far *sp;
```

ipcname()

On a NetIPC server, the *ipcname()* call is used to register a socket name in the Socket Registry (see chapter 1). Since BSD IPC applications have no access to the Socket Registry, you should create a well-known port with *ipccreate()* on the HP 3000 server.

NetIPC Example

```
descriptor_type call_sd; /* call socket
descriptor */
char *socketname;
int result;

socketname = "ABCDEFGH";
ipcname (call_sd, socketname, 8, &result);

/* Also see the discussion below on ipccreate(). */
```

ipclookup()

On a NetIPC client, the *ipclookup()* call is used to look up the socket name in the server's Socket Registry (see chapter 1). It is similar to the combination of the BSD *gethostbyname()* and *getservbyname()* calls. The *getservbyname()* call returns a structure containing the port address of the service. The following examples assume that the service name and the well-known port are configured together in the `/etc/services` file (`SERVICES.NET.SYS` for MPE/iX) on the client node.

NetIPC Example

```
descriptor_type desc_sd;
char *socketname, *nodename;
int flags, protocol, socket_kind, result;

socketname = "ABCDEFGH";
nodename = "REMOTE";
flags = 0;
ipclookup (socketname, 8, nodename,
strlen(nodename), &flags, &desc_sd, &protocol,
&socket_kind, &result);

/* Also see the discussion below on ipccreate(). */
```

BSD Example

```
struct hostent hp;
struct servent sp;
char *host_name, *serv_name;
host_name = "REMOTE";

hp = gethostbyname(host_name);

sp = getservbyname(serv_name, proto);
```

ipccreate()

On a NetIPC server, the *ipccreate()* call is used to create a source socket descriptor. It is similar to the BSD *socket()* and *bind()* calls. *socket()* creates a BSD socket and returns the socket descriptor for the socket. On a server the socket is usually bound to a well-known address through the *bind()* call. The server then listens at the address. The *bind()* call is optional on the client side. If omitted, the bind function will be performed by the *connect()* call.

NetIPC Example

```
flag_type flags;
opt_type opt;
descriptor_type call_sd;   /* call socket descriptor */
int opt_num_argument, result;

opt_num_arguments = 2;
initopt(opt, opt_num_arguments, &result);

opt_data = TCP_WELL_KNOWN_PORT;  /* a hardcoded constant */
addopt(opt, 0, NSO_PROTOCOL_ADDRESS, 2, &opt_data, &result);
opt_data = MAX_BACKLOG;  /* a hardcoded constant; max = 5 */
addopt(opt, 1, NSO_MAX_CONN_REQ_BACK, 2,
                  &opt_data, &result);

flags = 0;
ipccreate(NS_CALL, NSP_TCP, &flags, opt, &call_sd, &result);
```

BSD Example

```
int lsd;     /* listen socket descriptor */
int result;
struct sockaddr_in myaddr_in;
int backlog;

myaddr_in.sin_family = AF_INET;
myaddr_in.sin_addr.s_addr = INADDR_ANY;
              /* any remote hosts */
myaddr_in.sin_port = TCP_WELL_KNOWN_PORT;

lsd = socket(AF_INET, SOCK_STREAM, 0)
result = bind(lsd, &myaddr_in, sizeof(struct sockaddr_in));
backlog = 5
result = listen(lsd, backlog);
```

ipcdest()

On a NetIPC client, the *ipcdest()* call is used to create a destination descriptor on a call to *ipcconnect()*. This destination descriptor contains the information necessary to connect to a remote socket (IP address and port address). For BSD Sockets, the `sockaddr_in` structure contains similar information that you must explicitly fill in.

NetIPC Example

```
#define TCP_WELL_KNOWN_PORT     32000
char *node_name;  /* remote node name */
int flags, result;
prot_addr;
descriptor_type *dd;

flags = 0;
prot_addr = TCP_WELL_KNOWN_PORT;
node_name = "REMOTE";

ipcdest(NS_CALL, node_name, strlen(node_name),
     NSP_TCP, (char *) &prot_addr,
     sizeof(prot_addr), &flags, opt, &dd,
     &result);
```

BSD Example

```
struct sockaddr_in     peeraddr;
```

You must fill in the `sockaddr_in` structure prior to the call to connect. You should set the address family in the `sockaddr_in` structure to AF_INET (this is defined in `SYS/SOCKET.h`. Here is an example.

```
peeraddr.sin_family = AF_INET
```

The IP address could be hardcoded, but is typically obtained by *gethostbyname()*. Here is an example.

```
hp = gethostbyname(host_name);

peeraddr.sin_addr.s_addr = ((struct in_addr *)
     (hp->h_addr))->s_addr;
```

The port could also be hardcoded or obtained by *getservbyname()*. Here is an example.

```
peeraddr.sin_port = sp -> s_port;

OR

peeraddr.sin_port = TCP_WELL_KNOWN_PORT;
```

optoverhead()

The NetIPC *optoverhead()* call returns the number of bytes required in the option record. This feature is not required in applications that use BSD Sockets and should be removed.

initopt()

The NetIPC *initopt()* call initializes the option record. This feature is not required in applications that use BSD Sockets and should be removed.

addopt()

The NetIPC *addopt()* call adds an option to the option record. It is typically used to add a well-known port address to the option record. This feature is not required in applications that use BSD Sockets and should be removed.

readopt()

The NetIPC *readopt()* call reads the specified entry from an option record. This feature is not required in applications that use BSD Sockets and should be removed.

Establishing Connections

To start a connection, the NetIPC server must create a call socket. The socket can be created for a well-known address (port) or registered with a name in the Socket Registry. A client looks up the socket name and uses the protocol and destination address information to establish a connection.

With BSD IPC, when a socket is created on a server, it is bound to a well-known port. Clients establish connection by sending connection requests to the port.

After a NetIPC socket is created, by default, it is placed in synchronous (block) mode, that is, the calling process will be blocked if requests cannot be immediately satisfied. A blocked process will remain suspended until the request is satisfied, a signal arrives, an error occurs, or an asynchronous timeout occurs. NetIPC sockets have timeout values associated with them. The default timeout value is 60 seconds and can be set by users. The timeout values may be changed through *ipccontrol()*. You can set the value so NetIPC calls will be blocked indefinitely, if necessary. You can also set a shorter or longer value so NetIPC calls will timeout and return a *SOCKET_TIMEOUT* error when some condition cannot be fulfilled.

BSD Sockets are also created in blocking mode by default. BSD Sockets do not have a timeout value associated with them. A *recv()* will block until there is a message to be received, and a *send()* will block until there is a message to transmit. The BSD socket calls either block indefinitely (*blocked* I/O mode) or fail if some conditions cannot be satisfied immediately (*non-blocked* I/O mode). Blocked or non-blocked mode can be set by users with the *ioctl()* or *fcntl()* calls.

Call Mapping

The following explains how NetIPC and BSD Sockets calls "map" for establishing connections. Here is an overview of the differences.

- The source socket descriptor parameter of the *ipcconnect()* call "maps" to the socket descriptor parameter of the BSD *connect()* call.
- The destination socket descriptor parameter of the *ipcconnect()* call "maps" to the server address parameter of the BSD *connect()* call.
- The equivalent to the connection descriptor parameter of the *ipcconnect()* call is contained in the socket descriptor parameter of the BSD *connect()* call.
- The `hp` and `sp` pointers refer to the structures returned by *gethostbyname()* and *getservbyname()* described under *ipcdest()*.
- With NetIPC, the server is listening at the call socket. It returns a VC socket when it accepts the connection request from the client. With BSD IPC, the server explicitly calls *listen()*. When it returns, the server calls *accept()* to accept the client connection request.
- With NetIPC, the *ipccontrol()* call can be used by the server application to identify the nodename and TCP port of a client requesting a connection. With BSD Sockets, this information is provided to the server application at the time connection is accepted with *accept()*.

ipcconnect()

On the client, the *ipcconnect()* call is used to establish a virtual circuit between the source descriptor and the socket described by the destination descriptor. It is equivalent to the BSD *connect()* call.

NetIPC Example

```
descriptor_type  sd, dd, cd;
int result;

ipcconnect(sd, dd, NULL, NULL, cd, &result);
```

BSD Example

```
struct hostent hp;
int sd, addr_len, result;
struct sockaddr_in server_addr;
char *host_name

hp = gethostbyname(host_name);

server_addr.sin_family = AF_INET;
server_addr.sin_addr.s_addr =
    ((struct in_addr *)(hp->>h_addr))->>s_addr;
server_addr.sin_port = sp->>s_port;

result = connect(sd, &server_addr,
     sizeof(struct sockaddr_in));
```

ipcrecvcn()

On the server, the *ipcrecvcn()* call is used to wait for connection requests from clients after a call socket has been created. The corresponding BSD IPC call is *accept()*. *accept()* may be called after a *listen()* has been performed on the socket.

NetIPC Example

```
descriptor_type call_sd, vc_sd;
int opt_num_arguments, flags, result;

opt_num_arguments = 0;
initopt(opt, opt_num_arguments, &result);

flags = 0;
ipcrecvcn(call_sd, &vc_sd, &flags, opt, &result);
```

BSD Example

```
descriptor_type lsd, vc_sd;
struct sockaddr_in client_addr;

listen(lsd, MAX_BACKLOG);

memset ((char *)&client_addr, 0,
sizeof(sturct sockaddr_in));
     /* to clear client address struct */

vc_sd = accept(lsd, &client_addr,
sizeof(struct sockaddr_in));
     /* client address info now in &client_addr */
```

When *accept()* returns, the client's address information is readily available in the client_addr parameter.

ipccontrol()

The *ipccontrol()* call is used to modify connection characteristics. It is similar to the BSD *ioctl()* and *setsockopt()* calls.

NetIPC Example

```
descriptor_type call_sd;
int flags, timeout, result;

flags = 0;
timeout = 0; /* block mode; wait indefinitely */

ipccontrol(call_sd, NSC_TIMEOUT_RESET,
&timeout, 2, NULL, NULL, &flags, &result);
```

BSD Example

```
int sd, flag, result;

flag = 1;
result = ioctl(sd, FIOSNBIO, &flag);
     /* block mode; wait indefinitely */
```

Transferring Data

For HP 3000 applications, the NetIPC *ipcrecv()* and *ipcsend()* calls correspond to the BSD *recv()* and *send()* calls (respectively). With the MPE/iX 4.5 release, this functionality is also performed by the BSD *read()* and *write()* calls. For HP 9000, HP 1000, and PC applications, NetIPC has the ability to wait for a specified amount of data to receive, while the BSD *recv()* call will return as long as more than 1 byte of data transferred on a socket.

BSD IPC supports partial data transfer. For example, when you call *send()* to transfer 1000 bytes and at that time the socket buffer has only 800 bytes of space in it, then 800 bytes will be transferred first. The *send()* call will either block awaiting space for the remaining 200 bytes in BLOCKED MODE, or will return immediately to the user in NON-BLOCKED MODE to indicate that the first 800 bytes have been transferred.

NetIPC does not support this partial data transfer policy; data is transferred all at once. Using the same example, NetIPC will not transfer a byte of data until either timeout occurs or 1000 bytes of space become available in the socket buffer This behavior may result in a degradation of the NetIPC throughput when the message size is greater than half of the socket buffer size. To be more efficient, you can set the socket buffer size to double the largest message being sent for NetIPC sockets. This is not necessary for BSD IPC sockets.

With NetIPC, the HP 9000 and HP 1000 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. The PC range is 1 to 65,535 bytes. A socket buffer size should be specified within the correct range for the respective system.

Call Mapping

The following explains how NetIPC and BSD Sockets calls "map" for transferring data. Here is an overview of the differences. The code examples assume that you are using blocked mode.

- By default *ipcrecv()* blocks for 60 seconds while *recv()* blocks indefinitely. However, *recv()* users can have timeouts by enabling signals and using the *alarm()* function.
- In non-blocked mode, the *select()* call can be used to detect the readiness of sending the next segment of data.
- With BSD IPC on HP 9000 systems, the flags parameter is set to `MSG_PEEK` (for peeking incoming data), `MSG_OOB` (for sending out-of-band data), both, or zero. On HP 3000 systems running MPE/iX 4.5 or later and on HP 1000 systems, the flags parameter is set to `MSG_PEEK` or zero.
- The *select()* bitmap mask in NetIPC is an array of integers. BSD IPC has a pre-defined structure and several macros to manipulate bitmap masks. HP recommends that you use the existing macros.
- With BSD IPC, *select()* connection requests are treated as reads. They are detected through the read mask. With NetIPC, connection requests are notified through the exception mask. NetIPC on the HP 3000, however, also uses the read mask.

ipcrecv()

The *ipcrecv()* call is used to receive a response to a connection request and to receive user data on a connection. Its equivalents in BSD are *recv()*, *recvfrom()*, and *recvmsg()*. Only *recv()* will be presented in the code examples.

NetIPC Example

```
descriptor_type sd;
char data_buff[BUFF_LEN+1];
unsigned int buff_len;
flags_type flags;
opt_type opt;
int opt_num_arguments, result, time, request;

opt_num_arguments = 0;
initopt(opt, opt_num_arguments, &result);

flags = 0;
buff_len = BUFF_LEN;
ipcrecv(sd, data_buff, &buff_len,
     &flags, opt, &result);
```

BSD Example

```
int sd;
char data_buff[BUFF_LEN+1];
char *buff_ptr;
int amount_to_recv, amt_recvd, flags;

amount_to_recv = BUFF_LEN;
buff_ptr = data_buff[0];
amt_recvd = 0;
while (amount_to_recv > 0)
{
     flags = 0;
     amt_recvd = recv(sd,
         buff_ptr, amount_to_recv, flags);
     amount_to_recv -= amt_recvd;
     buff_ptr += amt_recvd;
}
date_buff[BUFF_LEN] = '\0';
```

ipcsend()

The *ipcsend()* call is used to transmit data on a connection. Its equivalents in BSD are *send()*, *sendto()*, and *sendmsg()*. Only *send()* will be presented in the code examples.

NetIPC Example

```
descriptor_type sd;
data_buffer data_buff;
unsigned int buff_len;
int result

flags = 0;
ipcsend(sd, data_buff, buff_len,
     &flags, opt, &result);
```

BSD Example

```
int flags, num, sd, buff_len;
char data_buff[BUFF_LEN+1];

flags = 0;
num = send(sd, data_buff, buff_len, flags);
```

ipcselect()

On the server, the *ipcselect()* (*select()* for MPE/iX and also HP 9000) call is used to provide a synchronous multiplexing mechanism. The *ipcselect()* call returns when at least one socket is ready for reading or writing or the socket has received a connection request. It is equivalent to the BSD *select()* call.

NetIPC Example

```
/* The following shows an implementation of
selecting from a maximum of 60 sockets. */
/* The arrays form the bitmap masks for keeping track
of read, write, and connection request conditions. */
int rmap[2], wmap[2], xmap[2];
/* Working bitmap masks */
int curr_rmap[2], curr_wmap[2], curr_xmap[2];
short timeout;
int i, offset, sbound, result;

for (i = 0; i << 2; i++) {
curr_rmap[i] = rmap[i];
curr_wmap[i] = wmap[i];
curr_xmap[i] = xmap[i];}

timeout = -1;
/* indefinite wait; block mode select*/
ipcselect(&sbound, curr_rmap, curr_wmap,
     curr_xmap, timeout, &result);

/* If some socket is ready for read */
if ((curr_rmap[0] || curr_rmap[1])) {
/* Find out which bit is set in the returned
     bitmap mask */
for (offset = 0; offset << sbound; offset++) {
     if (curr_rmap[ofset/32] &
        ((unsigned int)0x80000000 >> (offset % 32))) {
     ...}}}
```

BSD Example

```
/* The following code segment uses the pre-defined
structs and macros to manipulate bitmap masks. */
int offset, result;
int sbound;
struct fd_set read_mask, write_mask;
struct fd_set curr_read_mask, curr_write_mask;

curr_read_mask = read_mask;
curr_write_mask = write_mask;

/* indefinite wait; block mode select */
result = select(sbound, &curr_read_mask,
     &curr_write_mask, 0, (struct timeval *) 0);

for (offset = 0; offset << nfds; offset++)
{
     if (FD_ISSET(offset, &curr_read_mask)) {
     ...}
     else if (FD_ISSET(offset, &curr_write_mask)) {
     ...}}
```

Terminating Connections

When you call *ipcshutdown()*, all the data remaining in the socket queue may be lost without notice. To ensure that no data is lost during connection shutdown, you must specify the NSF_GRACEFUL_RELEASE flag.

BSD IPC supports the *shutdown()* system call on HP 9000, HP 3000, and HP 1000 systems. However, BSD IPC *shutdown()* is not equivalent to NetIPC *ipcshutdown()*. This *shutdown()* call provides a means to stop either sending or receiving data or both. It does not shut down a connection. NetIPC does not provide a facility equivalent to BSD IPC *shutdown()*.

In a BSD IPC application, use *close()* to shut down a connection. *close()* decrements the file descriptor reference count. When the last *close()* is executed on a socket descriptor, any data not previously sent are transferred before the socket is closed. This is called "graceful close," however, any unreceived data may be lost.

BSD IPC also supports abrupt close, all data not previously sent are immediately lost. To control the closing actions, use the *setsockopt()* call to turn on the SO_LINGER value with a zero or nonzero timeout value. The SO_LINGER option, however, is not available on the HP 1000. A zero timeout interval is for abrupt (hard) close. A nonzero value is to block the *close()* call for the specified time for "graceful close."

Call Mapping

The following explains how NetIPC and BSD Sockets calls "map" for terminating connections.

ipcshutdown()

The *ipcshutdown()* call is similar to the BSD IPC *close()* call. For PC BSD sockets, use the *close_socket()* call.

NetIPC Example

```
flags = 0;  /* or flags = NSF_GRACEFUL_RELEASE */
ipcshutdown(sd, &flags, &result);
```

BSD Example

```
close(sd);
```

To use the SO_LINGER option in BSD IPC, call the following after a BSD socket is created:

```
struct linger linger = {1, 1};
/* set linger flag & graceful disconnect */
setsockopt(sd, SOL_SOCKET, SO_LINGER, &linger,
sizeof(linger));
```

With BSD IPC, a connection can be gracefully disconnected or abruptly disconnected. Use *setsockopt()* to achieve this. You can also use it to modify the socket characteristics. *getsockopt()* retrieves the socket characteristics. Here is an example.

```
int sd, result;
struct linger linger = {1, 1};

result = setsockopt(sd, SOL_SOCKET,
SO_LINGER, &linger, sizeof(linger));
```

Byte Order Conversion Routines

HP RISC-based and Motorola 680X0-based computers process bytes in standard TCP/IP byte order, known as network byte order. PCs, however, use Intel architecture and process bytes in reverse order, known as host byte order.

PC programmers, therefore, must be aware of this difference, especially when filling in socket address structures. The PC socket developer's kits provide library routines for converting between the two architectures. To ensure source code portability, programmer's should use the following calls.

ConvertNetworkLong()

The *ConvertNetworkLong()* call is similar to the BSD *htonl()* and *ntohl()* calls.

NetIPC Example

```
long num;
ConvertNetworkLong(num);
```

BSD Example

```
long num;
htonl(num);
```

ConvertNetworkShort()

The *ConvertNetworkShort()* call is similar to the BSD *htons()* and *ntohs()* calls.

NetIPC Example

```
short num;
ConvertNetworkShort(num);
```

BSD Example

```
short num;
htons(num);
```

3

NetIPC and BSD IPC Communication

NetIPC and BSD IPC Communication

Figure 3-1 illustrates how a BSD Sockets application communicates with a NetIPC application.

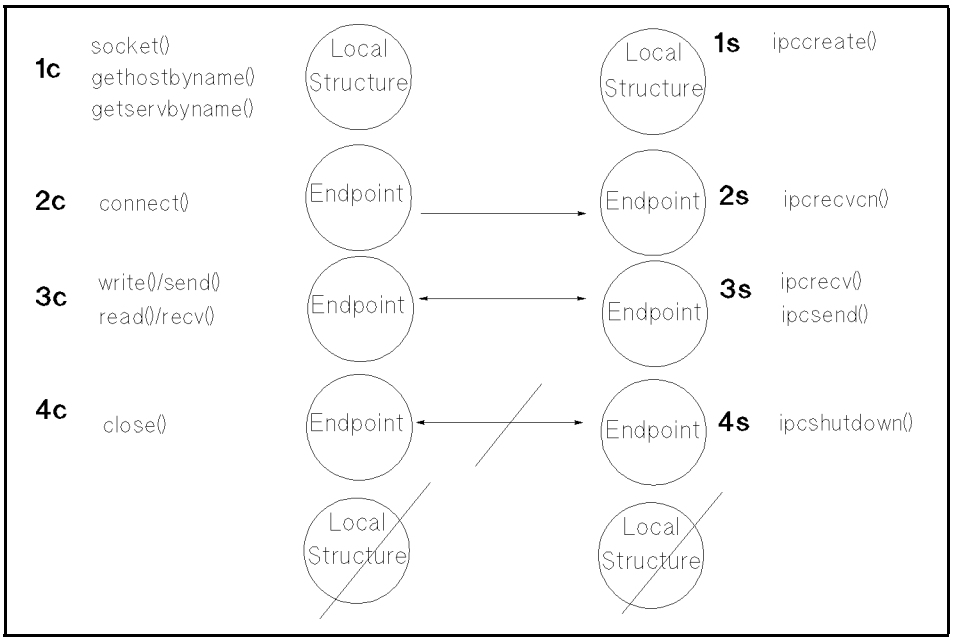

Figure 3-1. BSD Sockets to NetIPC Communication

BSD IPC Client and NetIPC Server

The NetIPC server must create a well-known port rather than relying on *ipcname()* which uses the Socket Registry. The BSD client will not be able to find the NetIPC server if the server relies on *ipcname()*, since BSD Sockets cannot access the Socket Registry. The NetIPC server creates its socket at a well-known port during the *ipccreate()* call.

The *connect()* call from the BSD client may or may not return successfully before *ipcrecvcn()* is called on the NetIPC server side, depending on the server system being used. If the NetIPC server system is an HP 9000 or HP 1000, then *connect()* will return successfully even before *ipcrecvcn()* is called. On an HP 3000, the *connect()* call will block (in blocked mode) until *ipcrecvcn()* has been called on the NetIPC side, or it will fail with an EWOULDBLOCK error (in non-blocked mode). This is due to differences in the implementation of TCP on HP 3000 and HP 9000 systems.

Creating a Well-Known Port

On a NetIPC server, *ipccreate()* with an appropriate option containing the well-known port, is used. The option record is created using the *initopt()* and *addopt()* calls. The NetIPC *initopt()* call initializes the option record, and the NetIPC *addopt()* call adds the well-known port to the option record.

The following example does not include any error checking. HP recommends that including error checking in all programs is good practice.

NetIPC Server Example

```
short TCP_port = 31767  /* example port # */;
short opt[100];
short flags, result;
int call_desc, vc_desc;

initopt (opt, 1, &result);

/* Set up the option record */
addopt (opt, 0, NSO_PROTOCOL_ADDRESS, 2,
&TCP_port, &result);

/* Create a call socket at the well-known port */
flags = 0;

ipccreate (NS_CALL, NSP_TCP, &flags, opt,
&call_desc, &result);

/* Listen at the well known port then
generate a VC socket */
initopt (opt, 0, &opterr);

flags = 0;

ipcrecvn (call_desc, &vc_desc, &flags, opt,
&result);
```

NetIPC Client and BSD IPC Server

Similar to the BSD client to NetIPC server communications, a NetIPC client process can connect to a BSD server. Figure 3-2 shows the communications between the NetIPC client and the BSD Sockets application server.

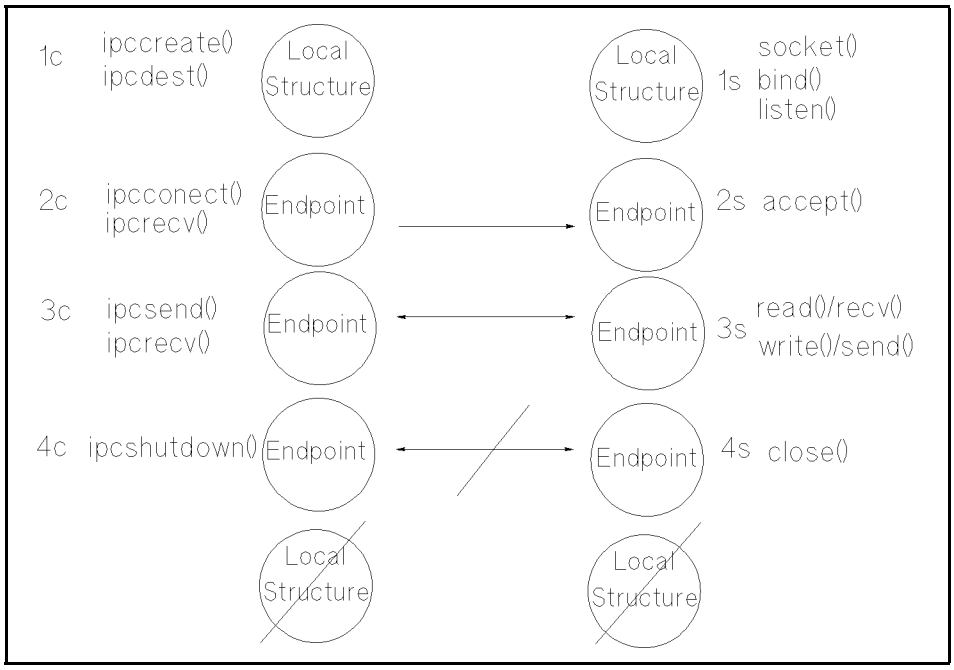

Figure 3-2. NetIPC to BSD Sockets Communication

The NetIPC client cannot use *ipclookup()* to find the server node and service by name. Since BSD Sockets do not support the Socket Registry, the client must use *ipcdest()* with a well-known socket. At the time of the *accept()* call in BSD IPC, the receiver can determine the information about the socket requesting the connection.

NetIPC and BSD IPC are simply interfaces to the transport protocol; they are not end to end protocols in themselves. When a client calls *ipcconnect()* and *ipcrecv()*, *ipcrecv()* will return when a connection has been established at the transport level. This implies that *listen()* has been called. The server process

may not have called *accept()* by the time the *ipcrecv()* returns even though *ipcconnect()* returns without an error. You can try to send and receive data after the initial *ipcrecv()* call, but *ipcsend()* and *ipcrecv()* may block before the BSD IPC application has called *accept()* to actually accept the connection.

Connecting To a Well-known Port

On a NetIPC client, the *ipcdest()* call is used to connect to a well-known port created by a BSD server application. The *ipcdest()* call creates a destination descriptor.

NetIPC Example

```
char *node_name;  /* remote node name */
int flags, result;
char prot_addr;
descriptor_type *dd;

flags = 0;
prot_addr = TCP_WELL_KNOWN_PORT;

ipcdest(NS_CALL, node_name, strlen(node_name),
NSP_TCP, (char *) &prot_addr, sizeof(prot_addr),
&flags, opt, &dd, &result);
```

4

BSD IPC Porting

BSD IPC Porting

The 4.3 version of Berkeley Software Distribution (BSD) offers a rich set of interprocess-communication (IPC) facilities referred to as Berkeley Sockets or BSD sockets. This chapter provides an overview of BSD Sockets and information about porting applications to the HP 3000 MPE/iX environment. In addition, this chapter discusses how to modify existing applications.

BSD Sockets Overview

Berkeley Sockets provide a C language application program interface (API) that is used as a de facto standard in writing many of the current distributed applications. This API provides a general interface to allow node-isolated and network-based applications to be constructed independently of the underlying communication facilities.

With the 4.0 release of MPE/iX, HP introduced the first portion of Berkeley Sockets/iX. This first portion supports stream and datagram communication types, and can operate on Local and Internet domains.

The basic building block for BSD communication is the socket. Sockets are communication endpoints that allow programs running on the same or different nodes to exchange messages and data.

The two common socket types are stream sockets and datagram sockets. A stream socket is a connection oriented model that supports reliable, sequenced flow of data. A datagram socket is a connectionless model that can potentially result in an unreliable and unsequensed flow of data.

Sockets can operate on different communication domains. Sockets exchange only with sockets in the same domain. The common socket domains are UNIX(Local) domain, and Internet(DARPA) domain. Sockets on Local domain can only be used to communicate with processes on the same node. Sockets on Internet domain can communicate with processes on the same or different nodes. Refer to the following figure for an illustration of how sockets communicate.

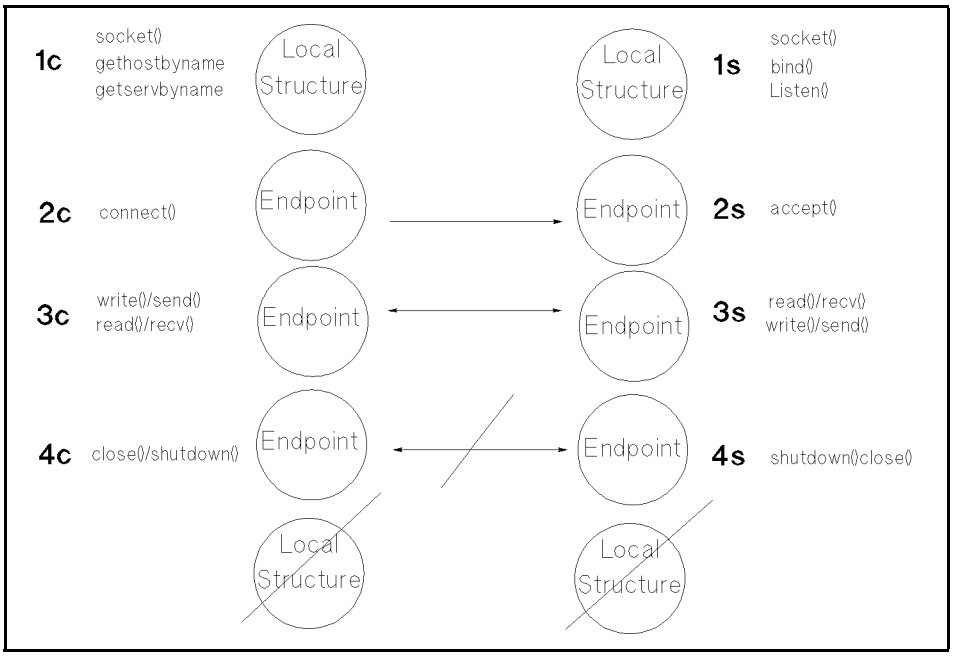

Figure 4-1. BSD Socket to Socket Communication

Establishing Connections

To establish a connection, the client process calls *socket()* to create the local data structure (see figure 4-1, step 1c). *socket()* creates an endpoint for communication and returns a descriptor which is used in all subsequent socket-related calls: `s=socket (AF_INET, SOCK_STREAM, 0);`

The server calls *socket()* to create the local data structure. It calls *bind()* to associate the socket with the server's protocol address: `bind(ls, &myaddr_in, sizeof(struct sockaddr_in))`. Then it calls *listen()* to have the transport protocol accept connection requests on the server's socket. With the *listen()* call, a queue for incoming connections is specified. The

listen queue is established for the socket specified by the s parameter (see figure 4-1, step 1s): `listen(ls,5)`.

The client requests a connection to the server's socket by calling *connect()* and blocking until the server has accepted the request (see figure 4-1, step 2c): `connect (s, &peeraddr_in, sizeof(struct sockaddr_in))`

The server calls *accept()*, and completes the "connection establishment phase" (see figure 4-1, step 2s). *accept()* extracts the first connection on the queue of pending connections, creates a new socket with the same properties as s, and allocates a new file descriptor ns for the socket: `accept(ls, &peeraddr_in, @adrlen);`

Transferring Data

At this point, data can be transferred by calling *send()* and *recv()* or the Unix file system calls *read()* and *write()* (see figure 4-1, steps 3c and 3s). *send()* is used to transmit a message to another socket: `send(s, buf, 10, 0)`. *recv()* is used to receive messages from a socket: `recv(s, buf, 10, 0)`.

Terminating Connections

Finally, either the client or the server process can terminate the connection by calling the BSD *shutdown()* call or the file system *close()* call (see figure 4-1, steps 4c and 4s): `shutdown(s, 1)`.

Utility Calls

The *gethostent*, *gethostbyname*, and *gethostbyaddr* subroutines return a pointer to an object with the fields that reflect information obtained from either the `/etc/hosts` (`HOSTS.NET.SYS` for MPE/iX) database or one of the name services identified in the `/etc/services` (`SERVICES.NET.SYS` for MPE/iX) file.

The *getservent*, *getservbyname*, and *getservbyport* subroutines each returns a pointer to an object with the fields of a line in the network services database, `etc/services` (`SERVICES.NET.SYS` for MPE/iX).

Porting BSD Applications to HP 3000 MPE/iX Environment

The following steps describe the process of porting BSD IPC applications from HP 9000 to HP 3000 systems. Refer to the flowchart in figure 4-2 for an illustration of this process.

- Identify unsupported BSD calls in the HP 9000 application.
- Modify the HP 9000 application to run on the HP 3000.
- Ensure that all the necessary include files are in the same group/account.
- Test the client/server applications.

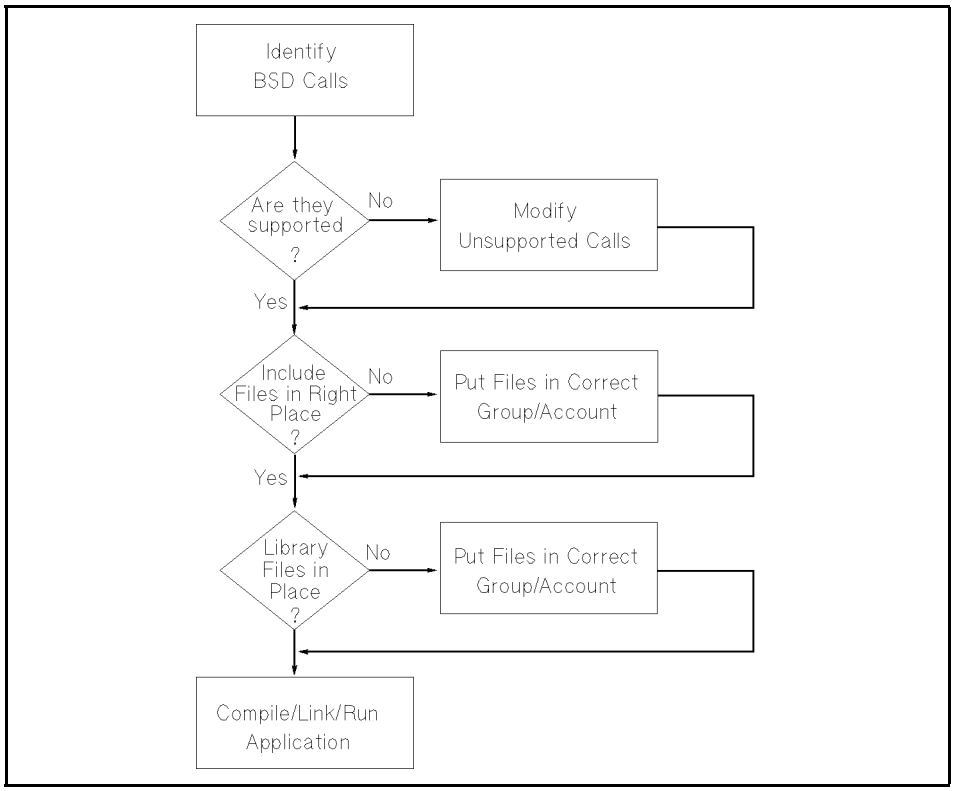

Figure 4-2. Porting BSD Applications to HP 3000 Systems

MPE/iX 4.0 Supported Calls

The 4.0 release of the MPE/iX operating system introduced the first portion of Berkeley Sockets system calls. All of these calls and intrinsics are used in the MPE/iX environment exactly as in the UNIX environment, for example, the same parameters apply.

Stream Sockets	Datagram Sockets
`socket()`	`socket()`
`bind()`	`bind()`
`listen()`	`listen()`
`accept()`	`accept()`
`connect()`	
`send()`	`sendto()`
`recv()`	`recvfrom()`
`shutdown()`	`shutdown()`
`getpeername()`	`getpeername()`
`socketpair()`	`socketpair()`

The following naming routines are also supported:

Stream Sockets	Datagram Sockets
`gethostbyxxx()`	`gethostbyxxx()`
`getnetbyxxx ()`	`getnetbyxxx ()`
`getprotbyxxx()`	`getprotbyxxx()`
`getservbyxxx()`	`getservbyxxx()`

Of the Naming Service Routines listed above, *gethostbyname* and *gethostbyaddr* are part of the MPE/iX Operating System. The rest of the routines are bundled with the Link products and must be purchased separately.

With MPE/iX release 4.0, *connect()*, *send()*, and *recv()* are not supported for datagram type sockets, and *recvfrom()* and *sendto()* are not supported on stream sockets.

Modifying Current Applications

Here is more specific information to help you modify your existing applications.

Establishing Connections for Datagram Sockets

Since *connect()* and *recv()* are not supported for datagram sockets, you should remove the *connect()* calls and replace the *recv()* calls with *recvfrom()* calls. If the source address is not the desired address when data is received, then the message is discarded and a new *recvfrom()* is posted.

Transferring Data for Datagram Sockets

Since *send()* is not supported for datagram sockets, you should replace the *send()* calls with *sendto()* calls. The destination address must be specified on each *sendto()* call instead of on the *connect()* call, which you should have removed.

Vectored Data Calls

If an application is using *sendmsg()* or *recvmsg()*, you should replace these with a call to *send()* or *recv()*. Instead of using vectored data, you should specify pointers to the actual data area. You should create a separate intrinsic call for each vector. There is no limit to the number of vectors you can use. If

you are using *sendmsg()* and *recvmsg()* to exchange file descriptors, there is no recommended modification.

Here is an example of how to convert a vectored data call. Assume that `MSG` points to the message structure used for *sendmsg()*. Within the structure is a pointer to an array of vectors, and each vector consists of a pointer and length. The message structure also contains a count of the number of vectors in the *send()*. To convert this to a *send()* call, you can use a `FOR` loop. *send()* will be called once for each vector. For each *send()* call, the current vector's pointer and length will be passed as the buff and len parameters for the *send()*. After the *send()* call, the current vector pointer will be updated to point to the next vector in the list.

READ and WRITE

read and *write* calls are not supported in MPE/iX release 4.0, but can be converted to *recv()* and *send()* calls. Similarly, *readv* and *writev* calls can be converted to *recv()* and *send()* calls, but the pointers to the actual data areas must be specified instead of using data vectors.

FORK and EXEC

fork and *exec* calls are supported in MPE/iX release 4.5. You can replace calls to *fork* and *exec* with calls to *CREATEPROCESS* and *ACTIVATE*. One significant difference is that the parent's data structures are not copied to the new process. The new process must re-initialize variables or obtain access to the parent's data. Messages can be exchanged between the two processes using *SENDMAIL* and *RECEIVEMAIL*.

Considerations for Different Compilers

You may find that your HP 9000 application includes UNIX calls that need to be translated to MPE/iX Operating System calls. Refer to the *UNIX to MPE/iX Cross Reference Guide*. This cross-reference guide is meant to assist software developers in porting their applications from UNIX platforms to MPE/iX platforms.

Also, in some instances a parameter declared as constant in the UNIX environment should be variable in the MPE/iX environment, for example:

```
vc_sd = accept (lsd, &peeraddr_in,
sizeof(struct sockaddr_in));
int x:
x=sizeof (struct sockaddr_in)
vc_sd = accept(lsd, &peeraddr_in, &x);
```

The MPE/iX equivalent to the `etc/hosts` file in the UNIX environment is the `HOSTS.NET.SYS` file. If your application refers to `etc/hosts`, replace it with `HOSTS.NET.SYS` as follows.

```
hp=gethostbyname (node_name);
if (hp == NULL) {
    printf("%s not found in /etc/hosts\n", node_name);
if (hp == NULL) {
    printf("%s not found in hosts.net.sys", node_name);
```

BSD Sockets Compilation Procedure

To compile a BSD IPC application, you need to create a library list file, called `rllist`, by entering following line:

```
libc.net.sys, socketrl.net.sys.
```

After creating the file, use the following steps to compile an application.

1. `ccxl file_name obj_file; info="-D_SOURCE-SOCKET"`
2. `link from=objc_file; to=exec_file; cap=ia,ba,ph; rl=^rllist;`

The following tables summarize the BSD Sockets implementations for different platforms.

Table 4-1. BSD Sockets Implementations - Calls

Type of Call	HP-UX BSD 4.3	BSD Sockets/iX	PC BSD
Establish Connection Calls:	socket()	socket()	socket()
	bind()	bind()	bind()
	connect()	connect()	connect()
	listen()	listen()	listen()
	accept()	accept()	accept()
	write()	write()	
	writev()		
	send()	send()	send()
	sendto()	sendto()	sendto()
	sendmsg()		
	read()	read()	
	readv()		
	recv()	recv()	recv()
	recvfrom()	recvfrom()	recvfrom()
	recvmsg()		
Terminate Connection Calls:	close()	close()	close_socket()
	shutdown()		
Utility Calls:	getpeername()	getpeername()	getpeername()
	getsockname()	getsockname()	getsockname()
	getsockopt()	getsockopt()	getsockopt()
	setsockopt()	setsockopt()	setsockopt()
	gethostname()	gethostname()	gethostname()
	sethostname()		
	getdomainname()		
	setdomainname()		
	ioctl()	ioctl()	ioctl()
	fcntl()		

Table 4-2. BSD Sockets Implementations - Routines

Type of Routine	HP-UX BSD 4.3	BSD Sockets/iX	PC BSD

Byte Order Conversion Routines:	`ntohs()`
	`ntohl()`
	`htons()`
	`htonl()`
		`ntohs()`
		`ntohl()`
		`htons()`
		`htonl()`
			`ntohs()`
			`ntohl()`
			`htons()`
			`htonl()`

Internet Address Routines:	`inet_addr()`
	`inet_network()`
	`inet_ntoa()`
	`inet_makeaddr()`
	`inet_netof()`
	`inet_inaof()`
			`inet_addr()`
			`inet_network()`
			`inet_ntoa()`
			`inet_makeaddr()`
			`inet_netof()`
			`inet_inaof()`

Domain Name Routines:	`res_init()`
	`res_send()`
	`res_mkquery()`
	`res_query()`
	`res_search()`
	`dn_expand()`
	`dn_comp()`		

Table 4-2. BSD Sockets Implementations - Routines (continued)

Type of Routine	HP-UX BSD 4.3	BSD Sockets/iX	PC BSD
Host Entry Routines:	gethostbyname()	gethostbyname()	gethostbyname()
	gethostbyaddr()	gethostbyaddr()	gethostbyaddr()
	gethostent()		gethostent()
	sethostent()		sethostent()
	endhostent()		endhostent()
	getnetbyname()	getnetbyname()	getnetbyname()
	getnetbyaddr()	getnetbyaddr()	getnetbyaddr()
	getnetent()		getnetent()
	setnetent()		setnetent()
	endnetent()		endnetent()
	getportbyname()	getportbyname()	getportbyname()
	getportbynumber()	getportbynumber()	getportbynumber()
	getportent()		getportent()
	setportent()		setportent()
	endportent()		endportent()
	getservbyname()	getservbyname()	getservbyname()
	getnetbyaddr()	getnetbyaddr()	getnetbyport()
	getservent()		getservent()
	setservent()		setservent()
	endservent()		endservent()

5

DSCOPY/9000 to FTP/9000 Migration

DSCOPY/9000 to FTP/9000 Migration

In the ongoing effort to promote standards for open systems networking, Hewlett-Packard recommends migrating from DSCOPY on the HP 9000 to FTP on the HP 9000. FTP/9000 is part of the well-known ARPA services and is already the predominant standard for file transfers between HP 3000, HP 9000, and HP 1000 systems.

For more information on using FTP on the HP 9000, refer to *Using ARPA Services*. For more information on using FTP on the HP 3000 S9xx, refer to the *HP ARPA File Transfer Protocol User's Guide*.

DSCOPY Options and FTP Commands

Following is a table that summarizes all of the DSCOPY options used on the HP 3000 S9xx and HP 9000 and the equivalent FTP commands that you can use to achieve the same result. FTP on the HP 9000 provides most of the functionality that DSCOPY does.

FTP 3000/9000/1000 refers to the ARPA FTP service on the HP 3000 S9xx, HP 9000, and HP 1000. Note the differences between transfers to an HP 3000 S9xx or HP 1000 and transfers to an HP 9000.

NOTE

The *;code*, *;rec*, and *;disc* functions are only applicable for transfers *to* the HP 3000. File transfers to the HP 9000 are only affected by the *ascii* and *binary* commands.

* Option is the default action and has no effect.
** Option is not available.
*** For the HP 1000, destination file attributes (file type, size, record length) can be specified in the destination file descriptor in *put* or *get* commands.
n/a Option is not applicable.

Table 5-1. DSCOPY Options and Equivalent FTP Commands

Option Mnemonic	DSCOPY 3000	DSCOPY 9000	DSCOPY 1000	FTP 3000	FTP 9000/1000
Ascii	ASC	-A	AS	ascii	ascii
Append	APP	**	**	append	append
Binary	BIN	-B	BI	binary	binary
Checkpointing	CHECKPT=	**	**	**	**
Clear	CLEAR	**	**	n/a	n/a
Compress	COMP	**	**	**	**
Direct	DIR	**	**	**	**
File Code	FCODE=	**	**	;code=<>	n/a|9000| ***
Fixed	FIX	-F	FI	;rec=,,f	n/a|9000| ***
File Size	FSIZE=	**	FS=	;disc=<>	n/a|9000| ***
Interactive Session	[RETURN]	-i	[RETURN]	*	*
Interchange	INT	*	IN	n/a	n/a
Insert Character	ICHAR=	-d<char>	IC=<char>	n/a	n/a
Move	MOVE	**	MO	get <file>	get <file>
				delete <file>	delete <file>
Overwrite	OVER	**	OV	**	**
Print results	*	-p	*	*	*
Prompt Lockword	**	-P	**	**	n/a
Quiet	QUIET	*	QU	**	**
Replace	REP	-r	RE	delete <file>	delete <file>
				put <file>	put <file>
Restart	RESTART	**	**	**	**
Record Size	RSIZE=	-L	RS=<recsize>	;rec=<>	n/a|9000| ***
Source Device	SDEV=	**	**	**	n/a
Search Character	SCHAR=	-s<char>	SC=	n/a	n/a
Sequential	SEQ	*	*	*	*
Show globals	SHOW	*	+SH	n/a	n/a
Strip	STRIP	**	ST	n/a	n/a
Target Device	TDEV=	**	**	**	n/a
Variable	VAR	**	VA	;rec=,,v	n/a|9000| ***

A

NetIPC Sample Programs

NetIPC Sample Programs

This appendix includes two working NetIPC sample programs, a client program and a server program. Use these programs as templates for your own applications.

NetIPC Client Program

```
/*-----------------------------------------------------------------------*/
/*                                                                       */
/*  Client: NetIPC Client Sample Program (non-Windows)                   */
/*                                                                       */
/*-----------------------------------------------------------------------*/

/*-----------------------------------------------------------------------*/
/*  COPYRIGHT (C) 1988 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.            */
/*-----------------------------------------------------------------------*/

/*
 * PURPOSE:
 *    Client to correspond with async Server example.
 *
 * REVISION HISTORY
 *
 * DESCRIPTION
 *    The Client uses NetIPC to send a user name to the Server and receive
 *    information associated with the user name from the Server.
 *
 *    General Algorithm:
 *    1.  Get the name of the remote node from the user.
 *    2.  Create a call socket (IPCCREATE).
 *    3.  Get the path descriptor for the Server's well-known socket
 *        (IPCDEST).
 *    4.  Request connection to the Server (IPCCONNECT).
 *    5.  Receive connection verification (IPCRECV).
 *    6.  Loop--ask the user for user name for information retrieval
 *        (until the user enters the string literal 'EOT').
 *    7.  Send the user name to the Server (IPCSEND).
 *    8.  Receive the information associated with the user name
 *        (IPCRECV).
 */
```

```
#define  LINT_ARGS

#include <stdio.h>         /* modified     */
#include <stdlib.h>        /* added        */
#include <string.h>        /* added        */
#include <sys/types.h>
#include <sys/errno.h>

#ifdef _MPEIX
  #include "nsipc"
  #pragma intrinsic IPCSEND
  #pragma intrinsic IPCRECV
  #pragma intrinsic IPCRECVCN
  #pragma intrinsic IPCERRMSG
  #pragma intrinsic IPCCONNECT
  #pragma intrinsic IPCCREATE
  #pragma intrinsic IPCSHUTDOWN
  #pragma intrinsic IPCDEST
  #pragma intrinsic IPCCONTROL
  #pragma intrinsic INITOPT
  #pragma intrinsic ADDOPT
#else /* _MPEIX */
  #include <sys/ns_ipc.h>
#endif /* _MPEIX */

#define    BUFFERLEN         20
#define    INFOBUFLEN        60
#define    LENGTH_OF_DATA    20
#define    MAX_BUFF_SIZE     1000
#define    TCP_ADDRESS       31500
           /* Well-known TCP address used by Server */
#define    INACTIVE_DESCRIPTOR -1
#define    INFINITE_WAIT     0
#define    INTSIZE           sizeof(short int)
#define    NODENAME_LEN      256
#ifdef _MPEIX
  #define    ipcsend           IPCSEND
  #define    ipccreate         IPCCREATE
  #define    ipcdest           IPCDEST
  #define    ipcconnect        IPCCONNECT
  #define    ipccontrol        IPCCONTROL
  #define    ipcrecv           IPCRECV
  #define    initopt           INITOPT
  #define    ipcshutdown       IPCSHUTDOWN
  #define    ipcerrmsg         IPCERRMSG
#endif /* _MPEIX */
```

```
/************************** FORWARD DECLARATIONS **************************/

void   ErrorRoutine(char * , int , ns_int_t );
void   ReceiveData(ns_int_t , char * );
void   SetUp(char * , ns_int_t * );
void   ShutdownDescriptor(ns_int_t );
void   CleanUp();

/************************** GLOBAL VARIABLES ****************************/

static char          version[] =
   "@(#) PC NetIPC client program. Version B.00.00";
static char          copyright[] =
   "(C) Copyright 1988 Hewlett-Packard Company.  All rights reserved.";

ns_int_t      cd = INACTIVE_DESCRIPTOR;

/*************************************************************************/
/*                                                                       */
/*    Procedure:  main                                                   */
/*                                                                       */
/*    Purpose:  Prompts the user for a remote node name.  Sets up a      */
/*                  connection to the server.  Continually prompts the   */
/*                  user for names to send to the server to lookup.      */
/*                  receives the information buffer associated with the  */
/*                  name from the server.                                */
/*                                                                       */
/*************************************************************************/

main()
{
   char       node_name[NODENAME_LEN];
   char       requested_name[BUFFERLEN+1]; /*Allow room for extra NULL*/
   char       *data_buf;     /*Allow room for extra NULL*/
   int        request;
   ns_int_t   dlen;
   ns_flags_t flags;
   int        i;
   int        result;

/*
**  Ask the user for the NS node name of the remote node
*/

   data_buf = (char *)malloc(INFOBUFLEN + 1);

   printf("Client: Enter the remote node name:  ");
   gets(node_name);
   printf("\n");
```

```
/*
**  Create call socket and connect to the server
*/

   SetUp(node_name, &cd);

/*
**  Process any incoming requests until "EOT" is received.
*/
   requested_name[1] = '1'; /* just an initial value != to EOT */

   while ( strcmp(requested_name, "EOT") != 0) {

     /*
      **  Ask the user for a name to be retrieved
      */

     printf("Client: Enter name for data retrieval (or EOT to exit):");
     gets(requested_name);

     if (strcmp(requested_name, "EOT") != 0)
       {
  /*
   **    Pad the name with spaces.
   */

  for (i = strlen(requested_name); i < BUFFERLEN; i++)
    requested_name[i] = ' ';

  /*
   **  Ask for the name the user requested.
   */

  flags = 0;
  ipcsend(cd, (char *)requested_name, BUFFERLEN, &flags, NULL,&result);
  if (result != NSR_NO_ERROR)
    ErrorRoutine("main calling ipcSend", result, cd);

  ReceiveData(cd, data_buf);

  /*
   **  Print out the data received
   */
```

```
  printf("Client data is: %s\n", data_buf);
       }
     else {
       printf("Received EOT\n");
       break;
     }

   } /* while ( strcmp(requested_name, "EOT") != 0); */

   CleanUp();
   return 0;

 } /* main */

/*************************************************************************/
/*                                                                       */
/*    Routine:  SetUp                                                    */
/*                                                                       */
/*    Description:  Perform setup operations:  Create a TCP call socket,  */
/*                  create destination descriptor for Server's well-known  */
/*                  call socket, establish VC with Server, set the VC    */
/*                  timeout to infinity, call IPCRECV to verify the Server */
/*                  received the connect request, wait for the server to  */
/*                  complete the VC.                                     */
/*                                                                       */
/*    Input:  node_name - name of remote node running server program.    */
/*                                                                       */
/*    Output:  cd - connection descriptor for VC.                        */
/*                                                                       */
/*    Global variables referenced:  NONE                                 */
/*                                                                       */
/*************************************************************************/

void
SetUp(node_name, cd)
char        *node_name;
ns_int_t    *cd;
{
      ns_int_t          sd = INACTIVE_DESCRIPTOR;
      ns_int_t          dd = INACTIVE_DESCRIPTOR;
      ns_int_t          dlen;
      int               result;
      short int         proto_addr;
      int               timeout;
      int               request;
      ns_flags_t        flags;

      dlen = 0;
```

```
/*
**  A call socket is created by calling IPCCREATE.  The value returned
** in the sd parameter will be used in the subsequent calls.
*/

ipccreate(NS_CALL, NSP_TCP, NULL, NULL, &sd, &result);

if (result != NSR_NO_ERROR)
      ErrorRoutine("SetUp calling ipcCreate", result, sd);

/*
**  The server is waiting on a well-known address (TCP_ADDRESS).
**  Create the destination descriptor for the socket from the
**  remote node.
*/

proto_addr = TCP_ADDRESS;
ipcdest(NS_CALL, node_name, strlen(node_name), NSP_TCP,
      (short int *)&proto_addr, INTSIZE, NULL, NULL, &dd, &result);

if (result != NSR_NO_ERROR)
      ErrorRoutine("SetUp calling ipcDest", result, dd);

/*
**  Now connect to the server
*/

*cd = INACTIVE_DESCRIPTOR;
ipcconnect(sd, dd, NULL, NULL, cd, &result);

if (result != NSR_NO_ERROR)
      ErrorRoutine("SetUp calling ipcConnect", result, *cd);

/*
**  Set the timeout to infinity with IPCCONTROL for later calls
*/

timeout = INFINITE_WAIT;
ipccontrol(*cd, NSC_TIMEOUT_RESET, (char *)&timeout, INTSIZE,
      NULL, NULL, NULL, &result);

if (result != NSR_NO_ERROR)
      ErrorRoutine("SetUp calling ipcControl", result, *cd);

/*
**  Verify the server received the connect request.  Wait for the
**  server to do an ipcrecvcn. (receive the ack/syn packet)
*/
```

```
      ipcrecv(*cd, &request, &dlen, &flags, NULL, &result);

      if (result != NSR_NO_ERROR)
            ErrorRoutine("Setup calling ipcRecv", result, *cd);


      /*
      **  Shutdown the source and destination descriptors since we don't
      **  need them any more.
      */

      ShutdownDescriptor(sd);
      ShutdownDescriptor(dd);
}

/*****************************************************************************/
/*                                                                           */
/*    Routine:  ReceiveData                                                  */
/*                                                                           */
/*    Description:  Receives data from Server.  Loops on IPCRECV until       */
/*                  the total amount of data is received.                    */
/*                                                                           */
/*    Input:  cd - connection descriptor to receive data on.                 */
/*                                                                           */
/*    Output:  info_buf - buffer containing inbound data.                    */
/*                                                                           */
/*    Global variables referenced:  NONE                                     */
/*                                                                           */
/*****************************************************************************/

void
ReceiveData(cd, info_buf)
ns_int_t              cd;
char                 *info_buf;
{
   ns_int_t      amount_to_recv;
   ns_int_t      amt_recvd;
   ns_flags_t    flags;
   int           request;
   char          *buffer;
   int           result;
   short int     res16;
   short int     opt[100];
   ns_int_t      rc;

   /* Set up option array */
```

```
    initopt(opt, 0, &res16);
    if (res16 != NSR_NO_ERROR)
      ErrorRoutine("ReceiveData calling initopt",res16, cd);

    flags = 0;
    amount_to_recv = INFOBUFLEN;
    buffer = info_buf;
    amt_recvd = 0;

    while (amt_recvd < amount_to_recv )
    {
      rc = amount_to_recv;  /* This is a two way value, input=max to recv,
       ** output = amount received.
       */
      ipcrecv(cd, buffer, &rc, &flags, opt, &result);

      if (result != NSR_NO_ERROR)
        ErrorRoutine("ReceiveData calling IPCRECV", result, cd);

      flags = 0;  /* Reset the flag, because NSF_MORE_DATA will always be set */
      amt_recvd += rc;
      buffer += rc;
    }

       /*
       **  Tack on an extra NULL so we can print the info out using printf().
       */

       *(info_buf + INFOBUFLEN) = '\0';
}

/*************************************************************************/
/*                                                                       */
/*     Routine:  CleanUp                                                 */
/*                                                                       */
/*     Description:  We have some problem, so we need to shutdown all of */
/*             the descriptors and terminate the program.                */
/*                                                                       */
/*     Input:  NONE                                                      */
/*                                                                       */
/*     Output:  NONE                                                     */
/*                                                                       */
/*     Global variables referenced:  cd                                  */
/*                                                                       */
/*************************************************************************/
```

```
void
CleanUp()
{
   ShutdownDescriptor(cd);
   exit(1);
}

/*************************************************************************/
/*                                                                       */
/*    Routine:  ErrorRoutine                                             */
/*                                                                       */
/*    Description:  We have some error, so print a message and then call */
/*           CleanUp which will shutdown all descriptors and terminate   */
/*           the program.                                                */
/*                                                                       */
/*    Input:  Msg - string which contains the name of the routine who    */
/*                  had the error.                                       */
/*            error - integer error number that occured.                 */
/*            descr - descriptor for which the error occured.            */
/*                                                                       */
/*    Output:  NONE                                                      */
/*                                                                       */
/*    Global variables referenced:  NONE                                 */
/*                                                                       */
/*************************************************************************/

void
ErrorRoutine(msg, error, descr)
char           *msg;
int             error;
ns_int_t        descr;
{
   char         buffer[80];
   ns_int_t     result;
   int          buffer_size;

   buffer_size = 80;

   printf("Client: Error occurred in %s.\n", msg);
   printf("Client: The error code is: %d.  The local descriptor is %d\n",
   error, descr);
   ipcerrmsg( error, buffer, &buffer_size,  &result);
   if (result)
     printf("ipcerrmsg failed\n");
   else
     printf("Client: NetIPC error = %s\n",buffer);
```

```
    if (errno)
      perror("\n");
    CleanUp();
}

/*****************************************************************************/
/*                                                                           */
/*    Routine:  ShutdownDescriptor                                           */
/*                                                                           */
/*    Description:  Shutdowns a given descriptor.  This descriptor can be    */
/*           either a source or connection descriptor.                       */
/*                                                                           */
/*    Input:  desc - descriptor to be shutdown.                              */
/*                                                                           */
/*    Output:  NONE                                                          */
/*                                                                           */
/*    Global variables referenced:  NONE                                     */
/*                                                                           */
/*****************************************************************************/

void
ShutdownDescriptor(descr)
ns_int_t      descr;
{
      int    result;

      /*
      **  Don't worry about errors here, since there isn't much we can do.
      */

      ipcshutdown(descr, NULL, NULL, &result);

}     /* ShutdownDescriptor */
```

NetIPC Server Program

```
/*-------------------------------------------------------------------------*/
/*                                                                         */
/*   SERVER: NetIPC Server Sample Program                                  */
/*-------------------------------------------------------------------------*/

/*-------------------------------------------------------------------------*/
/*  COPYRIGHT (C) 1988 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.              */
/*-------------------------------------------------------------------------*/

/* PURPOSE:
 *    To show the operation of asynchronous NetIPC calls.
 *
 * REVISION HISTORY
 *
 * DESCRIPTION
 *    The Server uses IPC to receive a user name from a Client and sends
 *    information associated with the user name back to the Client.
 *    The Server can have connections to 5 Clients.
 *
 *    General Algorithm:
 *    1.  Create a well-known call socket (IPCCREATE).
 *    2.  Post a nowait ipcrecvCn to receive connection requests sent from
 *        clients.
 *    3.  When the ipcrecvCn completes, receive the connection and post
 *        a nowait ipcrecv to receive the requested user name.
 *    4.  Since the ipcrecv may complete before receiving all of the user
 *        name, additional ipcrecv calls may have to be posted to receive
 *        all of the user name.
 *    5.  Once all of the user name is received, open the file
 *        named "datafile."  Scan datafile until the user
 *        record and information associated with the user name are found.
 *    6.  Call ipcsend (nowait) to send the information associated
 *        with the user name.
 *    7.  Post a nowait ipcrecv on the VC to receive next user name or
 *        shutdown notification from the remote (the ipcrecv completes
 *        with error 65, REMOTE ABORT).
 *    8.  Upon receipt of shutdown notification, call ipcshutDown to
 *        shut the VC.
```

```
 *    Since all IPC calls (except IPCCREATE) are done nowait, the main loop
 *    calls IPCWait, determines what type of event completed, and calls the
 *    appropriate procedure to handle the event.
 *-----------------------------------------------------------------------*/

#define  LINT_ARGS
#include <stdio.h>
#ifdef _MPEIX
  #include "nsipc"
  #pragma intrinsic IPCSEND
  #pragma intrinsic IPCRECV
  #pragma intrinsic IPCRECVCN
  #pragma intrinsic IPCERRMSG
  #pragma intrinsic IPCCONNECT
  #pragma intrinsic IPCCREATE
  #pragma intrinsic IPCSHUTDOWN
  #pragma intrinsic IPCDEST
  #pragma intrinsic IPCCONTROL
  #pragma intrinsic INITOPT
  #pragma intrinsic ADDOPT
#else
  #include <sys/ns_ipc.h>
#endif
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <sys/types.h>
#include <sys/errno.h>


#define       FALSE               0
#define       TRUE                !FALSE
#define       INBUFFERLEN         20
#define       OUTBUFLEN           60
#define       LENGTH_OF_DATA      20
#define       MAX_BUFF_SIZE       1000
#define       MAX_NUM_VCS         5
#define       INACTIVE_DESCRIPTOR -1
#define       INTSIZE             sizeof(int)
#define       INFINITE_WAIT       0

/*
** This TCP address is in the user-allocatable range. BE SURE
** THAT THIS ADDRESS IS NOT BEING USED BY ANOTHER APPLICATION.
*/
```

```
#define       TCP_ADDRESS          31500
#define       RECOVERABLE          1
#define       IRRECOVERABLE        2
#define       NSR_GRACEFUL_RELEASE 68
              /* This is not defined in <sys/ns_ipc.h> */

#ifdef _MPEIX
  #define ipcsend        IPCSEND
  #define ipccontrol     IPCCONTROL
  #define ipcrecv        IPCRECV
  #define ipcrecvcn      IPCRECVCN
  #define initopt        INITOPT
  #define addopt         ADDOPT
  #define ipccreate      IPCCREATE
  #define ipcshutdown    IPCSHUTDOWN
  #define ipcerrmsg      IPCERRMSG
#endif

/*************************** GLOBAL VARIABLES ***************************/

static char version[] =
   "@(#) PC NetIPC server program. Version B.00.00";
static char copyright[] =
   "(C) Copyright 1988 Hewlett-Packard Company.  All rights reserved.";

int                     vcs_available;
fd_set                  readfds;
fd_set                  writefds;
fd_set                  exceptfds;

ns_int_t      sd = INACTIVE_DESCRIPTOR;

struct vc_type       {
       ns_int_t      cd;
       unsigned int         amount_received;
       char                 in_buffer[INBUFFERLEN+1];
                            /*Allow room for a NULL*/
       char                 out_buffer[OUTBUFLEN];
} vc_table[MAX_NUM_VCS] = {INACTIVE_DESCRIPTOR, 0, 0, 0};
```

```
/************ PROTOTYPE DECLARATIONS FOR COMPILER PARAMETER CHECKING********/

void ErrorRoutine(char * , int , ns_int_t , int );
int HandleNewRequest(ns_int_t );
void ProcessRead(ns_int_t , unsigned );
void ReadData(char * , char *);
void SetUp();
void InitiateConnection(ns_int_t * );
void ShutdownDescriptor(ns_int_t * );
void CheckFile();
void CleanUp();
void RejectRequest(ns_int_t );
int FindVcInTable(ns_int_t , ns_int_t ** );
char *sock_strerror(int);

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

main()
{
   ns_flags_t              flags;
   int                     request;
   ns_int_t                dlen;
   int                     result;
   ns_int_t                serving_cd;
   ns_int_t               *serving_cd_ptr;
   int                     num_recvcns_posted = 0;
   int                     read_result[64], except_result[64];
   ns_int_t                timeout;
   int                     bound;
   int                     i;
   fd_set                  rfds;
   fd_set                  efds;
   int                     nfound;
   char                    *errptr;

   vcs_available = MAX_NUM_VCS;

   /*
   **  Check that the data file can be opened.
   */

   printf("CheckFile()\n");
   CheckFile();
```

```
FD_ZERO((fd_set *) &readfds);
FD_ZERO((fd_set *) &exceptfds);
/*
**  Create a call socket with a well known address for the clients
**  to use.
*/

SetUp();


/*
** The num_recvcns_posted variable keeps track of the number of
** outstanding connection requested.  We only want one at a time.
** If we get up to the maximum number of VCs established, then we
** will have zero recvcns posted.
*/

num_recvcns_posted = 1;


/*
**  Loop forever waiting to serve clients.  The following cases
**  are possible:
**
**  - a new clients request service (request = IPCRECVCN).
**  - a client asks for information (request = IPCRECV).
**  - a client receives the data (request = IPCSEND).
**  Handle each one of these cases in this loop.
**
**  If any other situations occur, exit out of the loop, and let the
**  clean up routine de-allocate the sockets for this server.
*/

while (TRUE)
{
  bound = MAX_NUM_VCS + sd ;
  rfds = readfds;
  efds = exceptfds;
  nfound = select(bound, (int *)&rfds, 0, (int *)&efds, 0);
  printf("connections: %d\n",MAX_NUM_VCS - vcs_available);

  if (nfound < 0){
    errptr = sock_strerror(errno);
    fprintf(stderr,"select(): %s (%d)\n",errptr, errno);
    CleanUp();
  }
```

```
        /*
        **  Check the listen queue for any new connection requests.
        **  Decrement the recvcns_posted count, and handle the new
        **  request.  HandleNewRequest returns TRUE if it successfully
        **  created another session.  If so, increment the count.
        **
        **  New NetIPC connections are indicated as an exception on
        **  hp-ux, but as a read event on MPEiX.
        **  If BSD Sockets where used, hp-ux would also handle a
        **  connection request as a read event.
        */

#ifdef _MPEIX
     if (FD_ISSET(sd, (fd_set *) &rfds))
#else
     if (FD_ISSET(sd, (fd_set *) &efds))
#endif
       {
  if (vcs_available)
    HandleNewRequest(sd) ;
  else
    RejectRequest(sd) ;
       }

     for (i = 0; i < (bound - sd); i++)
       {
         /*
         **    Process any requests.
         */
  if (vc_table[i].cd != -1)
    if (FD_ISSET(vc_table[i].cd, (fd_set *) &rfds))
      {
        ProcessRead(vc_table[i].cd, 0);
      }
       }
   } /* while */
   return;
}
```

```
/************************************************************************/
/*                                                                      */
/*     Routine:  CheckFile                                              */
/*                                                                      */
/*     Description:  Check that the file can be opened.                 */
/*                                                                      */
/*     Input:  NONE                                                     */
/*                                                                      */
/*     Output:  NONE                                                    */
/*                                                                      */
/*     Global variables referenced:                                     */
/*                                                                      */
/************************************************************************/

void
CheckFile()
{
       FILE   *fd;

       if ((fd = fopen("datafile", "r")) == NULL)
  ErrorRoutine("CheckFile", 0, 0, IRRECOVERABLE);
       fclose(fd);
}


/************************************************************************/
/*                                                                      */
/*     Routine:  SetUp                                                  */
/*                                                                      */
/*     Description:  Called when the server is first started up, SetUp  */
/*            initializes the vc_table and creates the server's source  */
/*            descriptor.                                               */
/*                                                                      */
/*     Input:  NONE                                                     */
/*                                                                      */
/*     Output:  vc_table - initialized with INACTIVE_DESCRIPTOR and     */
/*              contains the cd for the first connection.               */
/*                                                                      */
/*     Global variables modified:  sd - source descriptor for the server.  */
/*                                                                      */
/************************************************************************/
```

```
void
SetUp()
{
   short int        option[100], res16;
   int              result;
   short int        timeout;
   short int        port_address;
   int              i;

   /*
   **  Set up the opt array for the parm we will use.
   */

   initopt(option, 1, &res16);
   if (res16 != NSR_NO_ERROR)
     ErrorRoutine("InitOpt", result, 0, IRRECOVERABLE);

   /*
   **  Now add the option for the well-known address for the
   **  IPCCreate Call
   */

   port_address = TCP_ADDRESS;
   addopt(option, 0, NSO_PROTOCOL_ADDRESS, sizeof(short int),
   (short int *)&port_address,&res16);
   if (result != NSR_NO_ERROR)
            ErrorRoutine("AddOpt", result, 0, IRRECOVERABLE);


   /*
   **  A call socket is created by calling IPCCREATE.  The value returned
   **  in the sd parameter will be used in the following calls.
   */

   ipccreate(NS_CALL, NSP_TCP, NULL, option, &sd, &result);
   if (result != NSR_NO_ERROR)
            ErrorRoutine("IPCCreate", result, sd, IRRECOVERABLE);

   /*
   **  Set the sd timeout to infinity with IPCControl for later calls
   */

   timeout = INFINITE_WAIT;
   ipccontrol(sd, NSC_TIMEOUT_RESET, (char *)&timeout, sizeof(short int),
             NULL, NULL, NULL, &result);
   if (result != NSR_NO_ERROR)
            ErrorRoutine("SetUp: IPCControl", result, sd, IRRECOVERABLE);
```

```
   /*
   **  Set the 'select' bits.
   */

   FD_SET(sd, &readfds);
   FD_SET(sd, &exceptfds);

   /*
   **  Inititalize the vc_table
   */

   for (i=0; i < MAX_NUM_VCS; i++)
     vc_table[i].cd = INACTIVE_DESCRIPTOR;
}


/*************************************************************************/
/*                                                                       */
/*     Routine:  CleanUp                                                 */
/*                                                                       */
/*     Description:  We have some problem, so we need to shutdown all of */
/*                   the descriptors and terminate the program.          */
/*                                                                       */
/*     Input:  NONE                                                      */
/*                                                                       */
/*     Output:  NONE                                                     */
/*                                                                       */
/*     Global variables referenced:  vc_table                            */
/*                                                                       */
/*************************************************************************/

void
CleanUp()
{
   int    i;

   /*
   **  Shutdown the source descriptor.
   */

   ShutdownDescriptor(&sd);

   /*
   **  Shutdown all of the VC's that are active in the vc_table.
   **  Once again, don't worry about errors.
   */
```

```
   for (i = 0; i < MAX_NUM_VCS; i++)
   {
      if (vc_table[i].cd != INACTIVE_DESCRIPTOR)
               ShutdownDescriptor(&vc_table[i].cd);
   }
   exit(1);
}


/*****************************************************************************/
/*                                                                           */
/*     Routine:  ErrorRoutine                                                */
/*                                                                           */
/*     Description:                                                          */
/*        We have some error, so print a message.  If the severity is        */
/*        IRRECOVERABLE, call CleanUp which will shutdown all descriptors    */
/*        and terminate the program.                                         */
/*                                                                           */
/*     Input:  routine - string which contains the name of the routine who   */
/*                   had the error.                                          */
/*             error - integer error number that occured.                    */
/*             descr - descriptor for which the error occured.               */
/*             severity - RECOVERABLE it we want to just log a message       */
/*                   else IRRECOVERABLE and we want to terminate execution   */
/*                                                                           */
/*     Output:  NONE                                                         */
/*                                                                           */
/*     Global variables referenced:  NONE                                    */
/*                                                                           */
/*****************************************************************************/

#define BUFFSIZE 80

void
ErrorRoutine(routine, error, descr, severity)
  char            *routine;
  int              error;
  ns_int_t    descr;
  int              severity;
{
   char           buffer[BUFFSIZE];
   ns_int_t       result;
   int            buffer_size = BUFFSIZE;

   printf("Server: Error occurred in %s call.\n", routine);
   printf("Server: The error code is: %d.  The descriptor is: %d\n",
      error, descr);
```

```
   ipcerrmsg(error, buffer, &buffer_size, &result);
   if (result)
     printf("ipcerrmg failed!\n");
   else
     printf("Server: NetIPC error %s\n",buffer);

   if (severity == IRRECOVERABLE)
      CleanUp();
}

#undef BUFFSIZE

/*************************************************************************/
/*                                                                       */
/*     Routine:  FindVcInTable                                           */
/*                                                                       */
/*     Description:  Searches the VC table for a given descriptor        */
/*                                                                       */
/*     Input:  cd - descriptor to search the VC table for                */
/*                                                                       */
/*     Output:  cd_ptr - pointer to the descriptor in the VC table found */
/*                                                                       */
/*     Return:  TRUE if descriptor was found in table, else FALSE.       */
/*                                                                       */
/*     Global variables referenced:  vc_table                            */
/*                                                                       */
/*************************************************************************/

int
FindVcInTable(cd, cd_ptr)
  ns_int_t    cd;
  ns_int_t    **cd_ptr;
{
   int       i;

   for (i=0; i < MAX_NUM_VCS; i++)
   {
      if (cd == vc_table[i].cd)
      {
            *cd_ptr = &vc_table[i].cd;
            break;
      }
   }
```

```
   if (i <  MAX_NUM_VCS)
   {
      return(TRUE);
   }
   else
   {
      return(FALSE);
   }
}


/*************************************************************************/
/*                                                                       */
/*     Routine:  HandleNewRequest                                        */
/*                                                                       */
/*     Description:  A new client wants to talk to us, complete the vc   */
/*            establishment.                                             */
/*                                                                       */
/*     Input:  cd - descriptor for the newly completed virtual circuit.  */
/*                                                                       */
/*     Output:  NONE                                                     */
/*                                                                       */
/*     Returns:  TRUE if another recvcn was posted, else FALSE.          */
/*                                                                       */
/*     Global variables referenced:  NONE                                */
/*                                                                       */
/*************************************************************************/

int
HandleNewRequest(cd)
  ns_int_t      cd;
{
   int            result;
   short int      timeout;
   int            i;
   ns_int_t       dlen;
   ns_flags_t     flags;
   ns_int_t       temp;

/*
**  Find an available place in the table.
*/


   /*
   **  Search for the next available vc descriptor in the vc_table.
   */
```

```
   for (i=0; i < MAX_NUM_VCS; i++)
   {
      if (vc_table[i].cd == INACTIVE_DESCRIPTOR)
            break;
   }


   /*
   **  If we found an available vc descriptor in the vc_table, then
   **  initiate another connection.  If we are out of available
   **  descriptors, then skip it;  We will then initiate a connection
   **  as an existing connection is shutdown.
   */

   if (i < MAX_NUM_VCS)
   {
      InitiateConnection(&vc_table[i].cd);
      return(TRUE);
   }
   else /* This should never happen... */
   {
     printf("HandleNewRequest: vcs_available = %d\n",vcs_available);
     CleanUp();
   }
}

/*************************************************************************/
/*                                                                       */
/*     Routine:  ProcessRead                                             */
/*                                                                       */
/*     Description:  We have a client program which has send in a read   */
/*            request.  So process the read.                             */
/*                                                                       */
/*     Input:  cd - descriptor for the virtual circuit.                  */
/*             amt_recvd - amount of data received from completing Recv  */
/*                                                                       */
/*     Output:  NONE                                                     */
/*                                                                       */
/*     Global variables referenced:  NONE                                */
/*                                                                       */
/*************************************************************************/

void
ProcessRead(cd, amt_recvd)
  ns_int_t           cd;
  unsigned int       amt_recvd;
```

```
{
   int             result;
   ns_int_t        dlen;
   char            *buffer;
   int             i;
   ns_flags_t      flags;

   /*
   **  We have received data on the given VC.
   **  So determine which VC from the vc_table we have.
   */

   for (i = 0; i < MAX_NUM_VCS; i++)
   {
      if (cd == vc_table[i].cd)
            break;
   }

   if (i == MAX_NUM_VCS)
      ErrorRoutine("ProcessRead", 0, cd, IRRECOVERABLE);



   /*
   **  Receive the data
   */

   for (vc_table[i].amount_received= 0 ;
 vc_table[i].amount_received < INBUFFERLEN;
 vc_table[i].amount_received += dlen) {

     dlen = INBUFFERLEN - vc_table[i].amount_received;
     buffer = vc_table[i].in_buffer + vc_table[i].amount_received;
     flags = 0;

     ipcrecv(cd, buffer, &dlen, &flags, NULL, &result);

   /*
   ** Since MPEIX has Resets(aborts) and graceful releases on the
   ** read instead of an exception, we account for that.
   */


     if (result != NSR_NO_ERROR)
```

```
#ifdef _MPEIX
       if ((result == NSR_REMOTE_ABORT) || (result == NSR_GRACEFUL_RELEASE))
  {
    ShutdownDescriptor(&vc_table[i].cd);
    return;
  }
       else
#endif
  ErrorRoutine("ProcessRead calling ipcrecv", result, cd,
        IRRECOVERABLE);
   }

   /*
   **  If we have all of the name from the client, then get the
   **  data we need from the file to send to the client.  But first,
   **  NULL terminate the name field.  So we can print it out in
   **  ReadData.
   */

   vc_table[i].in_buffer[INBUFFERLEN] = '\0';

   ReadData(vc_table[i].in_buffer, vc_table[i].out_buffer);


   /*
   **  And send the data back to the client
   */

   ipcsend(cd, vc_table[i].out_buffer, OUTBUFLEN, NULL,NULL,&result);
   if (result != NSR_NO_ERROR)
      ErrorRoutine("ipcsend", result, cd, IRRECOVERABLE);


   /*
   **  Clear the amount_received field since we may be receiving
   **  more names from the user after the Send completes.
   */

   vc_table[i].amount_received = 0;
 }
```

```
/************************************************************************/
/*                                                                      */
/*     Routine:  ReadData                                               */
/*                                                                      */
/*     Description:  From the file 'datafile' look for the given name and  */
/*            if found return the information on the given name.  If the   */
/*            given name is not found return a string that says so.        */
/*                                                                      */
/*     Input:  name - a string name keyword to search for in 'datafile'    */
/*                                                                      */
/*     Output:  output_buffer - contains the information found for name.   */
/*                                                                      */
/*     Global variables referenced:  NONE                               */
/*                                                                      */
/************************************************************************/

void
ReadData(name, output_buffer)
  char   *name;
  char   *output_buffer;
{
   FILE    *datafile;
   char    *ptr;
   char    input_buffer[INBUFFERLEN+OUTBUFLEN+2];  /* Allow room for name */
                                                   /* data, new-line, and */
                                                   /* NULL                */

   /*
   **  Open the file named "datafile". Search until the last record
   **  is found, or we match the user name the client wants.  The
   **  format of the file is characters 1 - 20 comprise the name field.
   **  Characters 22 - 80 comprise the data field.  If there is a match,
   **  retreive the remaining data from the file, and send it back
   **  (be sure to pad the data field with spaces since NULL-terminated
   **  strings are not recognized by the HP3000 server program in Pascal.
   **  If there is no match, return "name not found" to the client.
   */

#ifdef _MPEIX
   if ((datafile = fopen("DATAFILE", "r")) == NULL)
#else
   if ((datafile = fopen("datafile", "rt")) == NULL)
#endif
     ErrorRoutine("ReadData", 0, 0, IRRECOVERABLE);
```

```
while (fgets(input_buffer, sizeof(input_buffer), datafile) != NULL)
{

   /*
   **  Now see if the name matches the name key.  Compare at most
   **    INBUFFERLEN characters and ignore cases during the compare.
   */

   if (strncmp(name, input_buffer, INBUFFERLEN) == 0)
   {

         /*
         **  We found the name the client requested in the file.  So fill
         **  the input line with spaces starting where the newline character
         **  which gets() puts on at the end of the line.
         */

         ptr = &input_buffer[INBUFFERLEN+1];
         while ((*ptr != '\n') & (*ptr != '\0'))
         {
         ptr++;
         }
         while (ptr < &input_buffer[INBUFFERLEN+OUTBUFLEN])
         {
               *ptr++ = ' ';
         }

         /*
         **  Print a message to the server console.
         */

         printf("Server: %s information found.\n", name);


         /*
         **  Copy the data found into the given output_buffer, close
         **  the datafile and return.
         */

         strncpy(output_buffer, &input_buffer[INBUFFERLEN+1],
                 OUTBUFLEN);
         fclose(datafile);
         return;
   }
```

```
    /*
    **  If we did not find the name, continue searching the file.
    */
  }  /* search the file */


  /*
  **  We've fallen out of the while loop because we reached the end
  **  of the file.  Print an error message and put a 60-byte error
  **  message in the data buffer.
  */

  printf("Server: %s not in file.\n", name);

  strncpy(output_buffer,
     "SERVER did not find the requested name in the datafile.     ",
     OUTBUFLEN);

  /*
  ** Close the datafile and return.
  */

  fclose(datafile);
  return;
}     /* ReadData */


/*****************************************************************************/
/*                                                                           */
/*     Routine:  InitiateConnection                                          */
/*                                                                           */
/*     Description:  Initiates a connection by calling ipcrecvCn             */
/*                                                                           */
/*     Input:  NONE                                                          */
/*                                                                           */
/*     Output:  cd - connection descriptor for the new connection.           */
/*                                                                           */
/*     Global variables referenced:                                          */
/*        sd - source descriptor                                             */
/*        vcs_available - global number of vcs available                     */
/*                                                                           */
/*                                                                           */
/*****************************************************************************/
```

```
void
InitiateConnection(cd)
  ns_int_t      *cd;
{
   int       result;

   /*
   **  Initiate the connection.  NOTE:  on a PC, the ipcrecvCn is always
   **  an unblocked call.  So we do not need to enable NOWAIT I/O here.
   */

   printf("ipcrecvCn()\n");

   ipcrecvcn(sd, cd, NULL, NULL, &result);

   FD_SET(*cd, &readfds);
   vcs_available--;

   if (result != NSR_NO_ERROR)
      ErrorRoutine("ipcrecvCn", result, *cd, IRRECOVERABLE);
}     /* InitiateConnection */

/*****************************************************************************/
/*                                                                           */
/*     Routine:  ShutdownDescriptor                                          */
/*                                                                           */
/*     Description:  Shutdowns a given descriptor.  This descriptor can be */
/*            either a source or connection descriptor.                      */
/*                                                                           */
/*     Input:  desc - descriptor to be shutdown.                             */
/*                                                                           */
/*     Output:  NONE                                                         */
/*                                                                           */
/*     Global variables referenced:                                          */
/*        vcs_available - global number of vcs available.                    */
/*                                                                           */
/*****************************************************************************/
```

```
void
ShutdownDescriptor(desc)
   ns_int_t      *desc;
{
   int         result;

   ipcshutdown(*desc, NULL, NULL, &result);

   /*
   ** Don't worry about errors here, since there isn't much we can do.
   */

   FD_CLR(*desc, &readfds);
   vcs_available++;
   *desc = INACTIVE_DESCRIPTOR;
 }

void
RejectRequest(cd)
  ns_int_t cd;
{
  ns_int_t     temp;

  InitiateConnection(&temp);
  ShutdownDescriptor(&temp);

}

char *
sock_strerror(error_num)
  int error_num;

{
  return ( strerror(error_num) );
}
```

B

BSD IPC Sample Programs

BSD IPC Sample Programs

This appendix includes two working BSD IPC sample programs, a client program and a server program. Use these programs as templates for your own applications.

BSD IPC Client Program

```
/*---------------------------------------------------------------------------*/
/*                                                                           */
/*  Client: BSD Sockets Client Sample Program (non-Windows)                  */
/*                                                                           */
/*---------------------------------------------------------------------------*/


/*---------------------------------------------------------------------------*/
/*  COPYRIGHT (C) 1988 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.                */
/*---------------------------------------------------------------------------*/

/*
 * PURPOSE:
 *    Client to correspond with async Server example.
 *
 * REVISION HISTORY
 *
 * DESCRIPTION
 *    The BSD client application sends a request to the server
 *    at a well-known port and receives the information requested.
 *    The host name used must be configured in the /etc/hosts file
 *    and the service name and well-known port must be configured
 *    in the /etc/services file.
 *
 *    Note that PCs compiled with the small and medium models
 *    must declare some variables as far.
 *
 *    General Algorithm:
 *    1.  Get the name of the remote node from the user.
 *    2.  Create a socket (socket).
 *    3.  Connect to the server's well-known port (connect).
 *    4.  Loop -- ask the user to enter a request for information
 *        until the user enters the string literal 'EOT'.
 *    5.  Send the request to the Server (send).
 *    6.  Receive the information requested (recv).
 */
```

```
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* BSD Sockets Include Files */

#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/errno.h>  /* needed for 3k and 9k systems */

#define    BUFFERLEN         20
#define    INFOBUFLEN        60
#define    TCP_ADDRESS       31500  /* Well-known server address. For BSD
                                   ** SOCKETS, you could add a service
        ** to the /etc/services file
        ** (services.net.sys for MPE/iX) and
        ** use getservbyname().  To use
        ** this, compile with GETSERVBYNAME
        ** defined.
        */


/************************* PROTOTYPE DECLARATIONS ************************/


void   SetUp(char *);
void   ReceiveData(char *);
void   CleanUp(void);
void   ShutdownDescriptor(void);
char   *sock_strerror(int);



/************************** GLOBAL VARIABLES *****************************/

static char          version[] =
   "@(#) PC BSD client program. Version B.00.00";
static char          copyright[] =
   "(C) Copyright 1988 Hewlett-Packard Company.  All rights reserved.";

int sd;     /* socket descriptor    */
static char serv_name[] = "myserv";
```

```
/*************************************************************************/
/*                                                                       */
/*    Procedure:  main                                                   */
/*                                                                       */
/*    Purpose:  Prompts the user for a remote node name.  Sets up a      */
/*              connection to the server.  Then continually prompts the  */
/*              user for names to send to the server to lookup.  Then    */
/*              receives the information buffer associated with the      */
/*              name from the server.                                    */
/*************************************************************************/

main()
{

    char  requested_name[BUFFERLEN+1]; /*Allow room for extra NULL*/
    char  data_buff[INFOBUFLEN + 1];   /*Allow room for extra NULL*/
    int   i;
    int num;
    char *errptr;
    char host_name[25];


    /*
    **    Ask the user for the host name of the remote node
    */

    printf("Client: Enter the remote host name:  ");
    gets(host_name);


    /*
    **    Create call socket and connect to the server
    */

    SetUp(host_name);

    do {

     /*
     **  Ask the user for a name to be retrieved
     */

     printf("Client: Enter name for data retrieval (or EOT to exit):");
     gets(requested_name);
```

```
        if (strcmp(requested_name, "EOT") != 0)
        {
            /*
            **    Pad the name with spaces so that send is always BUFFERLEN
            **    in length.
            */

            for (i = strlen(requested_name); i <  BUFFERLEN; i++)
            requested_name[i] = ' ';


            /*
            **    Send the request to the server.
            */

            num = send(sd, requested_name, BUFFERLEN, 0);
            if (num < 0)
            {
             errptr = sock_strerror(errno);
             printf("send(): %s (%d)\n", errptr, errno);
             CleanUp();
            }

            /*
            **    Get information buffer from server.
            */


            ReceiveData(data_buff);

            /*
            **    Print out the data received
            */

            printf("Client data is: %s\n", data_buff);
        }

      } while (strcmp(requested_name, "EOT") != 0);

      CleanUp();
      return(0);

  }
```

```
/************************************************************************/
/*                                                                      */
/*    Routine:  SetUp                                                   */
/*                                                                      */
/*    Description:  Perform setup operations:  Create a socket and      */
/*                  connect to the server's well-known port.            */
/*                                                                      */
/*    Input:  host_name - name of remote host running server program.   */
/*                                                                      */
/*    Output:  sd - socket descriptor.                                  */
/*                                                                      */
/*    Global variables referenced:  NONE                                */
/*                                                                      */
/************************************************************************/

void SetUp(host_name)
char *host_name;
{

     int rc;     /* return code        */
     int req;
     int len;
     int flag = 1;  /* for ioctl  on 9k */

     char argp;
     char *errptr;
     char buff[100];

     struct hostent *hp;
#ifdef GETSERVBYNAME
     struct servent *sp;
#endif
     struct sockaddr_in server;
     struct sockaddr_in client;

     /* allocate and clear out address structures for 3k and 9k
     */

     memset ((char *)&server, 0, sizeof(struct sockaddr_in));
     memset ((char *)&client, 0, sizeof(struct sockaddr_in));

     /*
     **  The IP address is determined by calling gethostbyname().
     */
```

```
        hp = gethostbyname((char *) host_name);
        if (hp == NULL)
        {
            printf("Unable to locate host entry.\n");
            CleanUp();
        }

    #ifdef GETSERVBYNAME
        /*
        **  The well-known port number is determined by calling
        **  getservbyname().
        */

        sp = getservbyname(serv_name, "tcp");
        if (sp == NULL)
        {
            printf("Unable to locate service entry.\n");
            CleanUp();
        }
    #endif

        /*
        **  A socket is created by calling socket().  The value returned
        **  willl be used in subsequent calls.
        */

        sd = socket(AF_INET, SOCK_STREAM, 0);

        if (sd < 0)
        {
            errptr = sock_strerror(errno);
            printf("%d: %s\n", errno, errptr);
            CleanUp();
        }

        /*
        **  Connect to the server.
        */

        server.sin_family = AF_INET;
    #ifdef GETSERVBYNAME
        server.sin_port = sp->s_port;
    #else
        server.sin_port = TCP_ADDRESS;
    #endif
        server.sin_addr.s_addr =
          ((struct in_addr *) (hp->h_addr))->s_addr;
```

```
     rc = connect(sd, &server, sizeof(struct sockaddr_in));
     if (rc < 0)
     {
       errptr = sock_strerror(errno);
       printf("connect(): %s (%d)\n", errptr, errno);
       CleanUp();
     }

/* Set sd for non-blocking i/o mode */

#ifdef _NONBLOCK
     rc = ioctl(sd, FIOSNBIO, &flag);
     if (rc <  0)
     {
         errptr = sock_strerror(errno);
         printf("ioctl(): %s (%d)\n", errptr, errno);
         CleanUp();
     }
#endif
}

/*************************************************************************/
/*                                                                       */
/*    Routine:  ReceiveData                                              */
/*                                                                       */
/*    Description:  Receives data from the server.  Loops on recv until  */
/*             the total amount of data is received.                     */
/*                                                                       */
/*    Input:  cd - connection descriptor to receive data on.             */
/*                                                                       */
/*    Output:  buff - buffer containing inbound data.                    */
/*                                                                       */
/*    Global variables referenced:  sd                                   */
/*                                                                       */
/*************************************************************************/

void ReceiveData(buff)
char *buff;
{
     unsigned int amt_to_recv;
     long amt_recvd;
     int rc;
     char *errptr;
     int error;

     amt_to_recv = INFOBUFLEN;
                 /*  The server should always send INFOBUFLEN bytes. */
     amt_recvd = 0;
```

```
    while (amt_recvd < amt_to_recv){
      rc = recv(sd, buff, amt_to_recv, 0);
      if ( rc < 0 ) {
  if (errno != EWOULDBLOCK){ /* This is for Non-Blocked i/o */
    error = errno;
    errptr = sock_strerror(errno);
    printf("ReceiveData recv: %s (%d)\n",errptr, error);
  }
  rc = 0;
       }
      amt_recvd += rc;
    }

    /*
    **  Tack on an extra NULL so we can print the info out using printf().
    */

    *(buff + INFOBUFLEN) = '\0';
}

/*****************************************************************************/
/*                                                                           */
/*     Routine:  CleanUp                                                     */
/*                                                                           */
/*     Description:  We have some problem, so we need to shutdown the        */
/*         connection and terminate the program.                             */
/*                                                                           */
/*     Input:  NONE                                                          */
/*                                                                           */
/*     Output:  NONE                                                         */
/*                                                                           */
/*     Global variables referenced:  sd                                      */
/*                                                                           */
/*****************************************************************************/

void CleanUp()
{
     close(sd);
     exit(1);
}

char *sock_strerror(error_num)

int error_num;
{

  return (strerror(errno));

}
```

```
/*---------------------------------------------------------------------------*/
/*                                                                           */
/*   SERVER: BSD Sockets Server Sample Program                               */
/*                                                                           */
/*---------------------------------------------------------------------------*/


/*---------------------------------------------------------------------------*/
/*  COPYRIGHT (C) 1988 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.                */
/*---------------------------------------------------------------------------*/

/* PURPOSE:
 *    To show the operation of BSD socket calls.
 *
 * REVISION HISTORY
 *
 * DESCRIPTION
 *    The Server uses BSD Sockets to receive a request from a client
 *    send the response back.  The server can accept connections from
 *    up to 5 clients.
 *
 *
 *    General Algorithm:
 *    1.  Create a well-known call socket.
 *    2.  Select waiting for incoming connections requests or data.
 *    3.  When the select completes on the call socket, accept the connection
 *        and wait on select again to receive the requested user name.
 *    4.  Since the recv may complete before receiving all of the user name,
 *        additional recv calls may have to be called to receive all of
 *        the user name.
 *    5.  Once all of the user name is received, open the file
 *        named "datafile."  Scan datafile until the user
 *        record and information associated with the user name are found.
 *    6.  Call send (nowait) to send the information associated
 *        with the user name.
 *    7.  Wait using select to receive next user name or
 *        shutdown notification from the remote.
 *    8.  Upon receipt of shutdown notification, call close the VC
 *
 *---------------------------------------------------------------------------*/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <sys/types.h>
#include <sys/errno.h>

/* BSD Sockets Include Files */

#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <errno.h>    /* needed for 3k and 9k systems */

#define FALSE              0
#define TRUE               !FALSE
#define INBUFFLEN          20
#define OUTBUFFLEN         60
#define LENGTH_OF_DATA     20
#define MAX_NUM_SOCKETS    5
#define TCP_ADDRESS        31500

#define BACKLOG            5
#define MAXHOSTNAME        32
#define MAX_NUM_CONN       32

#define INACTIVE_DESCRIPTOR   -1

/*************************** GLOBAL VARIABLES ***************************/

static char version[] =
   "@(#) PC NetIPC server program. Version B.00.00";
static char copyright[] =
   "(C) Copyright 1988 Hewlett-Packard Company.  All rights reserved.";

/*
**  Global variables for selecting the client sockets with I/O
**  pending.
*/

fd_set readfds;
fd_set writefds;
fd_set exceptfds;

/* int readfds, writefds, exceptfds; */
```

```
/*
**  The nfds variable is used by select.
*/

int nfds;
int vcs_available;

int sd; /* server socket descriptor */

struct csd_type     {
       int sd;
       struct sockaddr_in addr;
} csd_table[MAX_NUM_SOCKETS];

static char serv_name[] = "myserv";

/************ PROTOTYPE DECLARATIONS FOR COMPILER PARAMETER CHECKING ******/


void CheckFile(void);
void SetUp(void);
int HandleNewRequest(void);
void InitiateConnection(int);
void ProcessRead(int, int);
void ReadData(char * , char *);
void ShutdownDescriptor(int);
void CleanUp(void);
char *sock_strerror(int);


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

main()
{
    int i, dlen, csd;
    char *errptr;
    int nfound;
    fd_set rfds;

    dlen = LENGTH_OF_DATA;
    vcs_available = MAX_NUM_SOCKETS;
```

```
/*
**    Check that the data file can be opened.
*/

CheckFile();

/*
**    Initialize the client socket descriptor table.
*/

for (i = 0; i < MAX_NUM_SOCKETS; i++)
{
 csd_table[i].sd = INACTIVE_DESCRIPTOR;
}

/*
**    Initialize the descriptor sets.
*/

FD_ZERO((fd_set *) &readfds);

/*
**    Create a call socket with a well known address for the clients
**    to use.
*/

SetUp();

/*
**    Loop forever waiting to serve clients.  The following cases are
**    possible:
**
**    - a new clients request service.
**    - a client asks for information.
**
**    Handle each one of these cases in this loop.
**
**    If any other situations occur, exit out of the loop, and let the
**    clean up routine de-allocate the sockets for this server.
*/
```

```
    nfds = sd + 1;

    while (TRUE)
    {
     rfds = readfds;
     nfound = select(nfds, (int *)&rfds, 0, 0, 0);

     printf("nfds: %d\n", nfds);
     if (nfound < 0)
     {
         errptr = sock_strerror(errno);
         fprintf(stderr, "select(): %s (%d)\n", errptr, errno);
         CleanUp();
     }

     /*
     **  Check the listen queue for any new connection requests.
     **  Decrement the recvcns_posted count, and handle the new
     **  request.  HandleNewRequest returns TRUE if it successfully
     **  created another session.  If so, increment the count.
     */

     if (FD_ISSET(csd_table[0].sd, (fd_set *) &rfds))
     {
       if (vcs_available)
         HandleNewRequest();
     }

     for (i = 1; i < MAX_NUM_SOCKETS; i++)
     {
         /*
         **    Process any requests.
         */
       if (csd_table[i].sd != -1)
         if (FD_ISSET(csd_table[i].sd, (fd_set *) &rfds))
         {
          ProcessRead(csd_table[i].sd, dlen);
         }
     }
    }
    return(0);
}
```

```
/*************************************************************************/
/*                                                                       */
/*     Routine:  CheckFile                                               */
/*                                                                       */
/*     Description:  Check that the file can be opened.                  */
/*                                                                       */
/*     Input:  NONE                                                      */
/*                                                                       */
/*     Output:  NONE                                                     */
/*                                                                       */
/*     Global variables referenced:                                      */
/*                                                                       */
/*************************************************************************/

void CheckFile()
{
    FILE *fd;

    if ((fd = fopen("datafile", "r")) == NULL)
    {
     fprintf(stderr, "Unable to open datafile.\n");
     exit(1);
    }
    fclose(fd);
}

/*************************************************************************/
/*                                                                       */
/*     Routine:  SetUp                                                   */
/*                                                                       */
/*     Description:  Called when the server is first started up, SetUp   */
/*         initializes the csd_table, creates the server's source        */
/*            descriptor and then initiates the first connection.        */
/*                                                                       */
/*     Input:  NONE                                                      */
/*                                                                       */
/*     Output: csd_table - initialized with INACTIVE_DESCRIPTOR and      */
/*             contains the cd for the first connection.                 */
/*                                                                       */
/*     Global variables modified:  sd - source descriptor for the server.*/
/*                                                                       */
/*************************************************************************/
```

```
void SetUp()
{
    char option[40];
    int   result;
    int   timeout;
    int   port_address;
    char *errptr;
    int rc;

    struct hostent *hp;
#ifdef GETSERVBYNAME
    struct servent *sp;
#endif
    struct sockaddr_in sa;

    char localhost[MAXHOSTNAME];


    /* allocate and clear out address structure    for 3k and 9k
    */
    memset ((char*)&sa, 0, sizeof(struct sockaddr_in));

#ifdef GETSERVBYNAME
    /*
    **    Lookup the well-known port to use.
    */

    sp = getservbyname(serv_name, "tcp");
    if (sp == NULL)
    {
     errptr = sock_strerror(errno);
     fprintf(stderr, "getservbyname(): %s (%d)\n", errptr, errno);
     CleanUp();
    }
#endif

    /*
    **    Initialize the socket structure.
    */

    sa.sin_family = AF_INET;
#ifdef GETSERVBYNAME
    sa.sin_port = sp->s_port;
#else
    sa.sin_port = TCP_ADDRESS;
#endif
    sa.sin_addr.s_addr = INADDR_ANY;
```

```
    /*
    **    Create a socket.
    */

    sd = socket(AF_INET, SOCK_STREAM, 0);
    if (sd < 0)
    {
     errptr = sock_strerror(errno);
     fprintf(stderr, "socket(): %s (%d)\n", errptr, errno);
     CleanUp();
    }

    /*
    **    Bind the socket to the well-known port.
    */

    rc = bind(sd, (struct sockaddr_in *) &sa, sizeof(struct sockaddr_in));
    if (rc < 0)
    {
     errptr = sock_strerror(errno);
     fprintf(stderr, "bind(): %s (%d)\n", errptr, errno);
     CleanUp();
    }


    /*
    **    Listen creates a port at which connection requests come in
    **    and sets the maximum number of connections the server will
    **    queue.
    */

    rc = listen(sd, BACKLOG);
    if (rc <  0)
    {
     errptr = sock_strerror(errno);
     fprintf(stderr, "listen(): %s (%d)\n", errptr, errno);
     CleanUp();
    }

    /*
    **    Add the listen queue port to the descriptor set and initialize
    **    the connection table.
    */

    FD_SET(sd, &readfds);
    csd_table[0].sd = sd;

}
```

```
/*************************************************************************/
/*                                                                       */
/*     Routine:  HandleNewRequest                                        */
/*                                                                       */
/*     Description:  A new client wants to talk, establish the vc.       */
/*                                                                       */
/*     Input:  cd - descriptor for the newly completed virtual circuit.  */
/*                                                                       */
/*     Output:  NONE                                                     */
/*                                                                       */
/*     Returns:  TRUE if another recvcn was posted, else FALSE.          */
/*                                                                       */
/*     Global variables referenced:  NONE                                */
/*************************************************************************/

int HandleNewRequest()
{
    int   i;
    int   timeout;

    /*
    **    Search for the next available descriptor in the csd_table.
    */

    for (i = 0; i < MAX_NUM_SOCKETS; i++)
    {
     if (csd_table[i].sd == INACTIVE_DESCRIPTOR)
          break;
    }

    /*
    **    If we found an available descriptor in the csd_table, then
    **    initiate another connection.  If we are out of available
    **    descriptors, skip it; we will initiate a connection when an
    **    existing connection is shutdown.
    */

    if (i < MAX_NUM_SOCKETS)
    {
     InitiateConnection(i);
     return(TRUE);
    }
    else
    {
     fprintf(stderr, "Connect request queued.\n");
     return(FALSE);
    }
}
```

```
/*************************************************************************/
/*                                                                       */
/*     Routine:  InitiateConnection                                      */
/*                                                                       */
/*     Description:  Initiates a connection by calling accept            */
/*                                                                       */
/*     Input:  NONE                                                      */
/*                                                                       */
/*     Output:  cd - connection descriptor for the new connection.       */
/*                                                                       */
/*     Global variables referenced:  sd - source descriptor              */
/*                                                                       */
/*************************************************************************/

void InitiateConnection(num)
int num;
{
    int csd;
    struct sockaddr *addrptr;
    int addrlen;
    char *errptr;

    /*
    **    Accept the incoming connection request.
    */

    addrptr = (struct sockaddr *) &csd_table[num].addr;
    addrlen = sizeof(csd_table[num].addr);

    csd_table[num].sd = accept(sd, addrptr, &addrlen);

    if (csd_table[num].sd < 0)
    {
     errptr = sock_strerror(errno);
     fprintf(stderr, "accept(): %s (%d)\n", errptr, errno);
     CleanUp();
    }


    /*
    **    Add the connection to the descriptor set.
    */

    FD_SET(csd_table[num].sd, (fd_set *) &readfds);
```

```
    /*
    **    The nfds global variable should be set to one more than the
    **    highest numbered descriptor used.
    */

    nfds++;
    vcs_available--;

}

/*************************************************************************/
/*                                                                       */
/*     Routine:  ProcessRead                                             */
/*                                                                       */
/*     Description:  We have a client program which has send in a read   */
/*            request.  So process the read.                             */
/*                                                                       */
/*     Input:  cd - descriptor for the virtual circuit.                  */
/*             amt_recvd - amount of data received from completing Recv  */
/*                                                                       */
/*     Output:  NONE                                                     */
/*                                                                       */
/*     Global variables referenced:  NONE                                */
/*                                                                       */
/*************************************************************************/

void
ProcessRead(csd, amt_to_recv)
int csd;
int amt_to_recv;
{
    int   rc, result;
    int   amt_recvd, dlen;
    char inbuff[INBUFFLEN+1];
    char outbuff[OUTBUFFLEN];
    char *errptr;
    int   error;
    int   i, flags;

    flags = 0;

    /*
    **    Receive the request from the client.
    */
```

```
    amt_recvd = 0;
    while (amt_recvd < amt_to_recv){

      rc = recv(csd, inbuff, amt_to_recv, 0);
      if ( (rc == 0) || ( (rc < 0) & ( errno = ECONNRESET )) )
{  /* Received a fin (bsd client) or a reset (NetIPC client) */
ShutdownDescriptor(csd);
       return;
      }
      if ( rc < 0 ) {
if (errno != EWOULDBLOCK){ /* This is for Non-Blocked i/o */
  error = errno;
  errptr = sock_strerror(errno);
  printf("ReceiveData recv: %s (%d)\n",errptr, error);
}
rc = 0;
      }
      amt_recvd += rc;
    }


    /*
    **    If we have all of the name from the client, then get the
    **    data we need from the file to send to the client.  But first,
    **    NULL terminate the name field.  So we can print it out in
    **    ReadData.
    */

    inbuff[INBUFFLEN] = '\0';

    ReadData(inbuff, outbuff);

    /*
    **    And send the data back to the client
    */

    send(csd, outbuff, OUTBUFFLEN, flags);
    if (rc < 0)
    {
     errptr = sock_strerror(errno);
     fprintf(stderr, "send(): %s (%d)\n", errptr, errno);
     CleanUp();
    }

}
```

```
/************************************************************************/
/*                                                                      */
/*     Routine:  ReadData                                               */
/*                                                                      */
/*     Description:  From the file 'datafile' look for the given name and  */
/*            if found return the information on the given name.  If the   */
/*            given name is not found return a string that says so.        */
/*                                                                      */
/*     Input:  name - a string name keyword to search for in 'datafile'    */
/*                                                                      */
/*     Output:  output_buffer - contains the information found for name.   */
/*                                                                      */
/*     Global variables referenced:  NONE                               */
/*                                                                      */
/************************************************************************/

void ReadData(name, output_buffer)
char *name;
char *output_buffer;
{
    FILE *datafile;
    char *ptr;
    char input_buffer[INBUFFLEN+OUTBUFFLEN+2];   /* Allow room for name */
                               /* data, new-line, and */
                               /* NULL       */

    /*
    **    Open the file named "datafile".     Search until the last record
    **    is found, or we match the user name the client wants.  The
    **    format of the file is characters 1 - 20 comprise the name field.
    **    Characters 22 - 80 comprise the data field.  If there is a match,
    **    retreive the remaining data from the file, and send it back
    **    (be sure to pad the data field with spaces since NULL-terminated
    **    strings are not recognized by the HP3000 server program in Pascal.)
    **    If there is no match, return "name not found" to the client.
    */

    if ((datafile = fopen("datafile", "r")) == NULL)
    {
     fprintf(stderr, "Unable to open datafile.\n");
    }

    while (fgets(input_buffer, sizeof(input_buffer), datafile) != NULL)
    {
```

```
    /*
    **  Now see if the name matches the name key.  Compare at most
    **  INBUFFLEN characters and ignore cases during the compare.
    */

    if (strncmp(name, input_buffer, INBUFFLEN) == 0)
    {

        /*    We found the name the client requested in the file.  So
        **    fill the input line with spaces starting where the newline
        **    character which gets() puts on at the end of the line.
        */

        ptr = &input_buffer[INBUFFLEN+1];
        while ((*ptr != '\n') & (*ptr != '\0'))
        {
         ptr++;
        }
        while (ptr <= &input_buffer[INBUFFLEN+OUTBUFFLEN])
        {
         *ptr++ = ' ';
        }

        /*
        **    Print a message to the server console.
        */

        printf("Server: %s information found.\n", name);

        /*
        **    Copy the data found into the given output_buffer,
        **    close the datafile and return.
        */

        strncpy(output_buffer, &input_buffer[INBUFFLEN+1],
             OUTBUFFLEN);
        fclose(datafile);
        return;
     }

     /*
     **  If we did not find the name, continue searching the file.
     */
    }
```

```
    /*
    **    We've fallen out of the while loop because we reached the end
    **    of the file.  Print an error message and put a 60-byte error
    **    message in the data buffer.
    */

    printf("Server: %s not in file.\n", name);

    strncpy(output_buffer,
         "SERVER did not find the requested name in the datafile.     ",
          OUTBUFFLEN);

    /*
    **    Close the datafile and return.
    */

    *(output_buffer+OUTBUFFLEN-1) = NULL;
    fclose(datafile);
    return;
}

/*****************************************************************************/
/*                                                                           */
/*     Routine:  ShutdownDescriptor                                          */
/*                                                                           */
/*     Description:  Shutdowns a given descriptor.  This descriptor          */
/*                   can be either a source or connection descriptor.        */
/*                                                                           */
/*     Input:  desc - descriptor to be shutdown.                             */
/*                                                                           */
/*     Output:  NONE                                                         */
/*                                                                           */
/*     Global variables referenced:  NONE                                    */
/*                                                                           */
/*****************************************************************************/

void ShutdownDescriptor(csd)
int csd;
{
    int rc, i;

    for (i = 0; i < MAX_NUM_SOCKETS; i++)
    {
```

```
    if (csd == csd_table[i].sd)
     {
         csd_table[i].sd = INACTIVE_DESCRIPTOR;
         rc = TRUE;
         break;
     }
    }
    if (rc != TRUE)
    {
     fprintf(stderr, "Unable to find socket descriptor in table.\n");
    }

    /*
    **    Remove client socket descriptor from the descriptor sets,
    **    close down the descriptor, and decrement the number of open
    **    descriptors.
    */

    FD_CLR(csd, (fd_set *) &readfds);
    close(csd);
    nfds--;
    vcs_available++;
}

/*****************************************************************************/
/*                                                                           */
/*     Routine:  CleanUp                                                     */
/*                                                                           */
/*     Description:  We have some problem, so we need to shutdown all of     */
/*            the descriptors and terminate the program.                     */
/*                                                                           */
/*     Input:  NONE                                                          */
/*                                                                           */
/*     Output:  NONE                                                         */
/*                                                                           */
/*     Global variables referenced:  csd_table                               */
/*                                                                           */
/*****************************************************************************/
```

```
void CleanUp()
{
    int     i;

    /*
    **    Shutdown the source descriptor.
    */

    ShutdownDescriptor(sd);

    /*
    **    Shutdown all of the VC's that are active in the csd_table.
    **    Once again don't worry about errors.
    */

    for (i = 0; i <  MAX_NUM_SOCKETS; i++)
    {
     if (csd_table[i].sd != INACTIVE_DESCRIPTOR)
     {
         ShutdownDescriptor(csd_table[i].sd);
     }
    }

    /*
    **    Exit with a non-zero result.
    */

    exit(1);
}

char *sock_strerror(error_num)

int error_num;
{
  return ( strerror(error_num) );
}
```

C

NetIPC Sample Include File

NetIPC Sample Include File

This appendix includes a sample NetIPC include file. Use this sample file as a template for your own applications.

HP 3000 Include File

```
/* $Header: ns_ipc.h,v 1.51.61.3 92/04/22 12:05:33 smp Exp $ */

#ifndef _SYS_NS_IPC_INCLUDED
#define _SYS_NS_IPC_INCLUDED

/* ns_ipc.h: NetIPC (NS) definitions */

#ifdef _KERNEL_BUILD
#include "../h/stdsyms.h"
#else  /* ! _KERNEL_BUILD */
#include "stdsyms"
#endif /* _KERNEL_BUILD */

#ifdef _INCLUDE_HPUX_SOURCE

    /* Types */

#ifdef _KERNEL_BUILD
#  include "../h/types.h"
#else  /* ! _KERNEL_BUILD */
#  include <sys/types.h>
#endif /* _KERNEL_BUILD */

   typedef int ns_int_t;
   typedef int ns_flags_t;
   typedef char *ns_opt_t;
   typedef char *opt_t;


   /* Function prototypes */

#  ifndef _KERNEL
#  ifdef _cplusplus
     extern "C" {
#  endif /* _cplusplus */
```

```
#  ifndef _MPEIX
#  ifdef _PROTOTYPES
     extern void addopt(short [], short, short, short, short [], short *);
     extern void initopt(short [], short, short *);
     extern int optoverhead(short, short *);
     extern void readopt(short [], short, short *, short *, short [],
                         short *);

     extern char *ipcerrstr(int);
     extern void ipcerrmsg(int, char *, int *, int *);
     extern void ipcconnect(ns_int_t, ns_int_t, ns_int_t *, short [],
                         ns_int_t *, ns_int_t *);
     extern void ipccontrol(ns_int_t, ns_int_t, const char *, ns_int_t,
                         char *, ns_int_t *, ns_int_t *, ns_int_t *);
     extern void ipccreate(ns_int_t, ns_int_t, ns_int_t *, short [],
                         ns_int_t *, ns_int_t *);
     extern void ipcdest(ns_int_t, const char *, ns_int_t, ns_int_t,
                         short *, ns_int_t, ns_int_t *, short [],
                         ns_int_t *, ns_int_t *);
     extern void ipcgetnodename(char *, ns_int_t *, ns_int_t *);
     extern void ipclookup(const char *, ns_int_t, const char *, ns_int_t,
                         ns_int_t *, ns_int_t *, ns_int_t *, ns_int_t *,
                         ns_int_t *);
     extern void ipcname(ns_int_t, const char *, ns_int_t, ns_int_t *);
     extern void ipcnameerase(const char *, ns_int_t, ns_int_t *);
     extern void ipcrecv(ns_int_t, void *, ns_int_t *, ns_int_t *,
                         short [], ns_int_t *);
     extern void ipcrecvcn(ns_int_t, ns_int_t *, ns_int_t *, short [],
                         ns_int_t *);
     extern void ipcselect(ns_int_t *, int [2], int [2], int [2], ns_int_t,
                         ns_int_t *);
     extern void ipcsend(ns_int_t, const void *, ns_int_t, ns_int_t *,
                         short [], ns_int_t *);
     extern void ipcsetnodename(const char *, ns_int_t, ns_int_t *);
     extern void ipcshutdown(ns_int_t, ns_int_t *, short [], ns_int_t *);
```

```
#  else /* not _PROTOTYPES */
     extern void addopt();
     extern void initopt();
     extern int optoverhead();
     extern void readopt();
     extern char *ipcerrstr();
     extern void ipcerrmsg();
     extern void ipcconnect();
     extern void ipccontrol();
     extern void ipccreate();
     extern void ipcdest();
     extern void ipcgetnodename();
     extern void ipclookup();
     extern void ipcname();
     extern void ipcnameerase();
     extern void ipcrecv();
     extern void ipcrecvcn();
     extern void ipcselect();
     extern void ipcsend();
     extern void ipcsetnodename();
     extern void ipcshutdown();
#  endif /* not _PROTOTYPES */
#  endif /* _MPEIX */

#  ifdef _cplusplus
     }
#  endif /* _cplusplus */
#  endif /* not _KERNEL */


   /*
    *   Naming constants
    */
#  define NS_MAX_NODE_NAME      50
#  define NS_MAX_NODE_PART      16
#  define NS_MAX_SOCKET_NAME    16

   /*
    * MAX TIMEOUT as enforced by itimerfix and hence by ipccontrol
    */
#  define NIPC_MAX_TIMEO        1000000

   /*
    *   socket type definitions
    */
#  define NS_CALL       3
#  define NS_VC         6
#  define NS_DEST       7
#  define NS_NULL_DESC  (-1)
```

```
   /*
    *   protocol IDs
    */
#  define NSP_TCP       4

   /*
    *   flags values
    */
#  define NSF_VECTORED          0x00000001      /* Netipc std bit 31 */
#  define NSF_PREVIEW           0x00000002      /* Netipc std bit 30 */
#  define NSF_MORE_DATA         0x00000020      /* Netipc std bit 26 */
#  define NSF_DATA_WAIT         0x00000800      /* Netipc std bit 20 */
#  define NSF_GRACEFUL_RELEASE  0x00004000      /* Netipc std bit 17 */
#  define NSF_MESSAGE_MODE      0x40000000      /* Netipc std bit 1  */

   /*
    *   ipccontrol() request values
    */
#  define NSC_NBIO_ENABLE              1    /* */
#  define NSC_NBIO_DISABLE             2    /* */
#  define NSC_TIMEOUT_RESET            3    /* */
#  define NSC_TIMEOUT_GET              4    /*  not in std */
#  define NSC_SOCKADDR                16
#  define NSC_RECV_THRESH_RESET     1000    /* */
#  define NSC_SEND_THRESH_RESET     1001    /* */
#  define NSC_RECV_THRESH_GET       1002    /* */
#  define NSC_SEND_THRESH_GET       1003    /* */
#  define NSC_GET_NODE_NAME         9008    /* */


   /*
    * NS options
    */
#  define NSO_NULL                  (ns_opt_t)0/* null option structure */
#  define NSO_MAX_SEND_SIZE         3
#  define NSO_MAX_RECV_SIZE         4
#  define NSO_MAX_CONN_REQ_BACK     6
#  define NSO_DATA_OFFSET           8
#  define NSO_PROTOCOL_ADDRESS      128
```

```
    /*
     * Netipc errors
     */
#   define NSR_NO_ERROR          000     /* no error occurred            */
#   define NSR_BOUNDS_VIO        3       /* parameter bounds violation   */
#   define NSR_NETWORK_DOWN      4       /* network not initialized      */
#   define NSR_SOCK_KIND         5       /* invalid socket kind value    */
#   define NSR_PROTOCOL          6       /* invalid protocol id          */
#   define NSR_FLAGS             7       /* error in flags parameter     */
#   define NSR_OPT_OPTION        8       /* invalid opt array option     */
#   define NSR_PROTOCOL_NOT_ACTIVE  9    /* protocol not active          */
#   define NSR_KIND_AND_PROTOCOL 10      /* sock kind/protocol mismatch  */
#   define NSR_NO_MEMORY         11      /* out of memory (mbufs)        */
#   define NSR_ADDR_OPT          14      /* illegal proto addr in opt arr*/
#   define NSR_NO_FILE_AVAIL     15      /* no file table entries avail  */
#   define NSR_OPT_SYNTAX        18      /* error in opt array syntax    */
#   define NSR_DUP_OPTION        21      /* duplicate option in opt array*/
#   define NSR_MAX_CONNECTQ      24      /* max cnct rqs err in opt array*/
#   define NSR_NLEN              28      /* invalid name length          */
#   define NSR_DESC              29      /* invalid descriptor           */
#   define NSR_CANT_NAME_VC      30      /* can't name vc socket         */
#   define NSR_DUP_NAME          31      /* duplicate name specified     */
#   define NSR_NAME_TABLE_FULL   36      /* table is full                */
#   define NSR_NAME_NOT_FOUND    37      /* specified name not matched   */
#   define NSR_NO_OWNERSHIP      38      /* user doesn_t own the socket  */
#   define NSR_NODE_NAME_SYNTAX  39      /* invalid node name syntax     */
#   define NSR_NO_NODE           40      /* node does not exist          */
#   define NSR_CANT_CONTACT_SERVER 43    /* can't contact remote server  */
#   define NSR_NO_REG_RESPONSE   44      /* no response from remote reg  */
#   define NSR_SIGNAL_INDICATION 45      /* syscall aborted due to signal*/
#   define NSR_PATH_REPORT       46      /* can't interpret path report  */
#   define NSR_BAD_REG_MSG       47      /* received garbage registry msg*/
#   define NSR_DLEN              50      /* invalid data length value    */
#   define NSR_DEST              51      /* invalid dest descriptor      */
#   define NSR_PROTOCOL_MISMATCH 52      /* source and dest have dif prot*/
#   define NSR_SOCKET_MISMATCH   53      /* dest has wrong type socket   */
#   define NSR_NOT_CALL_SOCKET   54      /* invalid socket descriptor    */
#   define NSR_WOULD_BLOCK       56      /* would block to satisfy req.  */
#   define NSR_SOCKET_TIMEOUT    59      /* timer popped                 */
#   define NSR_NO_DESC_AVAIL     60      /* no file descriptos avail     */
#   define NSR_CNCT_PENDING      62      /* must call IPCRECV            */
#   define NSR_REMOTE_ABORT      64      /* remote aborted the connection*/
#   define NSR_LOCAL_ABORT       65      /* local side aborted the cnct  */
#   define NSR_NOT_CONNECTION    66      /* not a connection descriptor  */
#   define NSR_REQUEST           74      /* invalid IPCCONTROL request   */
#   define NSR_TIMEOUT_VALUE     76      /* invalid in IPCCONTROL        */
#   define NSR_ERRNUM            85      /* invalid netipc error number  */
#   define NSR_VECT_COUNT        99      /* invalid byte count in vector */
```

```
#  define NSR_TOO_MANY_VECTS    100     /* too many vect data descripts */
#  define NSR_DUP_ADDRESS       106     /* address already in use       */
#  define NSR_REMOTE_RELEASED   109     /* graceful release; can't recv */
#  define NSR_UNANTICIPATED     110     /* netipc subsystem is disabled */
#  define NSR_DEST_UNREACHABLE  116     /* unable to reach destination  */
#  define NSR_VERSION           118     /* version number mismatch      */
#  define NSR_OPT_ENTRY_NUM     124     /* bad entry number specified   */
#  define NSR_OPT_DATA_LEN      125     /* bad entry length specified   */
#  define NSR_OPT_TOTAL         126     /* initopt( illegal total       */
#  define NSR_OPT_CANTREAD      127     /* cant read option  readopt(   */
#  define NSR_THRESH_VALUE      1002    /* bad integer to IpcControl    */
#  define NSR_NOT_ALLOWED       2003    /* user not super-user          */
#  define NSR_MSGSIZE           2004    /* message size too big         */
#  define NSR_ADDR_NOT_AVAIL    2005    /* address not available        */

   /* used to map kernel errors to NSR equivalents */
#  define NIPC_ERROR_OFFSET     10000

   /* these exist for historical reasons */
#  define NSO_MIN_BURST_IN              7             /* ignored in 8.0 */
#  define NSO_MIN_BURST_OUT             11            /* ignored in 8.0 */
#  define NSOL_MIN_BURST_IN             2
#  define NSOL_MIN_BURST_OUT            2
#  define NSOL_MAX_CONN_REQ_BACK        2
#  define NSOL_MAX_RECV_SIZE            2
#  define NSOL_MAX_SEND_SIZE            2

#  ifndef MIN
#     define MIN(a,b) (((a)<(b))?(a):(b))
#  endif

#endif /* _INCLUDE_HPUX_SOURCE */

#endif /* _SYS_NS_IPC_INCLUDED */
```