DECnet–ULTRIX

digital

Programming

Order Number: AA–EA88D–TE

DECnet–ULTRIX

Programming

May 1990

This manual offers guidelines for application programming in the DECnet–ULTRIX environment, describes DECnet–ULTRIX system calls and subroutines, and shows DECnet–ULTRIX data structures and programming examples.

Supersession/Update Information:	This is a revised manual.
Operating System and Version:	ULTRIX V4.0
Software Version:	DECnet–ULTRIX V4.0

digital™

Order Number: AA–EA88D–TE

AA–EA88D–TE
May 1990

DEC	PDP	VAX
DECnet	ULTRIX	VMS
DECUS	UNIBUS	digital™

This manual was produced by Networks and Communications Publications.

Contents

Preface

The DECnet–ULTRIX product is layered software that runs on an ULTRIX system. With this software, an ULTRIX system functions as an end node in a DECnet network. The DECnet–ULTRIX product is an end-node implementation of Digital Network Architecture (DNA) Phase IV.

Manual Objectives

Using both tutorial and reference material, this manual show programmers how to write programs for client and server applications in the DECnet–ULTRIX environment.

Intended Audience

This manual is for programmers using DECnet–ULTRIX software to write network applications. The manual assumes the following:

- You are familiar with an editor, such as **vi** or **ed**.
- You have a working knowledge of the C programming language and experience writing system or network programs.
- You are familiar with the ULTRIX system, including its naming conventions, system commands, system calls, and subroutines.

Structure of This Manual

The *DECnet–ULTRIX Programming* manual is divided into two parts, five chapters, and two appendixes.

Part I introduces DECnet–ULTRIX programming concepts and guidelines for application programming in the DECnet–ULTRIX programming environment:

Chapter 1	Describes DECnet–ULTRIX programming concepts.
Chapter 2	Describes DECnet–ULTRIX programming tools and how they work in the DECnet–ULTRIX programming environment.
Chapter 3	Explains how to write programs for clients and server applications in the DECnet–ULTRIX programming environment.

Part II contains descriptions and other reference information about the DECnet–ULTRIX system calls and subroutines:

Chapter 4	Describes the DECnet–ULTRIX system calls.
Chapter 5	Describes the DECnet–ULTRIX subroutines.

The appendixes show DECnet data structures and programming examples.

Appendix A	Contains the DECnet–ULTRIX data structures.
Appendix B	Contains DECnet–ULTRIX programming examples.

Related Documents

To supplement the *DECnet–ULTRIX Programming* manual, refer to the following manuals:

- *DECnet–ULTRIX Release Notes*

  This document contains miscellaneous information and updates not included in other books in the DECnet–ULTRIX documentation set.

- *DECnet–ULTRIX DECnet–Internet Gateway Use and Management*

  This manual describes the DECnet–Internet Gateway and contains directions for installing, using, and managing, it.

- *DECnet–ULTRIX Network Management*

  This manual defines the DECnet–ULTRIX network databases and components. It describes the Network Control Program (ncp) and how it is used to configure, monitor, and test your network. Other topics include loopback testing, event logging, and instructions for displaying network information.

- *DECnet–ULTRIX NCP Command Reference*

  This reference manual describes the ncp commands used for defining, monitoring, and testing your network.

- *DECnet–ULTRIX Installation Guide*

  This manual describes procedures for installing a DECnet–ULTRIX node and testing it for proper operation. This manual also lists the DECnet–ULTRIX distribution files and the path names to which they are installed.

- *ULTRIX Guide to the Data Link Interface (DLI)*

  This manual describes procedures for using DLI to write application programs at the data link layer.

- *ULTRIX Introduction to Network Programming*

  This manual describes network programming concepts for programming in the ULTRIX environment.

To obtain a detailed description of DNA, refer to the *DECnet Digital Network Architecture (Phase IV), General Description.*

Graphic Conventions

This manual uses the following graphic conventions:

Convention	Meaning
special	Command options, system calls, subroutines, and data structures appear in **special type**.
command( )	Cross-references to specific command documentation include the section number in the reference manual where the commands are documented. For example: See the **socket(2dn)** system call. This indicates that you can find the material on the **socket** system call in Section 2dn of the reference pages.
`literal`	Indicates terms that are constant and must be typed just as they are presented.
[ ]	Square brackets indicate optional arguments. Do not type the brackets.
...	Horizontal ellipsis points indicate that the preceding item can be repeated one or more times. **NOTE** In examples, vertical ellipsis points represent either user input or system input that has been omitted to emphasize specific information.
lowercase/ UPPERCASE	Because DECnet–ULTRIX software is case-sensitive, you must type all literal input in the case shown. UPPERCASE is also used for the names of all DECnet nodes, including DECnet–ULTRIX nodes. This convention follows DECnet protocol, which names and recognizes all nodes in UPPERCASE. However, node names are not case-sensitive and need not be typed in the case shown.
`example`	Indicates an example of system output. System output is in black type; user input is in red type.
italics	Indicate a variable, for which either you or the system must specify a value.
%	The default user prompt in multiuser mode.
#	The default superuser prompt.
key	Indicates a key on your keyboard. CTRL/*key* represents a CONTROL key sequence, where you press the CONTROL key at the same time as the specified key.

Other conventions are as follows:

- All numbers are decimal unless otherwise noted.
- All Ethernet addresses are hexadecimal.

Part I

Overview

Chapter 1

Introduction to the DECnet–ULTRIX Programming Environment

This chapter introduces some of the DECnet programming concepts on which the DECnet–ULTRIX programming interface is based. All terms and concepts in this chapter are presented in the context of the DECnet–ULTRIX programming environment.

For more information about programming in the ULTRIX environment, see the *ULTRIX Network Programming Guide* and the article "A 4.2 BSD Interprocess Communication Primer," in the *ULTRIX Supplementary Documentation, Volume III.*

1.1 DECnet–ULTRIX Programming Interface

The DECnet–ULTRIX programming interface lets you write cooperating programs that exchange data over a DECnet network. The interface provides the following support:

Client–server communication. This is sometimes called task-to-task communication. The client application initiates a connection and requests services from the server application. The server application either accepts or rejects the request. Client-server communication lets DECnet–ULTRIX Phase IV applications communicate with remote Phase III and Phase IV DECnet applications through a socket-level programming interface.

DECnet and TCP/IP coexistence. DECnet protocols and Transmission Control Protocol/Internet Protocol (TCP/IP) coexist and can share system resources, including Ethernet and Digital Data Communications Message Protocol (DDCMP) hardware. You can modify most TCP/IP programs to use DECnet protocols, or DECnet programs to use TCP/IP protocols. You can use DECnet and TCP/IP simultaneously on an Ethernet and alternate between the two protocols on DDCMP point-to-point lines.

File access. Programs on any other DECnet Phase III/IV system can access DECnet–ULTRIX files for sequential reading, writing, directories, or deletion.

Access Control. DECnet–ULTRIX supports two ways for your client application to gain access to the server: access-control information and proxy.

See Sections 3.1.3 and 3.1.4 for instructions on how to use proxy and access-control information in a connection request.

1.2 Network Objects

In the DECnet–ULTRIX programming environment, a network object is a server application that can be accessed by name or number from other DECnet nodes. A client application identifies the server application it wants to connect to by specifying the server's object name or number as part of a connection request.

See the *DECnet–ULTRIX NCP Command Reference* manual for examples of network objects.

1.3 Communication Domains

A communication domain is a set of protocols that have common communication properties. DECnet–ULTRIX introduces the DECnet domain into the ULTRIX Interprocess Communication (IPC) environment for applications that communicate through the DECnet standard protocols.

1.4 Sockets

A socket is an addressable endpoint for communication. The client and server applications each create a socket that acts as a handle for sending and receiving data.

Each communication domain supports a different set of socket types. The DECnet communication domain supports the following socket types for DECnet–ULTRIX applications:

Sequenced-packet sockets. A sequenced-packet socket supplies a bidirectional, reliable, ordered, first-in,first-out (FIFO), unduplicated flow of data.

The socket preserves the record boundaries. A **write** operation transmits one message across the connection; a **read** operation—if it completes successfully—returns a single, logical message.

Stream sockets. A stream socket also supplies a bidirectional, reliable, ordered, FIFO, unduplicated flow of data.

Stream sockets provide a byte stream without using message boundaries. Data supplied as part of a **write** operation may or may not transmit a message across the connection, and a **read** operation may return data from one or more data packets. These possibilities depend on system variables unknown to the program.

1.5 Blocking and Nonblocking Input/Output Modes

Blocking and nonblocking are input/output (I/O) modes that cause a calling process to either wait (blocked) or not wait (nonblocked) for an I/O operation. Blocking prevents an I/O system call from returning control to a calling procedure until the operation completes. The nonblocking I/O mode returns control to the calling procedure immediately with an error message if there are not enough resources available to complete the operation.

1.6 Accept-Immediate and Accept-Deferred Modes

DECnet supports two modes for accepting incoming connections: immediate and deferred.

> **Accept-Immediate mode** makes it possible for the server program to send and receive data as soon as the accept call operation completes. However, in this mode, the server does not have access to any optional data or access-control information that may have been supplied with the connection request.
>
> **Accept-Deferred mode** lets the server program store, examine, and process any access-control information or optional data that is supplied as part of a connection request. The server must then accept or reject the connection.
>
> As long as the socket is in accept-deferred mode, a server program can retrieve access-control information or retrieve and return optional data when a connect is pending; that is, after an **accept** call has successfully completed, but before the server accepts or rejects the connection.

1.7 Access-Control Information

The DECnet architecture lets the client requesting the connection pass access-control information to a server application. The server application then uses this information to determine if access should be granted. This information consists of three strings: *username, password*, and *account*. These are defined as follows:

username	A name of up to 39 characters assigned to the user on the server system.
password	A string of up to 39 characters that you use to gain access to the user account on the server system.
account	A string of up to 39 characters that some DECnet systems use to identify the remote users and their privileges upon logging in. The server ignores this string if it is not required.

Different servers interpret and use these strings according to their own requirements. In many cases, servers compare received access-control information against the system password file. Usually, the results of this comparison determine whether a connection request is accepted and, if it is, what privileges and quotas are allowed.

1.8 Proxy Access

In the DECnet domain, proxy access is a method of screening client application access to the server application without supplying a password.

When the client requests a connection, the node on which the client resides passes the identity of the client application to the target node on which the server resides. The supplied name (a log-in name or user ID) of the user initiating the request for the client must correspond with an entry listed in the target node's proxy access file.

This procedure is more secure than sending a password over the network.

1.9 Optional Data

In the DECnet domain, optional data is a string of up to 16 bytes that clients and servers can exchange on either a connect or disconnect sequence. This data is interpreted differently according to the application.

Some application protocols exchange additional identifying information (such as a protocol version number) at the time of the connection. This information is used to determine whether a connection request should be accepted. For example, DECnet Network Management uses optional data to exchange protocol version numbers before a connection is established. Also, when a socket is disconnected or a connection request is rejected, the application may use optional data to send an error message.

1.10 Out-of-Band Messages

An out-of-band message is an unsolicited, high-priority message that one application sends to another outside of the normal data channel. In most cases, it informs the receiving application of an unusual or abnormal event in the sending application.

Chapter 2

DECnet–ULTRIX Programming Tools

This chapter describes some of the tools for writing DECnet–ULTRIX client and server applications. It explains how the tools work and recommends methods of using them in the DECnet–ULTRIX programming environment.

Tools for programming client and server tasks in the DECnet domain include:

- The DECnet library (**libdnet.a**), which contains subroutines that simplify many basic programming operations. Two important subroutines included in this library are:
  - The **dnet_conn** subroutine, a routine that establishes a connection to a specified network object on a remote node.
  - The **dnet_eof** subroutine, a routine that tests the state of an established connection to a remote DECnet application.

  NOTE

  When you build programs that use these routines, you must specify **-ldnet** in the command line or makefile. Versions of the libraries suitable for use by **lint** are contained in the unsupported subset. See the *DECnet–ULTRIX Installation* manual for the subset name and location.

- The object spawner, a server program that listens for connection requests on behalf of all servers that are not actively listening for connection requests.
- System calls used within the DECnet domain to perform network connection and data transfer functions. Examples of these calls are **accept**, **bind**, **write**, and **close**.

2.1 How the dnet_conn Subroutine Works

When you use **dnet_conn** to establish a connection for a client application, the subroutine performs connection tasks in the following order:

1. Creates a socket in the DECnet domain.
2. Formats access-control information and optional connection data.
3. Issues a connection request to a server application.
4. Returns optional data received from the server if the connection is established.

The **dnet_conn** subroutine accepts the following as input:

- The name of the node to which you connect.

- The name or number of the server on the node.
- The socket type.
- A buffer for outgoing optional data, and a buffer for incoming optional data.

The **dnet_conn** subroutine also lets you pass access-control information to the server by appending it to the node name.

NOTE

> The name of the node supplied to **dnet_conn** may be a node alias as defined in the **.nodes** file. Programs that use **dnet_conn** will prompt you for a password if you choose to omit the password field in an access-control string. To provide account security, the password that you type after the prompt does not echo.

If a connection is established successfully, **dnet_conn** returns a socket descriptor that can be used for subsequent **read** and **write** operations. If an error is encountered, **dnet_conn** returns a -1 value with additional error detail available in the external variable **errno**. Use the **nerror** system call to print out relevant DECnet error messages.

NOTE

> The **dnet_conn** subroutine no longer returns the ULTRIX diagnostic message [ECONNABORTED] or [ECONNRESET]. If DECnet is not installed on the system, the **socket** request that **dnet_conn** makes will fail with the ULTRIX error message [EPROTONOSUPPORT], which is equivalent to the DECnet error message, "Protocol not supported."

2.2 How the dnet_eof Subroutine Works

When you use **dnet_eof** to test the state of an established connection, the subroutine performs the following steps:

1. Tests a DECnet socket to determine if an end-of-file (EOF) condition exists.
2. Returns a value of 0 if it determines a connection is in an active state.
3. Returns a nonzero value if it determines that a connection is in an inactive state.

This subroutine is useful for determining if any data exists for a **read** operation and if the socket is connected.

See **dnet_eof(3dn)** for more information about how to use the **dnet_eof** subroutine.

2.3 How the DECnet Object Spawner Works

The logic sequence for using the object spawner follows:

1. The object spawner creates a socket in the DECnet domain and listens for incoming connection requests on behalf of multiple DECnet objects (named and numbered).
2. When the object spawner receives a request to connect to an object, the spawner checks the object database to verify that either the server program or the **default** object is defined. If neither is found, it rejects the connection.

3. If access-control information is specified with the connection request, the object spawner verifies the information with the system password file. If the information is invalid, the connection is rejected. If it is valid, go to step 7.
4. If proxy is requested with the connection request, the object spawner verifies the proxy request with the system proxy file. If no entry is found, the object spawner uses the default user associated with the object entry for the server.
5. If access-control information is not specified or proxy is not requested, the object spawner uses the default user associated with the object entry for the server.
6. If the default user is specified for the object entry and defined in the system password file, the connection is accepted. Otherwise, the connection is rejected.
7. The object spawner redirects standard input and standard output to the network connection.
8. The object spawner executes the server program after setting up the environment. The environment is based on information in the password file entry of the user through which access was granted to the object. In addition, the setting for the process group is set equal to the process ID, and the group access list is initialized. Standard error is then redirected to **/dev/null**.

The logic sequence for server programs using the object spawner depends on the specified mode of acceptance, as follows:

1. If the accept mode is immediate, the object spawner completes the server connection. The server program can then exchange data by reading standard input and writing standard output.
2. If deferred mode was chosen, the connection request must be completed by the server using either standard input or standard output as the socket handle. The server program uses the **getsockopt** and **setsockopt** calls to complete the connection, as shown in steps 6 through 8 of the system call logic sequence described in Section 2.4.1.

2.4 How DECnet–ULTRIX System Calls Work

The following sections describe how client and server applications use system calls to establish a connection, exchange data, and terminate a connection.

2.4.1 Using System Calls for Client Programs

The following list shows how a client application can use system calls to establish a network connection:

1. To initiate a connection, a client program creates a socket for the connection by issuing a **socket** call. This call creates a socket of the specified type in the DECnet domain. The **socket** call returns a descriptor for the socket, which is used for subsequent program requests.
2. To set up access control, use one or both of the following methods:
    a. To pass access-control information, issue the **setsockopt** call on the descriptor returned from the socket call with the option DSO_CONACCESS. The structure **accessdata_dn**, which contains the data you have specified, is passed as a parameter to the call.

b. To use proxy access, set the SDF_PROXY bit in the *sdn_flags* field of the **sockaddr_dn** structure before issuing the **connect** call. If the program is to be executed with superuser privileges, you may want to bind a name to be used as the proxy source name to the socket, before issuing the **connect** call.

3. To pass optional data, issue **setsockopt** again with the option DSO_CONDATA. The structure **optdata_dn**, which contains the data you have specified, is passed as a parameter.
4. A client program issues a **connect** call to request a connection to a specified object. If the preceding **setsockopt** calls were successful, DECnet will use the data you have supplied when the **connect** call is issued.
5. The client program can issue a **getsockopt** call to retrieve incoming optional data.
6. If the **connect** call is successful, the program can use the socket to send and receive data by means of the **read** and **write** or **recv** and **send** calls.
7. The program can use the **getsockopt** or **setsockopt** call to send or receive optional data with the **close** call.
8. The **close** call terminates the connection.

Table 2-1 summarizes the logic sequences for a typical DECnet–ULTRIX client application using system calls. See Appendix B for programming examples.

Table 2–1: Client Program Calling Sequences

Function	System Calls
Create a socket for the connection.	**socket**
Send optional data and/or access-control information with the connection request.	**setsockopt***
Define a name for the socket.	**bind***
Request a connection to a server program.	**connect**
Retrieve optional data from the server.	**getsockopt** *
Transfer normal data.	**send** **recv** **read** **write**
Transfer out-of-band data.	**send** * **recv** *
Send or receive optional data with the **close** call	**setsockopt*** **getsockopt** *
Terminate the connection.	**close**

*Optional.

2.4.2 Using System Calls for Server Programs

The following list describes the logic sequence for a typical DECnet–ULTRIX server program using system calls to establish a connection.

1. The server program creates a socket in the DECnet domain by issuing a **socket** call.

2. The object name or number is stored in the **sockaddr_dn** structure. The server issues a **bind** call to assign the object name or number. (See the description of the **bind** call in **bind(2dn)**).
3. A server can issue a **setsockopt** call to set the mode of acceptance to deferred.
4. A server issues a **listen** call, which declares that the socket is available for receiving connection requests directed to the bound name.
5. An **accept** call completes when the system receives a connection request. (Note that an **accept** call in deferred mode must be issued to receive a request, but does not actually accept the connection.) If the **accept** is successful, a new server socket is created.

   NOTE

   If you specified accept-immediate mode, you can use the socket to send and receive data. If you specified accept-deferred mode, however, you must complete the following steps before attempting to transfer data.

6. A server issues a **getsockopt** call to retrieve any access-control information or optional data that was supplied with the **connect** call.
7. A server issues a **setsockopt** call to supply any optional data that it wants to return to the client program.
8. The server issues a **setsockopt** call to accept or reject the connection.
9. If the server accepts the connection, the program can use the socket to send and receive data by means of the **read** and **write** or **recv** and **send** calls.
10. The program can use the **getsockopt** or **setsockopt** call to send or receive optional data with the **close** call.
11. The **close** call terminates the connection.

Table 2-2 summarizes the calling sequences for a typical DECnet–ULTRIX server application using system calls. See Appendix B for programming examples.

Table 2–2: Server Program Calling Sequences

Function	System Calls
Create a socket to listen for connection requests.	**socket**
Define a name for the socket.	**bind**
Set the mode of acceptance. The default mode is IMMEDIATE, and the other possible mode is DEFERRED.	**setsockopt** *
Declare the socket available for connection requests.	**listen**
Block the server program until it receives a connection request.	**accept**
When in accept-deferred mode, receive optional data or access-control information.	**getsockopt** *
When in accept-deferred mode, supply optional data, such as the server software version number.	**setsockopt** *

*Optional

(continued on next page)

Table 2–2 (Cont.): Server Program Calling Sequences

Function	System Calls
When in accept-deferred mode, accept or reject the connection.	setsockopt *
Transfer normal data.	send recv read write
Transfer out-of-band data.	send* recv *
Send or receive optional data with the close call.	setsockopt * getsockopt *
Terminate the connection.	close

*Optional

2.5 Three Ways to Use DECnet–ULTRIX Programming Tools

You can use DECnet–ULTRIX tools to establish a connection between client and server applications in three ways:

- Let DECnet–ULTRIX subroutines and the DECnet object spawner handle programming tasks for you.
- Use system calls to perform the same tasks yourself.
- Combine DECnet–ULTRIX subroutines and DECnet object spawner functions with system call functions.

After you have established a connection between the client and server, you can use system calls and subroutines to perform data transfer tasks and disconnect the network connection.

2.5.1 Using dnet_conn with the DECnet Object Spawner

You can let the **dnet_conn** subroutine and the object spawner establish the connection between the client and server. **dnet_conn** initiates the connection and performs connection tasks for the client application. The object spawner performs connection tasks on behalf of the server.

The object spawner is the recommended tool for server programs because it provides the following services:

- Eliminates the need for coding connection request processing, including access-control information and proxy handling, in the server program.
- Reduces the number of idle processes because it listens on behalf of multiple servers.

See Appendix B for a programming example that shows how you can use **dnet_conn** and the object spawner to establish a connection between client and server applications.

2.5.2 Using DECnet–ULTRIX System Calls

You can use DECnet–ULTRIX system calls to establish a session and accept connection requests. The system calls can perform all the tasks that **dnet_conn** performs for you. They can also give you more programming flexibility by letting you control each task during the connection process.

See Appendix B for programming examples.

2.5.3 Combining DECnet–ULTRIX Tools

You can combine tools to establish a session. For example, you can use **dnet_conn** to initiate a connection request for the client and use the system calls to accept the request for the server. Also, you can use system calls to initiate a connection request for the client and let the object spawner accept the request for the server. Table 2-3 shows four ways to establish a session with DECnet–ULTRIX tools.

Table 2–3: Methods for Establishing a Client–Server Session

Client Application Task	Server Application Task
dnet_conn requests a session.	DECnet object spawner establishes the connection.
dnet_conn requests a session.	System calls accept or reject the request.
System calls request a session.	DECnet object spawner accepts the request.
System calls request a session.	System calls accept or reject the request.

Chapter 3

Programming in the DECnet Domain

This chapter explains procedures for writing DECnet–ULTRIX applications by using either subroutines or system calls. The following sections explain how to perform four basic network application programming tasks:

- Program a client application to initiate a network connection.
- Program a server application to respond to a network connection request.
- Perform data transfer tasks after establishing a connection.
- Disconnect a session between a client and server application.

3.1 How to Program a Client-Initiated Connection

To initiate a network connection between your client application and a remote DECnet application, you can use **dnet_conn** or system calls to:

1. Choose the socket type for the client.
2. Identify the node and server to which the client is attempting to connect.
3. Set up either access-control or proxy for client access to servers.
4. Send and receive optional data.

3.1.1 Choosing a Socket Type for the Client

DECnet supports two socket types: sequenced-packet sockets and stream sockets. Table 3-1 compares these sockets.

Table 3–1: Socket Types Compared

Sequenced-Packet Socket	Stream Socket
Preserves message boundaries	Does not preserve message boundaries
DECnet-VAX default socket	Commonly used for ULTRIX applications
Not available on TCP/IP	Available on TCP/IP

When writing applications, use the same socket type as the program you connect to uses. If a connection uses both types of sockets, the program using the stream socket:

- Has no control over how much data the sequenced-packet socket will get when it performs its next **read** operation.

- Does not receive any indication of the record boundary on a **read** operation– even though the program using the sequenced-packet socket creates a record boundary with each **write** operation.

Applications that use message boundaries to interpret data cannot be used in connections using both stream and sequenced-packet sockets. If you cannot guarantee that both ends of a connection will use the same socket type, design an application protocol that does not use record boundaries.

3.1.1.1 Using dnet_conn

To specify the socket type as an argument in **dnet_conn**, use this format:

s=dnet_conn(*node,object,type,*)

where

type	is either SOCK_STREAM, if you want to specify a stream socket, or SOCK_SEQPACKET, if you want to specify a sequenced-packet socket. If the socket type is set to 0, use the default, SOCK_SEQPACKET.

EXAMPLE:

This example shows a sequenced-packet socket being selected for node NAVAHO with object 17.

```
s=dnet_conn(NAVAHO,17,SOCK_SEQPACKET,)
```

For more information about socket types, see the description in **dnet_conn(2dn)**.

3.1.1.2 Using System Calls

To specify a socket type during a **socket** call operation, use this format:
s=socket(*node,object,type,*)

where

type	is either SOCK_STREAM, if you want to indicate a stream socket, or SOCK_SEQPACKET, if you want to indicate a sequenced-packet socket.

EXAMPLE:

This example shows a sequenced-packet socket being selected for a DECnet node.

```
s=socket(AF_DECnet,SOCK_SEQPACKET,)
```

For more information about socket types, see the description in **socket(2dn)**.

3.1.2 Specifying a Node and Server

Before your client application can request a connection, you must:

1. Identify the node on which the server resides.

   You can use a node name (an alphanumeric string of one to six characters) or node address (an area number from 1 to 63, followed by a period and a node number from 1 to 1,023).

2. Identify the server you want to connect to.

   The client application can specify the server application in one of two ways:

   - By a network object name of up to 16 characters.
   - By a network object number from 1 to 255.

If the remote object is defined (that is, an object number has been assigned to the server process—either by Digital or by the user), the object number is the recommended method for requesting the network service to avoid any conflicts in naming conventions.

See the *DECnet–ULTRIX NCP Command Reference* manual for detailed information on preassigned network object numbers and procedures for defining network objects in the object database.

3.1.2.1 Using dnet_conn

To specify a node and server while requesting a connection, use the following format:

dnet_conn(*node,object,*)

where

node	is either the node name or address used to specify the node.
object	is either the object name or the object number used to specify the server.

The following examples show you how to use **dnet_conn** to specify a node and a server by name and number while establishing a connection.

EXAMPLE 1:

In this example, the client program uses **dnet_conn** to connect to object number 17 on node NAVAHO.

```
s=dnet_conn("navaho","#17",)
```

EXAMPLE 2:

In this example, the client program uses **dnet_conn** to connect to object **xyz** on the node with a DECnet address 55.342.

```
s=dnet_conn("55.342","xyz",)
```

3.1.2.2 Using System Calls

To specify the node and server, you must use the **sockaddr_dn** data structure.

EXAMPLE 1:

In this example, the client program connects to server **xyz** on node 55.342:

```
#define SERVERNAME "xyz" ❶
#define NODE       "55.342"
       .
       .
       .
   struct sockaddr_dn sockaddr; ❷
   struct dn_naddr*node_addr;
   int sock;
       .
       .
       .
   bzero(&sockaddr, sizeof(sockaddr));

   sockaddr.sdn_family = AF_DECnet; ❸

   sockaddr.sdn_objnamel = strlen(SERVERNAME);  ❹
       strncpy(sockaddr.sdn_objname,SERVERNAME,
   sizeof(sockaddr.sdn_objname));

   node_addr = dnet_addr(NODE);  ❺

   sockaddr.sdn_add = *node_addr;

   if (connect(sock, &sockaddr, sizeof(sockaddr)) < 0)
   {
           perror("connect");
           exit(1);
   }
       .
       .
       .
```

COMMENTS:

❶ Defines the server using a network object name. This name can be up to 16 characters long.

❷ The **sockaddr_dn** data structure is defined in the file **/sys/netdnet/dn.h** (see Appendix A).

❸ You must specify the AF_DECnet address family.

❹ These three lines show how the server name is put into the **sockaddr_dn** structure.

❺ If you are using the **dnet_addr** subroutine to specify a node in the **sockaddr_dn** data structure, you must use a node address.

EXAMPLE 2:

This example shows you how to connect to server 17 on node NAVAHO by number.

```
#define SERVERNUMBER 17 ❶
#define NODENAME     "navaho"
         .
         .
         .
struct sockaddr_dn sockaddr;  ❷
struct nodeent *nodep;
int sock;
         .
         .
         .
bzero(&sockaddr, sizeof(sockaddr));

sockaddr.sdn_family = AF_DECnet;  ❸
sockaddr.sdn_objnum = SERVERNUMBER;

nodep = getnodebyname(NODENAME);  ❹
bcopy(nodep->n_addr, sockaddr.sdn_nodeaddr, nodep->n_length);  ❺
sockaddr.sdn_nodeaddrl = nodep->n_length;

if (connect(sock, &sockaddr, sizeof(sockaddr)) < 0)
{
        perror("connect");
        exit(1);
}
         .
         .
         .
```

COMMENTS:

❶ Defines the server using a network object number. This can be any number from 1 to 255.

❷ The **sockaddr_dn** data structure is defined in the file **/sys/netdnet/dn.h** (see Appendix A).

❸ You must specify the AF_DECnet address family.

❹ If you are using the **getnodebyname** subroutine to specify a node in the **sockaddr_dn** structure, you must specify a node name.

❺ These three lines show how to fill in the node address fields of the **sockaddr_dn** data structure.

See Appendix B for more detailed programming examples.

3.1.3 Specifying Access-Control Information

You can use either the **dnet_conn** subroutine or system calls to specify access-control information for the client application.

3.1.3.1 Using dnet_conn

To specify access-control information, use the following format:
dnet_conn(*"node/username[/password][/account]"*)

where

node	is a string that specifies the node name or address, followed by the *username*, *password*, and *account* strings separated by slashes (/).
username	is a name of up to 39 characters assigned to the user on the server system.
password	is a string of up to 39 characters that you use to gain access to the user account on the server system.
account	is a string of up to 39 characters used by some DECnet systems. The server ignores this string if it is not required.

3.1.3.2 Using System Calls

To specify access-control information, issue a **setsockopt** call with the DSO_CONACCESS option. The access-control data is passed in the **accessdata_dn** data structure (described in Appendix A). The access-control information is used when the client issues the **connect** call.

EXAMPLE:

This example shows a **setsockopt** call being issued with a DSO_CONACCESS option in the **accessdata_dn** structure.

```
set_access_control(socket, user, password) ❶
int socket;
char *user, *password;

{

     struct accessdata_dn acc_data;

     bzero(&acc_data, sizeof(acc_data));

     acc_data.acc_userl = strlen(user);
     strncpy(acc_data.acc_user, user, acc_data.acc_userl);

     acc_data.acc_passl = strlen(password);                     ❷
     strncpy(acc_data.acc_pass, password, acc_data.acc_passl);

     return(setsockopt(sock, DNPROTO_NSP, DSO_CONACCESS,
            &acc_data, sizeof(acc_data)));

}
```

COMMENTS:

❶ The lengths of the user name and password are defined in **/sys/netdnet/dn.h**.

❷ The *account* string is not used in this example.

NOTE

The **setsockopt** call must precede the **connect** call to supply access information for the connection request.

After the client issues a **connect** call, DECnet flushes any access-control information previously set with the **setsockopt** call and the DSO_CONACCESS option. Therefore, you must specify new access-control data for any subsequent connection requests that the client issues on the same socket.

3.1.4 Requesting Proxy

You can use either the **dnet_conn** subroutine or system calls to request proxy for a client application.

3.1.4.1 Using dnet_conn

The **dnet_conn** subroutine requests proxy by default. If the default (proxy) setting is not changed, **dnet_conn** binds the user's log-in name (converted to uppercase) to the socket. This bound name is used as the source name for the outgoing connection only when a program's user ID is set to root or invoked by the superuser. Otherwise, the ASCII form of the user's ID is used as the source name for proxy access.

If you do not want **dnet_conn** to request proxy access at the remote system, set the external variable, *proxy_requested*, equal to zero.

EXAMPLE:

In this example, the *proxy_requested* variable is set to zero.

```
extern char proxy_requested;
proxy requested=0;
```

3.1.4.2 Using System Calls

To request proxy, set the SDF_PROXY bit in the *sdn_flags* field of the **sockaddr_dn** structure before issuing the **connect** call.

EXAMPLE:

In this example, the client program issues a request for proxy access.

```
#define SERVERNAME "xyz"
#define NODE       "55.342"
    .
    .
    .
    struct sockaddr_dn sockaddr;
    struct sockaddr_dn bindaddr;
    struct dn_naddr *node_addr;
    char *user_name, *getlogin();
    int sock, len, status;
    .
    .
    .
    bzero(&sockaddr, sizeof(sockaddr));
    bzero(&bindaddr, sizeof(bindaddr));

    if (((user_name = getlogin()) == NULL) || (*user_name == NULL))
        user_name = "anonymous";

    bindaddr.sdn_family = AF_DECnet;
    len = strlen(user_name);
    if (len > sizeof(bindaddr.sdn_objname))
        len = sizeof(bindaddr.sdn_objname); ❶
    bindaddr.sdn_objnamel = len;
    strncpy(bindaddr.sdn_objname, user_name, len);
```

```
if (bind(sock, &bindaddr, sizeof(bindaddr)))
{
    perror("bind");
    exit(1);
}
sockaddr.sdn_flags |= SDF_PROXY; ❷
sockaddr.sdn_family = AF_DECnet;
sockaddr.sdn_objnamel = strlen(SERVERNAME);
strcpy(sockaddr.sdn_objname, SERVERNAME);

node_addr = dnet_addr(NODE);

sockaddr.sdn_add = *node_addr;
.
.
.
status = connect(sock, &sockaddr, sizeof(sockaddr));
.
.
.
```

COMMENTS:

❶ Bind a name to the socket before issuing the **connect** call.

❷ If your program is running with superuser privileges, the name you bind to the socket is used as the source name for proxy access.

NOTE

If you do not bind a name to the socket, or if you issue a **connect** call without root privileges, DECnet–ULTRIX uses the user's ID in ASCII as the source name for proxy access.

3.1.5 Setting Up Optional Data for the Client

Use **dnet_conn** or system calls to exchange up to 16 bytes of optional data while a connection is being established.

3.1.5.1 Using dnet_conn

The **dnet_conn** subroutine lets you specify a buffer containing optional data to be sent to the server. It also lets you specify a buffer containing any optional data returned by the server. After a client program sends optional data with a connection request, DECnet flushes the data.
To specify optional data, use the following format:

dnet_conn (node,object,type,*opt_out,opt_outl,opt_in,opt_inl***)**

where

opt_out	specifies the address of the outgoing data.
opt_outl	specifies the length of the outgoing data.
opt_in	specifies the address of the buffer that will store the optional data returned by the server.
opt_inl	is the address of an integer that specifies the size of that buffer before the call and will contain the actual number of bytes of optional data returned by the server on successful completion of **dnet_conn**.

EXAMPLE 1:

You can specify no outgoing optional data to be supplied and no incoming optional data to be expected. For example, when using a sequenced-packet socket to connect to object SOAPBOX on node ALEXUS, issue the following call:

```
s=dnet_conn("alexus","soapbox", SOCK_SEQPACKET,0,0,0,0);
```

EXAMPLE 2:

In this example, the client program connects to the network management object **nml** (object number 19).

```
char in_data[16], out_data[] = {4,0,0}; ❶
int in_length, out_length = sizeof(out_data);
int sock;
    .
    .
    .
    sock = dnet_conn("alexus/root", ❷
                    "#19", 0, ❸
                    out_data, out_length, ❹
                    in_data, &in_length); ❺
    .
    .
    .
```

COMMENTS:

❶ This example shows the current version of the NICE protocol used by **nml**.

❷ This call would be issued if you wanted to connect to node ALEXUS as user ROOT.

❸ In this example, a socket type of 0 defaults to SOCK_SEQPACKET.

❹ The protocol version number is sent as optional data and a version number is expected in return.

❺ A buffer is provided in which the **nml** object can return its protocol version number.

3.1.5.2 Using System Calls

To specify that optional data is to be sent to the server, use the **optdata_dn** structure. To set up the connection data to send to the server, use the **setsockopt** call.

NOTE

You must set the optional data each time you reissue the connection request.

EXAMPLE 1:

In this example, the client program sends three bytes of optional data to the server.

```
char version[] = {4, 0, 0};
        .
        .
        .
        struct optdata_dn out_opt; ❶
        int sock;
        .
        .
        .
        bzero(&out_opt, sizeof(out_opt));
```

```
out_opt.opt_optl = sizeof(version);                             ❷
bcopy(version, out_opt.opt_data, out_opt.opt_optl);

if (setsockopt(sock, DNPROTO_NSP, DSO_CONDATA,
              &out_opt, sizeof(out_opt)) < 0)
{
        perror("setsockopt");
        exit(1);
}
.
.
.
```

COMMENTS:

❶ Defined in the **/sys/netdnet/dn.h** file.

❷ Optional data is copied into the **optdata_dn** structure. The data length is also put into this structure.

After the connection has been established, use the **getsockopt** call to retrieve the data returned by the server program.

EXAMPLE 2:

In this example, up to 16 bytes of data are placed in *in_opt* and the data count is placed in *in_opt_len.*

```
struct optdata_dn in_opt;
int sock, in_opt_len;
.
.
.
bzero(&in_opt, sizeof(in_opt));

in_opt_len = sizeof(in_opt);

if (getsockopt(sock, DNPROTO_NSP, DSO_CONDATA,
              &in_opt, &in_opt_len) < 0)
{
        perror("getsockopt");
        exit(1);
}
.
.
.
```

3.2 How to Establish a Connection for the Server

The DECnet domain supports two methods for programming the server:

- Use the DECnet object spawner to listen for connection requests on behalf of your server. When the spawner receives a request for your server, it executes a copy of your server and redirects standard input and standard output to the network connection.
- Use system calls to set up a server that can run independently as a daemon.

You can use either method to perform the following tasks:

- Specify the socket type: stream or sequenced-packet.
- Assign a name to the server.
- Select an accept mode: immediate or deferred.
- Verify remote user access to the server.

- Exchange optional data with client applications.

3.2.1 Choosing a Socket Type for the Server

When writing a server application, you must choose either of the two available socket types: sequenced-packet or stream. (Table 3-1 compares the characteristics of each socket type.)

NOTE

Be sure to use the same socket type as for the client application.

3.2.1.1 Using the Network Control Program (NCP)

The DECnet object spawner uses the object database, which consists of entries defined by the network manager. One of the characteristics defined for the object entry is the socket type. Use **ncp** commands to choose between a stream socket and sequenced-packet socket. For example, to define a stream socket as the socket type for an object entry, you can use the **ncp** command **set object**:
set object *object-name type*
where

object *object-name*	Specifies that parameters are to be created or modified for the named object only (a maximum of 16 alphanumeric characters).
type	is either SOCK_STREAM, if you are specifying a stream socket, or SOCK_SEQPACKET, if you are specifying a sequenced-packet socket.

EXAMPLE:

This example shows how to use the **set object** command to choose a socket type for object entry **myserver**.

```
> ncp RET
ncp> set object myserver type stream RET
```

See the *DECnet–ULTRIX NCP Command Reference* manual for more information about using **ncp** commands.

3.2.1.2 Using System Calls

To specify a socket type, use the following format:

s=socket (*node,object,type,*)

where

type	is either SOCK_STREAM, if you are specifying a stream socket, or SOCK_SEQPACKET, if you are specifying a sequenced-packet socket.

EXAMPLE:

This example shows how to use the **socket** call to specify a socket type.

```
s=socket (AF_DECnet,SOCK_STREAM,0,);
```

3.2.2 Assigning a Name to Your Server

All server applications must have an object name or number associated with them. Object names contain 1 to 16 alphanumeric characters. Object numbers range from 0 to 255; however, some numbers are reserved for certain types of applications. Table 3-2 shows these assignments:

Table 3–2: Object Number Assignments

Object Number	Type of Application
1-127	Reserved for DECnet-supplied programs, such as **fal** and **dlogin**.
128-255	Should be used if you are writing a server application that performs a known network service.

3.2.2.1 Using the Object Spawner

If you have chosen an object number from 1 to 255, you must define your object name and number in the object database.
EXAMPLE:

In the following example, the **ncp set object** command defines the Network Management Listener (**nml**) as object number 19.

```
% ncp RET
ncp> set object nml number 19 RET
```

(For more details, see the *DECnet–ULTRIX Network Management* manual.)

If your object number is 0, you can define it in the database or the spawner will use the entry for the **default** object.

NOTE

The **default** DECnet object is no longer shipped with a default user defined for it. Therefore, incoming connections to this object without valid access-control information or proxy will not be accepted. If you want to allow unrestricted access to your system through this object, issue the following **ncp** command:

```
% ncp define object default user guest RET
```

In previous DECnet–ULTRIX releases, the process group ID for processes created by the DECnet spawner was set to 0. Starting with Version 2.2, the process group ID setting is equal to the process ID setting.

3.2.2.2 Using System Calls

You must specify your object name, object number, or both in a **sockaddr_dn** structure. If you are using object number 0, you must also specify an object name. You can then issue a **bind** call on the same socket you use to listen for calls.
EXAMPLE:

In this example, 128 is specified as an object number.

```
int s;
struct sockaddr_dn server;

server.sdn_family = AF_DECnet;
server.sdn_objnum = 128;
if (bind(s,&server, sizeof(struct sockaddr_dn))<0)

   exit();
```

3.2.3 Selecting Accept-Immediate or Accept-Deferred Modes

The server uses accept-immediate or accept-deferred mode to accept incoming connections. (See **accept(2dn)** for details on how to use the **accept** call to establish a network connection.)

NOTE

Accept-immediate mode is the default setting for both the DECnet object spawner and system calls.

3.2.3.1 Using the Object Spawner

Use the **set object** command to choose between accept-immediate and accept-deferred modes for an object entry.
EXAMPLE:

This example shows you how to use the **set object** command to select accept-deferred mode for the server, **myserver**.

```
% ncp RET
ncp> set object myserver accept deferred RET
```

For more information about **ncp** commands, see the *DECnet–ULTRIX NCP Command Reference* manual.

3.2.3.2 Using System Calls

Use the **setsockopt** call to select accept-immediate or accept-deferred mode. Specify the ACC_IMMED option to select accept-immediate mode; specify the ACC_DEFER option to select accept-deferred mode.
EXAMPLE:

In this example, the socket accept mode is set to deferred.

```
char val = ACC_DEFER;

setsockopt(s,DNPROTO_NSP,DSO_ACCEPTMODE,&val,sizeof(val));
```

3.2.4 Verifying Remote User Access to the Server

A server can screen incoming connection requests from the client based on the access-control or proxy information the client supplies.

3.2.4.1 Using the Object Spawner

Regardless of the accept mode chosen for the server, the spawner verifies access-control information or proxy information that the client supplies. The spawner also rejects or processes connection requests based on the following conditions:

- If the information is invalid, the spawner rejects the connection request.
- If the information is valid, the spawner processes the **connect** request according to the accept mode specified for the server in the object database.
- If immediate mode was specified, the spawner accepts the request and initiates the server.
- If deferred mode was specified, the spawner initiates the server. The server must then use the **setsockopt** call with either the DSO_CONACCEPT or DSO_CONREJECT option to either accept or reject the request.

See Section 2.3 for more information about how the object spawner uses access-control or proxy information.

3.2.4.2 Using System Calls

If you are not using the spawner to process requests for the server, you can use the **getsockopt** system call to screen requests. If the server is bound and it is listening on a specific address for its own connection requests, it can verify access to the service based on incoming access-control information or proxy.

EXAMPLE:

In this example, the **getsockopt** call retrieves incoming access-control information.

```
struct accessdata_dn acc_data;

getsockopt(s,DNPROTO_NSP,DSO_CONACCESS,&acc_data,sizeof(acc_data));
```

NOTE

A process has access to data in the password field of the **accessdata_dn** structure only if it is running with superuser privileges. If not, the password field of the **accessdata_dn** structure will be null.

3.2.5 Exchanging Optional Data

In the DECnet domain, client and server can exchange up to 16 bytes of optional data during the connection and disconnection processes. Both server and client interpret this data according to an application-specific design. The following steps describe how the client and server applications exchange optional data:

1. Before the server can read optional connection data, you must ensure that the socket is in accept-deferred mode.
2. If the server is going to accept the connection, use the **setsockopt** call with the DSO_CONDATA option to specify the outgoing optional data.
3. To accept the connection, issue the **setsockopt** call with the DSO_CONACCEPT option.
4. If the server is going to reject the connection, specify outgoing optional data using the **setsockopt** call with the DSO_DISDATA option.

5. To reject the connection, issue the **setsockopt** call with the DSO_CONREJECT option.

EXAMPLE 1:

This example shows how the **optdata_dn** structure specifies optional data before accepting or rejecting a connection.

```
struct optdata_dn optional;
char message[] = { 1, 2, 3, 4, 5 };

bzero(&optional, sizeof(optional));
bcopy(message, optional.opt_data, sizeof(message));
optional.opt_optl = sizeof(message);
```

EXAMPLE 2:

In this example, the server program sends optional data and accepts a connection.

```
setsockopt(sock, DNPROTO_NSP, DSO_CONDATA, &optional,
                sizeof(optional));
setsockopt(sock, DNPROTO_NSP, DSO_CONACCEPT, 0, 0);
```

EXAMPLE 3:

In this example, the server program sends optional data and rejects a connection.

```
setsockopt(sock, DNPROTO_NSP, DSO_DISDATA, &optional,
                sizeof(optional));
setsockopt(sock, DNPROTO_NSP, DSO_CONREJECT, 0, 0);
```

3.3 Transferring Data After Establishing a Connection

After establishing a connection, the client and server applications use their sockets to send and receive data via the **send**, **recv**, **write**, and **read** system calls.

NOTE

If you are using the DECnet object spawner, standard input and standard output are redirected to the network connection.

DECnet–ULTRIX software supports the following services during data transfer between client and server applications:

- Blocking or nonblocking input/output modes
- Out-of-band messages
- Zero-length message detection on sequenced packet sockets

3.3.1 Using Blocking or Nonblocking I/O

Blocking mode is the default mode for ULTRIX software. Unless otherwise specified, an ULTRIX **read** call blocks a calling process until data is available for the **read** operation, and an ULTRIX **write** call blocks a calling process until enough resources are available to buffer data for the **write** operation.

If no data is available when a **read** call is issued, or if there are not enough resources to buffer data for a **write** operation, a nonblocking I/O call returns control to the calling process immediately with an EWOULDBLOCK message. Otherwise, the calling procedure regains control as soon as the **read** or **write** operation completes.

EXAMPLE:

To set up the nonblocking I/O mode, include the ULTRIX **fcntl(2)** system call in the beginning of your application, as follows:

```
fcntl(sock,F_SETFL,FNDELAY);
```

3.3.2 Using Out-of-Band Messages

An application can send an out-of-band message from 1 to 16 bytes long ahead of normal data messages. However, it can send only one out-of-band message over a socket at a time.

The receiving application must read any pending out-of-band message before the sending program can send another. The signal SIGURG indicates the arrival of out-of-band data.

To send or receive an out-of-band message, an application must specify the MSG_OOB flag with the **send** or **recv** call, depending on the following conditions:

- The **send** call specifies the socket used to send an out-of-band message and the buffer used to contain the message.
- The **recv** call specifies the socket used to receive an out-of-band message and the buffer used to contain the message.

EXAMPLE 1:

In this example, the application sends "buffer" as an out-of-band message:

```
char buffer[] = { 1, 2, 3, 4, 5 };
send(sock, buffer, sizeof(buffer), MSG_OOB);
```

You can also use the **select** call to wait for out-of-band data. When the **select** call returns and indicates that an out-of-band message is present, you can use a **recv** call to read the message.

EXAMPLE 2:

In this example, the application uses the **select** call to determine if out-of-band data has arrived.

```
int EMask;
EMask = 1<<sock;
select(sock+1, (int *)0, (int *)0, &EMask, (struct timeval *)0);

if( EMask & 1<<sock )
    msgsize = recv(sock, buffer, sizeof(buffer), MSG_OOB);
```

3.3.3 Detecting Zero-Length Messages

On a sequenced-packet socket, a returned value of zero on a **read** operation indicates that either the end of the file has been reached or a zero-length message has been received. An end-of-file status on a socket indicates that the logical link was disconnected and communication over the socket is not possible.

To distinguish between an end-of-file message and a zero-length packet, use the **dnet_eof** subroutine. If the return value from the **dnet_eof** call is zero, a zero-length packet has been received. Otherwise, the logical link has been disconnected.

EXAMPLE:

This example shows how to use **dnet_eof** to distinguish zero-length messages from end-of-file messages.

```
/* Read the next packet on a DECnet sequenced packet socket */
length = read(sock, buff, buffsize);
if( length == -1 )
/* read failed, refer to read(2dn) for more information */
else if( length == 0  &&  dnet_eof(sock) )
/* End Of File has been reached */

/* If here, then we have successfully read a packet */
```

3.4 How to Disconnect a DECnet Connection

Either the client or the server application can initiate the disconnection of a DECnet connection. The application that initiates the disconnection can also specify the optional disconnect data to send to the other application.

To disconnect a DECnet connection:

1. Before initiating the disconnection, the application can specify between 1 and 16 bytes of optional disconnection data by issuing a **setsockopt** call with a DSO_DISDATA option.
2. The application either closes all references to the DECnet socket or issues a single **shutdown** call to request disabling of the **send** or **send** and **recv** operations. The **shutdown** call is used when the application is set to reference the DECnet socket after the connection is terminated.
3. If an application does not initiate the disconnection, it can retrieve the optional disconnection data sent by the application that initiated the disconnection. The application can issue a **getsockopt** call with the DSO_DISDATA option.

EXAMPLE 1:

In this example, the application terminates the DECnet connection while sending optional disconnection data.

```
    struct optdata_dn disdata;
    char message[] = {1, 2, 3, 4, 5};

    /* Prepare optional disconnect data */
    bzero(&disdata, sizeof(disdata));
    bcopy(message, disdata.opt_data, sizeof(message));
    disdata.opt_optl = sizeof(message);

    setsockopt(sock,DNPROTO_NSP,DSO_DISDATA,&disdata,sizeof(disdata))S ();
    close(sock);
```

EXAMPLE 2:

In this example, the application determines that the DECnet connection has been terminated, and retrieves any optional data that may have been sent by the application that terminated the connection.

```
    struct optdata_dn disdata;
    int length;

    length = read(sock, buff, buffsize);

    /* Check to see if the connection has been disconnected */
    if(length == 0  &&  dnet_eof(sock))
    {

    int structsize;
```

```
 /* Retrieve any optional disconnect data that was sent */
 structsize = sizeof(disdata);
 getsockopt(sock,DNPROTO_NSP,DSO_DISDATA,&disdata,&structsize);

/* No longer need the socket descriptor. Closing it will free
up local network resources that were associated with it */
close(sock);

 }
```

NOTE

The successful completion of a **write** call does not necessarily indicate that the data has already been sent to the remote node. A successful **write** operation means that the local system has accepted the data and will transmit it as soon as possible.

The effect of issuing a **close** call while data that has not been sent is queued for a remote application depends on the value of the LINGER option, which is set through the **setsockopt** call. If the SO_LINGER option is set, the **close** operation will be delayed while an attempt is made to send or acknowledge all data. Otherwise, the data in the queue is eliminated and the system processes the **close** operation with an abort condition.

Part II

Reference

Chapter 4

DECnet–ULTRIX System Calls

This reference chapter describes DECnet–ULTRIX system calls in detail. The format for this information corresponds to that in the ULTRIX reference pages. See the *ULTRIX Reference* manuals for more information about format.

Each system call begins a separate page in alphabetical order. The name of the system call appears in a running head followed by the appropriate section number and a suffix. For example, **accept(2dn)** appears on the reference pages describing the **accept** call. The **2** indicates that the section describes system calls. The **dn** indicates that the system call is used in the DECnet domain.

4.1 System Call Summary

Table 4-1 summarizes the function of each DECnet–ULTRIX system call.

Table 4–1: DECnet–ULTRIX System Calls

System Call	Function
accept	Accepts a connection request.
bind	Binds a name to a socket.
close	Terminates a logical link and deactivates a socket descriptor.
connect	Initiates a connection request.
getpeername	Returns the name of the peer connected to a socket.
getsockname	Returns the current name of a socket.
getsockopt	Returns the options associated with a socket.
listen	Listens for pending connection requests.
read	Reads (receives) data.
recv	Receives normal data and out-of-band messages.
select	Performs synchronous I/O multiplexing.
send	Sends data and out-of-band messages.
setsockopt	Sets socket options.
shutdown	Shuts down a DECnet connection.
socket	Creates a new socket.
write	Writes (sends) data.

4.2 On-Line Manual Pages

The system call descriptions also appear as on-line documentation in **accept(2dn)**, **bind(2dn)**, **close(2dn)**, and so on.

4.3 Format and Conventions

The descriptions of the DECnet–ULTRIX system calls have the following format:

SYNTAX

Gives the complete syntax for the system call. The following conventions apply to syntax lines:

`command`	Indicates terms that are constant and must be typed exactly as presented.
...	Indicates that the preceding item can be repeated one or more times.
italics	Indicate a variable, for which either you or the system must specify a value.
%	The default user prompt in multiuser mode.
#	The default superuser prompt.

DESCRIPTION

Supplies function and background information.

RETURN VALUE

Explains the meaning of a value returned by a utility when it completes or does not complete an operation.

DIAGNOSTICS

Lists diagnostic messages that can be returned.

RESTRICTIONS

Describes restrictions that apply to the use of the system call or subroutine.

SEE ALSO

Provides cross-references to associated information in this manual and in other DECnet–ULTRIX and ULTRIX manuals.

In text, cross-references to specific manual reference pages include the section number in the ULTRIX or DECnet-ULTRIX reference manual where the commands are documented. For example, **socket(2dn)** refers to the description of the **socket** system call in Section 2dn of the ULTRIX reference pages.

4.4 System Call Descriptions

The following pages describe each system call in detail.

accept (2dn)

NAME

accept — accept a connection request

SYNTAX

```
#include <sys/types.h>\bold
#include <sys/socket.h>\bold
#include <netdnet/dn.h>\bold
```

```
ns = accept (s,addr,addrlen)
int ns,s;
struct  sockaddr_dn *addr;
int *addrlen;
```

where

Input Arguments:

s	specifies a descriptor for a socket that has been returned by the **socket** call, bound to a name by the **bind** call, and is listening for connection requests after issuing a **listen** call.
addr	is the address of a **sockaddr_dn** structure. This address identifies the source entity that is requesting the connection.
addrlen	specifies the size of the source address.

Return Arguments:

ns — is the new descriptor for the accepted socket.

addr — specifies the address of a **sockaddr_dn** structure. This call fills in the following data fields:

sdn_family	specifies the communications domain as AF_DECnet.
sdn_objnum	specifies the source DECnet object number.
sdn_objnamel	is the size of the source node's object name. This argument is used only when the DECnet object number, *sdn_objnum*, is 0.
sdn_objname	defines the name of the network program, which can be up to 16 characters. This argument is used only when the DECnet object number, *sdn_objnum*, is 0.
sdn_nodeaddrl	is the size of the node address for the source program.
sdn_nodeaddr	specifies the node address for the source program.

addrlen — specifies the actual length (in bytes) of the returned address.

DESCRIPTION

The **accept** call extracts the first connection request on the queue of pending connections, creates a new socket having the same properties as *s*, and allocates a new file descriptor, *ns*, for the socket. *s* remains open and listens for connection requests.

There are two modes of accepting an incoming connection: immediate and deferred. You can specify the mode of acceptance with the **setsockopt** call.

Accept-immediate mode, specified as ACC_IMMED, causes both client and server programs to complete the protocol exchange at the Network Services Protocol (NSP) level. The server program ignores any access-control information or optional data that the source program may have sent. ACC_IMMED is the default.

Deferred mode, specified as ACC_DEFER, causes the **accept** call to be completed by a server program without a full protocol exchange between itself and the client program. Deferred mode allows a server program to examine and process any access-control information or optional data before notifying the client program of the acceptance or rejection of its connect request.

RETURN VALUE

If the **accept** call succeeds, it returns a nonnegative integer, which is a descriptor for the accepted socket. If an error occurs, the call returns a value of -1 and the external variable **errno** will contain the type of error.)

DIAGNOSTICS

The call succeeds unless:

[EBADF]	The *s* argument is not a valid descriptor.
[ECONNABORTED]	DECnet is shutting down on the local node.
[EFAULT]	The *addr* argument is not in a write-enable part of the user address space.
[EWOULDBLOCK]	The socket is marked for nonblocking, and no connections are waiting to be accepted.

SEE ALSO

bind(2dn), listen(2dn), setsockopt(2dn), socket(2dn)

bind (2dn)

NAME

bind — bind a name to a socket

SYNTAX

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdnet/dn.h>

bind (s,name,namelen)

int s;
struct  sockaddr_dn name;
int namelen;
```

where

s	specifies a descriptor for a socket that has been returned by the **socket** call.
name	specifies the address of a **sockaddr_dn** structure.
namelen	specifies the size of the address of the **sockaddr_dn** structure. This call fills in the following data fields:

sdn_family	specifies the communications domain as AF_DECnet.
sdn_objnum	specifies the DECnet object number to which you are binding. If the object number is 0, the DECnet object name, *sdn_objname*, is used.
sdn_objnamel	specifies the size of the object name to which you are binding. This argument is used only when the DECnet object number, *sdn_objnum*, is 0.
sdn_objname	specifies the name of the source network program. This argument is used only when the DECnet object number, *sdn_objnum*, is 0.

DESCRIPTION

The **bind** call assigns a name to a socket. When a socket is created with the **socket** call, it exists in a name space (address family) but has no assigned name. The **bind** call requests that a specific name be assigned to the socket.

NOTE

The DECnet object numbers 1 through 127 are reserved by Digital for sockets that are in programs running as superuser.

RETURN VALUE

If the **bind** call is successful, a value of 0 is returned. If the call is unsuccessful, a value of -1 is returned and the external variable **errno** contains error detail.

DIAGNOSTICS

The call succeeds unless:

[EACCESS]	The requested address is reserved, and the current user does not have superuser privileges.
[EADDRINUSE]	The specified name is already bound to a listening socket on the local machine.
[EAFNOSUPPORT]	The *sdn_family* is not AF_DECnet.
[EBADF]	The *s* argument is not a valid descriptor.
[EFAULT]	The *name* argument is not located in a valid part of the user's address space.
[EINVAL]	The *namelen* argument is not the size of *sockaddr_dn*, or *sdn_objnamel* is not in the range of 0 to 16.

RESTRICTION

Bound names are ignored for sockets set up to initiate connections in all programs except programs running as superuser.

SEE ALSO

socket (2dn)

close (2dn)

NAME

close — terminate a DECnet connection

SYNTAX

```
close (s)
int s;
```

where

s	specifies a descriptor for a socket that has been returned by the **socket** or the **accept** call.

DESCRIPTION

The **close** call terminates an outstanding connection over a DECnet socket descriptor and deactivates the descriptor. When the last **close** call is issued on that descriptor, all associated naming information and queued data are discarded. (The **close** call deletes a descriptor from the per-process object reference table. If this is the last reference to the underlying socket, the socket is deactivated.)

The effect of issuing a **close** call while unsent data is queued for a remote program depends on the value of the linger option set with the **setsockopt** call. If SO_LINGER is set, the **close** operation is delayed while an attempt is made to send or acknowledge all data; otherwise, the data in the queue is eliminated and the system processes the **close** with an abort.

A **close** of all of a program's descriptors is automatic when an **exit** call is issued, but because there is a limit on the number of active descriptors per program, the **close** call is necessary for programs that deal with many descriptors.

RETURN VALUE

If the call succeeds, a value of 0 is returned. If an error occurs, a value of -1 is returned. Additional error detail is specified in the external variable **errno**.

DIAGNOSTICS

The call succeeds unless:

[EBADF]	The *s* argument is not a valid descriptor.

SEE ALSO

accept(2dn), setsockopt(2dn), socket(2dn)

connect (2dn)

NAME

connect — initiate a connection request

SYNTAX

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdnet/dn.h>

connect (s,name,namelen)
int s;
struct  sockaddr_dn * name;
int namelen;
```

where

s	specifies a descriptor for a socket that has been returned by the **socket** call.
name	specifies the address of a **sockaddr_dn** structure. This address identifies the destination process for the **connect**.
namelen	specifies the size of the address of the **sockaddr_dn** structure. This call fills in the following data fields:

sdn_family	specifies the communications domain as AF_ DECnet.
sdn_flags	specifies the object flag option, which you can use to request outgoing proxy.
sdn_objnum	specifies the DECnet object number to which you are connecting. If the object number is 0, the DECnet object name, *sdn_objname*, is used.
sdn_objnamel	specifies the size of the object name to which you are connecting. This argument is used only when the DECnet object number, *sdn_objnum*, is 0.
sdn_objname	specifies the name of the destination program. This argument is used only when the DECnet object number, *sdn_objnum*, is 0.

DESCRIPTION

The **connect** call issues a connect request to a server program. The server program is specified by the *name* argument, which is an address in the DECnet domain.

Optional user data and access-control information are passed with the **connect** call if this data is previously set with the **setsockopt** call.

RETURN VALUE

If the **connect** succeeds, a value of 0 is returned. If the **connect** fails, a value of -1 is returned and the external variable **errno** contains error detail.)

DIAGNOSTICS

The call succeeds unless:

[EACCESS]	The access-control information was rejected.
[EADDRNOTAVAIL]	No such node.
[EAFNOSUPPORT]	Addresses in the specified address family cannot be used with this particular socket.
[EBADF]	The *s* argument is not a valid descriptor.
[ECONNREFUSED]	The attempt to connect was refused by the remote object.
[EFAULT]	The *name* argument specifies an area outside the process address space.
[EHOSTDOWN]	Local node is shut down.
[EHOSTUNREACH]	Node unreachable.
[EINVAL]	The *namelen* argument is not the size of **sockaddr_ dn**, or *sdn_objnamel* is not in the range of 0 to 16.
[EISCONN]	The socket is already connected.
[ENETDOWN]	The remote node is shutting down.
[ENOSPC]	There are no resources available for a new connection at either the local or remote system.
[ESRCH]	Unrecognized object at the remote node.
[ETIMEDOUT]	Connection establishment was timed out before a connection was established.
[ETOOMANYREFS]	The remote object has too many active connections.
[INPROGRESS]	The socket is nonblocking and the connection is in progress. To determine when the connection has either completed or failed, select the socket for **write** using the **select** call.

SEE ALSO

accept(2dn), close(2dn), setsockopt(2dn), socket(2dn)

dnet_conn(3dn)

getpeername (2dn)

NAME

getpeername — get name of connected peer

SYNTAX

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdnet/dn.h>

getpeername (s,name,namelen)
int s;
struct  sockaddr_dn *name;
int *namelen;
```

where

Input Arguments:

s	specifies a descriptor for a socket that has been returned by the **socket** or the **accept** call.
name	specifies the address of a **sockaddr_dn** structure.
namelen	specifies the length of the address of the **sockaddr_dn** structure.

Return Arguments:

name specifies the address of a *sockaddr_dn* structure. This call fills in the following data fields:

sdn_family	specifies the communications domain as AF_DECnet.
sdn_objnum	specifies the DECnet object number for the peer program. If the object number is 0, the DECnet object name, *sdn_objname*, is used.
sdn_objnamel	is the length of the peer object name. This argument is used only when the object number, *sdn_objnum*, is 0.
sdn_objname	specifies the name of the peer network program, which can be up to a 16-element array of characters. This argument is used only when the object number, *sdn_objnum*, is 0.
sdn_nodeaddrl	is the size of the remote node address.
sdn_nodeaddr	specifies the remote node address.

namelen returns the length of the address of a **sockaddr_dn** structure.

DESCRIPTION

The **getpeername** call returns the name of the peer DECnet program connected to a specified socket.

RETURN VALUE

If the call succeeds, a value of 0 is returned. If an error occurs, a value of -1 is returned. When an error condition exists, the external variable **errno** contains error detail.

DIAGNOSTICS

The call succeeds unless:

[EBADF]	The *s* argument is not a valid descriptor.
[EFAULT]	The *name* argument points to memory located in an invalid part of the process address space.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[ENOTCONN]	The socket *s* is not connected.

getsockname (2dn)

NAME

getsockname — return the current name for a socket

SYNTAX

```
getsockname (s,name,namelen)
int s;
struct  sockaddr_dn *name;
int *namelen;
```

where

Input Arguments:

s	specifies a descriptor for a socket that has been returned by the **socket** or the **accept** call.
name	specifies the address of a **sockaddr_dn** structure. This address is the name that was bound to the socket.
namelen	specifies the length of the address of the **sockaddr_dn** structure.

Return Arguments:

name specifies the address of a **sockaddr_dn** structure. This call fills in the following data fields:

sdn_family	specifies the communications domain as AF_ DECnet.
sdn_objnum	specifies the DECnet object number for the socket. If the object number is 0, the DECnet object name, *sdn_objname*, is used.
sdn_objnamel	is the size of the object name. This argument is used only when the object number, *sdn_objnum*, is 0.
sdn_objname	specifies the DECnet object name, which can be up to a 16-element array of characters. This argument is used only when the object number, *sdn_objnum*, is 0.
sdn_nodeaddrl	is the size of the local node address.
sdn_nodeaddr	specifies the local node address.

namelen returns the length of the address of the **sockaddr_dn** structure.

DESCRIPTION

The **getsockname** call returns the current name of a specified socket.

RETURN VALUE

If the call succeeds, a value of 0 is returned. If an error occurs, a value of -1 is returned. When an error condition exists, the external variable **errno** contains error detail.

DIAGNOSTICS

The call succeeds unless:

[EBADF]	The *s* argument is not a valid descriptor.
[EFAULT]	The *name* argument points to memory located in an invalid part of the process address space.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.

SEE ALSO

bind(2dn)

getsockopt (2dn) and setsockopt (2dn)

NAME

getsockopt, setsockopt — get and set options on sockets

SYNTAX

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdnet/dn.h>

setsockopt (s,level,optname,optval,optlen)
int s,level,optname;
char *optval;
int *optlen;

getsockopt (s,level,optname,optval,optlen)
int s,level,optname;
char *optval;
int *optlen;
```

where

Input Arguments:

s	specifies a descriptor for a socket in the AF_DECnet domain.
level	specifies the level at which options are interpreted: The **socket** call level SOL_SOCKET causes options to be interpreted at the socket level. A level of DNPROTO_NSP instructs DECnet to interpret an option.
optname	specifies an option to be interpreted at the level specified.
optval, optlen	specify option values that are used with the **getsockopt** and **setsockopt** calls. The interpretations of these arguments relative to each call are as follows:

getsockopt:

optval	specifies the buffer that will contain the returned value for the requested option.
optlen	is the value result parameter that initially contains the size of the buffer pointed to by *optval* and is modified on return to indicate the actual length of the returned value.

setsockopt:

optval	specifies the address for the buffer that contains information for setting option values.
optlen	specifies the length of the option value buffer.

DESCRIPTION

The **getsockopt** and **setsockopt** calls manipulate various options associated with a socket. Options may exist at multiple levels, so you must specify the level for the desired operation. The socket level SOL_SOCKET options are as follows:

SO_DEBUG	enables the recording of debugging information.
SO_LINGER	controls the actions taken when unsent messages are queued on a socket and a **close** call is issued. If SO_LINGER is set, the system will block the process until all data has been received by the remote system or until it is unable to deliver the information.

At the DECnet level, socket options can define the way in which a pending connection is accepted. Options at this level also control the sending and receiving of optional user data and access-control information, and return information on current link status. The DECnet options follow:

DSO_ACCEPTMODE

The accept mode option is used at the DECnet level for processing **accept** calls. The program must issue a **bind** call on the socket before this option is valid. A **setsockopt** call to set the accept mode is valid only if issued before a **listen** call is performed.

There are two values that can be supplied for this option: ACC_IMMED (immediate mode) and ACC_DEFER (deferred mode). (The optional value (*optval*) for this option is the *char* type.)

ACC_IMMED	The default mode for this option. When immediate mode is in effect, control is immediately returned to a server program following an **accept** call with the connection request accepted.
ACC_DEFER	Enables a server program to complete an **accept** call without fully completing the connection to the client program. The server program can then examine the access-control information and/or optional data before accepting or rejecting the pending connection. The server program can then issue the **setsockopt** call with the appropriate reject or accept option.

DSO_CONACCEPT	Allows the server program to accept the connection on socket returned by the **accept** call and previously set to ACC_DEFER mode. Any optional data previously set by DSO_CONDATA will be sent. (There is no optional value (*optval*) for this option.)
DSO_CONREJECT	Lets the server program reject a pending connection on a socket returned by the **accept** call and previously set to ACC_DEFER mode. Any optional data previously set by DSO_DISDATA will be sent. The reject reason is passed with this option as a *short int* value.

DSO_CONDATA	This option allows up to 16 bytes of optional user data to be sent by the **setsockopt** call. The data is sent as a result of the **connect** or the **accept** (with the deferred option) call. The optional data is passed in an **optdata_dn** structure. (See the **optdata_dn** data structure in Appendix A for formatting information.) The data is read by the task issuing the **getsockopt** call with this option.
DSO_DISDATA	Allows up to 16 bytes of optional data associated with the **setsockopt** call. It is sent as a result of the **close** call. The optional data is passed in a **optdata_dn** structure. (See the **optdata_dn** data structure in Appendix A for formatting information.) The data is read by the program issuing the **getsockopt** call with this option.
DSO_CONACCESS	Allows access-control information to be set by the client program and received by the server program. The data is sent with the **setsockopt** call and passed to the server when an ensuing **connect** call is issued. The server retrieves the information by issuing a **getsockopt** call. The access data is sent to the server program. It is passed with the **connect** call in an **accessdata_dn** data structure. (See the **accessdata_dn** data structure in Appendix A for formatting information.) The access data is read by the program issuing the **getsockopt** call with this option.

NOTE

The DSO_CONACCESS socket option for the **getsockopt** call will function only in programs running as root.

DSO_LINKINFO

Gets information on the state of a DECnet logical link. Link state information is passed in a *linkinfo_dn* structure in the *idn_linkstate* field. The following are possible link states and their respective values:

`LL_INACTIVE`	`logical link inactive`
`LL_CONNECTING`	`logical link connecting`
`LL_RUNNING`	`logical link running`
`LL_DISCONNECTING`	`logical link disconnecting`

See Appendix A for information on formatting for the **linkinfo_ dn** data structure.

RETURN VALUE

If the call completes successfully, a value of 0 is returned. If the call fails, a value of -1 is returned and the external variable **errno** contains error details.

DIAGNOSTICS

The call succeeds unless:

[EBADF]	The *s* argument is not a valid descriptor.
[EBUSY]	The pending connection has gone away.
[EDOM]	The acceptance mode is not valid.
[EFAULT]	The options are not located in a valid part of the process address space.
[EMSGSIZE]	The size of the option buffer is incorrect.
[ENOBUFS]	No buffer space is available to return access-control data.
[ENOPROTOOPT]	No access-control information was supplied with the connection request.
[EOPNOTSUPP]	The option is unknown.

listen (2dn)

NAME

listen — listen for pending connect requests

SYNTAX

```
listen (s,backlog)
int s,backlog;
```

where

s	specifies a descriptor for a socket that has been returned by the **socket** call and bound to a name by the **bind** call.
backlog	defines the maximum length for the queue of pending connection requests for a particular socket. If a connection request arrives when the queue is full, the connection will be rejected.

DESCRIPTION

The **listen** call declares a socket as being available to receive connection requests and listens for incoming connections. The **listen** call must be issued before the server program accepts an incoming connection request.

RETURN VALUE

If the call succeeds, a value of 0 is returned. If an error occurs, a value of -1 is returned. When an error condition exists, the external variable **errno** contains error details.

DIAGNOSTICS

The call succeeds unless:

[EADDRINUSE]	The name bound to the socket is already being used for a listen socket.
[EBADF]	The *s* argument is not a valid descriptor.

SEE ALSO

accept(2dn), connect(2dn), socket(2dn)

read (2dn)

NAME

read — read or receive data

SYNTAX

```
#include <sys/types.h>
#include <sys/sockets.h>

cc = read (s,buf,buflen)
int cc,s;
char *buf;
int buflen;
```

where

Input Arguments:

s	specifies a descriptor for a socket that has been returned by the **socket** call.
buf	specifies the address of buffer into which data is read.
buflen	specifies the size of the message buffer.

Return Arguments:

cc	is the length of the returned message.

DESCRIPTION

The **read** call is used to read normal data messages from another DECnet program. You can use **read** only on a connected socket. If no messages are available at the socket, the **read** call waits for a message to arrive. However, if the socket is nonblocking, a status of -1 is returned with the external variable **errno** set to EWOULDBLOCK.

You can use the **select** call to determine when more data will arrive.

The length of the message is returned in *cc*. If a message is too long to fit in the supplied buffer, the excess bytes can be discarded, depending on the type of socket from which the message is received. Sequenced-packet sockets discard extra bytes. Stream sockets store extra bytes in the kernel and use them for the next **read** call.

RETURN VALUE

If the call succeeds, the number of bytes actually read and placed in the buffer are returned. The system reads the number of bytes requested only if the descriptor references a file containing that many bytes before the end of file. If the end of file has been reached, a value of 0 is returned.

NOTE

A returned value of 0 can also indicate that a zero-length message has been received on a sequenced-packet socket. See **dnet_eof(3dn)** for more information.

If an error occurs, a value of -1 is returned and the global variable **errno** is set to indicate the error.

DIAGNOSTICS

The call succeeds unless:

[EBADF]	The *s* argument is not a valid file descriptor open for reading.
[EFAULT]	The *buf* argument points outside the allocated address space.
[EINTR]	A **read** operation from a slow device was interrupted by the delivery of a signal before any data arrived.
[EWOULDBLOCK]	The socket is marked nonblocking and the **read** operation would have blocked.

SEE ALSO

connect(2dn), socket(2dn)

dnet_eof(3dn)

dup(2), socketpair(2)

recv (2dn)

NAME

recv — receive normal data and out-of-band messages

SYNTAX

```
#include <sys/types.h>
#include <sys/socket.h>

cc = recv (s,buf,buflen,flags)
int cc,s;
char *buf;
int buflen,flags;
```

where

Input Arguments:

s	specifies a descriptor for a socket that has been returned by the **socket** call.
buf	specifies the address of the buffer that will contain the received message.
buflen	specifies the size of the message buffer.
flags	are set to MSG_PEEK, which looks at incoming messages, or to MSG_OOB, which indicates that a program will receive out-of-band messages.

Return Arguments:

cc	is the length of the returned message.

DESCRIPTION

The **recv** call is used to receive normal or out-of-band data from another DECnet program. **recv** can be used only on a connected socket (see **connect(2dn)**). If no messages are available at a socket, the **recv** call waits for a message to arrive. However, if the socket is nonblocking, a status of -1 is returned with the external variable **errno** set to EWOULDBLOCK.

Use the **recv** call instead of the **read** call when you want to specify the MSG_PEEK and MSG_OOB *flags* arguments to look at incoming messages and to receive out-of-band messages.

You can use the **select** call to determine when more data will arrive.

The length of the message is returned in *cc*. If a message is too long to fit in the supplied buffer, the excess bytes may be discarded depending on the type of socket from which the message is received. Sequenced-packet sockets discard extra bytes. Stream sockets store extra bytes in the kernel and use them for the next **recv** call.

The *flags* argument for a **recv** call is formed by **oring** one or more of the following values:

```
#define    MSG_PEEK  0x1  /* peek at incoming message */
#define    MSG_OOB  0x2  /* process out-of-band data */
```

Out-of-band messages are sent to a receiving program ahead of normal data messages. Out-of-band messages are sent and received as DECnet interrupt messages and can be from 1 to 16 bytes in length. The signal SIGURG indicates the arrival of out-of-band data. You can also use the **select** call to determine if out-of-band data has arrived by using the *exceptfds* argument.

RETURN VALUE

If the call succeeds, the number of received characters is returned. If an error occurs, a value of -1 is returned. Additional error detail will be specified in the external variable **errno**.

DIAGNOSTICS

The call succeeds unless:

[EBADF]	The *s* argument is not a valid descriptor.
[EFAULT]	The data was specified to be received into a nonexistent or protected part of the process address space.
[EWOULDBLOCK]	The socket is marked nonblocking and the **receive** operation would have blocked.

RESTRICTION

The MSG_PEEK *flags* argument cannot be used with out-of-band messages.

SEE ALSO

read(2dn), send(2dn), socket(2dn), write(2dn)

select (2dn)

NAME

select — synchronous I/O multiplexing

SYNTAX

```
#include <sys/time.h>

nfound = select (nfds,readfds,writefds,exceptfds,timeout)
int nfound,nfds, *readfds, *writefds, *exceptfds;
struct timeval *timeout;
```

where

Input Arguments:

nfds	specifies the number of descriptors to be checked. For example, the bits from 0 to *nfds*-#1 in the masks are examined.
readfds	specifies the descriptor to be examined for read (or receive) data ready. This descriptor can be given as a null pointer (0) if it is of no interest.
writefds	specifies the descriptor to be examined for write (or send) data ready. This descriptor can be passed as a null pointer (0) if it is of no interest.
exceptfds	specifies the descriptor to be examined for out-of-band data ready. This descriptor can be given as a null pointer (0) if it is of no interest.
timeout	specifies an address of a **timeval** structure. If *timeout* is a nonzero pointer, it specifies a maximum interval to wait for the selection to complete. If *timeout* is a zero pointer, the select blocks indefinitely. To affect a poll, *timeout* should be nonzero, pointing to a zero-valued *timeval* structure.

Return Argument:

nfound	is the total number of ready descriptors returned.

DESCRIPTION

The **select** call examines the I/O descriptors specified by the bit masks *readfds*, *writefds*, and *exceptfds* to determine whether the descriptors are ready for reading or writing or have an out-of-band condition pending, respectively. File descriptor *f* is represented by the bit 1<<f in the mask. The *nfds* descriptors are checked; that is, the bits from 0 through *nfds* -#1 in the masks are examined. The **select** call returns a mask of those descriptors that are ready. The total number of ready descriptors is returned in *nfound*.

If *timeout* is a nonzero pointer, it specifies a maximum interval to wait for the selection to complete. If *timeout* is a zero pointer, the **select** call blocks indefinitely. To affect a poll, the *timeout* argument should be nonzero and pointing to a zero-valued **timeval** structure.

The *readfds*, *writefds*, and *exceptfds* arguments can be defined as 0 if no descriptors are of interest.

RETURN VALUE

The **select** call returns the number of descriptors that are contained in the bit masks. If an error occurs, a value of -1 is returned. Additional error details will be contained in the external variable **errno**. If the time limit expires, a value of 0 is returned.

DIAGNOSTICS

The call succeeds unless:

[EBADF]	One of the bit masks is specified as an invalid descriptor.
[EINTR]	An asynchronous signal was delivered before any of the selected events occurred or the time limit expired.

RESTRICTION

The descriptor masks are always modified on return, even if the call returns as the result of the timeout.

SEE ALSO

accept(2dn), connect(2dn), read(2dn), recv(2dn), send(2dn), write(2dn)

send (2dn)

NAME

send — send normal data and out-of-band messages

SYNTAX

```
#include <sys/types.h>
#include <sys/socket.h>

cc = send (s,msg,msglen,flags)
int cc,s;
char *msg;
int msglen,flags;
```

where

Input Arguments:

s	specifies a descriptor for a socket that has been returned by the **socket** call.
msg	specifies the address of the buffer that contains the outgoing message.
msglen	specifies the size of the message.
flags	are set to MSG_OOB, which sends an out-of-band message.

Return Argument:

cc	is the number of characters sent.

DESCRIPTION

The **send** call transmits normal or out-of-band data to another program. It can be used only when a socket is in a connected state. See **connect(2dn)** for more information.

Use the **send** call instead of **write** when you want to specify the MSG_OOB *flags* argument to indicate that out-of-band data will be sent to the destination program. Out-of-band messages are sent to a receiving program ahead of normal data messages. Out-of-band messages are sent and received as DECnet NSP interrupt messages and can be from 1 to 16 bytes long.

The number of characters sent is returned in *cc*. If no message space is available at the receiving socket to hold the message being transmitted, the **send** call will, in most cases, block. If the socket is set in nonblocking I/O mode, **send** returns an error with **errno** set to EWOULDBLOCK.

You can use the **select** call to determine when it is possible to send more data.

RETURN VALUE

If the call succeeds, the number of characters sent are returned. If an error occurs, a value of -1 is returned. Additional error detail will be specified in the external variable **errno**.

DIAGNOSTICS

The call succeeds unless:

[EBADF]	The *s* argument is not a valid descriptor.
[EFAULT]	An invalid user address space was specified for an argument.
[EMSGSIZE]	The socket requires that the message be sent atomically, but the size of the message made this impossible. Note that zero-length messages are illegal.
[EWOULDBLOCK]	The socket is marked nonblocking and the **send** operation would have blocked.

SEE ALSO

read(2dn), recv(2dn), write(2dn)

setsockopt (2dn)

NAME

See **getsockopt(2dn)**

shutdown (2dn)

NAME

shutdown — shut down a logical link

SYNTAX

```
shutdown (s,how)
int s,how;
```

where

s	is a descriptor for the socket associated with the DECnet logical link that you want to shut down.
how	is an integer specifying how the connection is shut down. If the value of *how* is 0, further receives are disabled. If the value of *how* is 1, further sends are disabled. If the value of *how* is 2, further sends and receives are disabled.

DESCRIPTION

The **shutdown** call shuts down all or part of a DECnet logical link connection on the socket specified by the argument *s*.

RETURN VALUE

If the call succeeds, a value of 0 is returned. If the call fails, a value of -1 is returned.

DIAGNOSTICS

<he call succeeds unless:

[EBADF]	The *s* argument is not a valid descriptor.
[ENOTCONN]	The specified socket is not connected.
[ENOTSOCK]	The *s* argument is a file, not a socket.

SEE ALSO

connect(2dn), socket(2dn)

socket (2dn)

NAME

socket — create a socket and return a descriptor

SYNTAX

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdnet/dn.h>
```

```
s = socket (af,type,protocol)
int s,af,type,protocol;
```

where

Input Arguments:

af	specifies the address format for the DECnet communication domain as AF_DECnet.
type	specifies the socket type. The DECnet domain supports the following socket types:

- SOCK_STREAM. Stream sockets provide bidirectional, reliable, sequenced, and unduplicated byte streams.
- SOCK_SEQPACKET. Sequenced-packet sockets provide bidirectional, reliable, sequenced data flow while preserving record boundaries in data.

protocol	specifies the protocol to be used with the socket. Valid protocols are 0 (default) and DNPROTO_NSP (DECnet protocol). If you specify the socket type SOCK_SEQPACKET, you must set the protocol to zero.

Return Argument:

s	is the value for the socket descriptor.

DESCRIPTION

The **socket** call creates a socket and returns a socket descriptor.

A socket is an addressable endpoint of communication. A program uses the socket to transmit and receive data to and from a similar socket in another program. Subsequent calls on a particular socket reference that socket's descriptor.

RETURN VALUE

If the call completes successfully, a socket descriptor value is returned. This descriptor is used for subsequent system calls on this particular socket. If an error occurs, a value of -1 is returned. Additional error detail is contained in the external variable **errno**.

DIAGNOSTICS

The call succeeds unless:

[EAFNOSUPPORT]	The specified address family is not supported in this version of the system.
[EMFILE]	Too many open files.
[ENFILE]	The per-process descriptor table is full.
[ENOBUFS]	No buffer space is available. The socket cannot be created.
[EPROTONOSUPPORT]	The specified protocol is not supported.
[ESOCKTNOSUPPORT]	The specified socket type is not supported in this address family.

SEE ALSO

dnet_conn(3dn)

write (2dn)

NAME

write — write or send data

SYNTAX

```
#include <sys/types.h>
#include <sys/socket.h>

cc = write (s,msg,msglen)
int cc,s;
char *msg;
int msglen;
```

where

Input Arguments:

s	specifies a descriptor for a socket that has been returned by the **socket** call.
msg	specifies the address of the buffer that contains the outgoing message.
msglen	specifies the size of the message.

Return Argument:

cc	is the number of bytes sent.

DESCRIPTION

The **write** call is used to transmit normal data messages to another program. You can use **write** only when a socket is in a connected state. See **connect(2dn)** for more information.

The number of bytes sent is returned in *cc*. If no message space is available at the receiving socket to hold the message being transmitted, the **write** call will, in most cases, block. If the socket is set in nonblocking I/O mode, the **write** returns in error with **errno** set to EWOULDBLOCK.

You can use the **select** call to determine when it is possible to send more data.

RETURN VALUE

If the call succeeds, the number of bytes actually written is returned. If an error occurs, a value of -1 is returned and **errno** is set to indicate the error.

DIAGNOSTICS

The call succeeds unless:

[EBADF]	The *s* argument is not a valid descriptor open for writing.
[EFAULT]	Part of *s* or data to be written to the file points outside the process's allocated address space.
[EMSGSIZE]	An attempt is made to transmit a zero-length message or a message that is larger than the DECnet pipeline quota on a sequenced-packet socket.
[EPIPE]	An attempt is made to write to a socket type SOCK_STREAM, which is not connected to a peer socket.
[EWOULDBLOCK]	The socket is marked nonblocking and the **write** operation would have blocked.

SEE ALSO

connect(2dn), read(2dn), recv(2dn), send(2dn)

Chapter 5

DECnet–ULTRIX Subroutines

This reference chapter describes the DECnet–ULTRIX subroutines, which you can find in the C library, **/lib/libdnet.a**. The format for this information corresponds to that in the ULTRIX reference pages. See the *ULTRIX Reference* manuals for more information about format.

Each subroutine begins a separate page in alphabetical order. The name of the subroutine appears in a running head followed by the appropriate section number and a suffix. For example, **dnet_addr(3dn)** appears on the pages describing the **dnet_addr** subroutine. The 3 indicates that the section describes subroutines. The **dn** indicates that the subroutine is used in the DECnet domain.

5.1 Subroutine Summary

Table 5-1 summarizes the function of each DECnet–ULTRIX subroutine.

Table 5–1: DECnet–ULTRIX Subroutines

Subroutine	Function
dnet_addr	Converts an ASCII node address to binary.
dnet_conn	Connects to a specified network object on a remote node and sends any access-control information or optional user data.
dnet_eof	Tests the current state of a connection to determine whether the end-of-file has been reached.
dnet_getalias	Gets extended node information.
dnet_htoa	Returns a DECnet ASCII node name that corresponds to a 16-bit binary node address contained in a structure of the type **dn_naddr**. If a node name is not found for the node address, **dnet_htoa** returns an ASCII string representation of the node address.
dnet_ntoa	Converts a 16-bit binary node address, which is contained in a structure of the type **dn_naddr**, to its ASCII string representation.
dnet_otoa	Converts a DECnet object name or number to its ASCII string representation.

(continued on next page)

Table 5–1 (Cont.): DECnet–ULTRIX Subroutines

Subroutine	Function
getnodeadd	Returns a pointer to a structure of the type dn_naddr, which contains your local DECnet–ULTRIX node address.
getnodeent	Accesses the network node database and returns node information.
getnodename	Returns an ASCII string representation of your local DECnet–ULTRIX node name.
nerror	Produces DECnet error messages.

5.2 On-Line Manual Pages

The subroutine descriptions appear also as on-line documentation; for example, dnet_addr(3dn), dnet_conn(3dn), dnet_eof(3dn), and so on.

See the *DECnet–ULTRIX Use* manual for instructions on how to use on-line manual pages.

5.3 Format and Conventions

The descriptions of the DECnet–ULTRIX system calls have the following format:

SYNTAX

Gives the complete syntax for the subroutine. Syntax lines use the graphic conventions described at the end of this chapter. The following conventions apply to syntax lines:

`command`	Indicates terms that are constant and must be typed exactly as presented.
...	Indicates that the preceding item can be repeated one or more times.
italics	Indicate a variable, for which either you or the system must specify a value.
%	The default user prompt in multiuser mode.
#	The default superuser prompt.

DESCRIPTION

Supplies function and background information.

RETURN VALUE

Explains the meaning of a value returned by a subroutine when it completes or does not complete an operation.

DIAGNOSTICS

Lists diagnostic messages that can be returned.

RESTRICTIONS

Describes restrictions that apply to the use of the subroutine.

SEE ALSO

Provides cross-references to associated information in this manual and in other DECnet–ULTRIX and ULTRIX manuals.

In text, cross-references to specific manual reference pages include the section number in the ULTRIX or DECnet-ULTRIX reference manual where the commands are documented. For example, **dnet_conn(3dn)** refers to the description of the **dnet_conn** subroutine in Section 3dn of the ULTRIX reference pages.

5.4 Subroutine Descriptions

The following pages describe each subroutine in detail.

dnet_addr (3dn)

NAME

dnet_addr — convert ASCII node address to binary

SYNTAX

```
#include <netdnet/dn.h>

struct dn_naddr *
dnet_addr (cp)

char *cp;
```

where

Input Argument:

cp	is a character pointer to the ASCII node address string. A DECnet node address is specified as *a.n*, where *a* is the area number and *n* is the node number. A DECnet node address includes an area number (which identifies a node's area in a multiple area network) and a node number (which uniquely identifies a DECnet node). In a multiple area network, *a* is the area number for that node. For a node in a single area network, the *a* argument defaults to 1.

Return Argument:

dn_naddr specifies the node address structure. The following fields are filled in by this subroutine:

a_len	specifies the length of the returned node address.
a_addr	specifies the node address.

DESCRIPTION

The **dnet_addr** subroutine converts an ASCII node address string to binary and returns a pointer to a **dn_naddr** structure, which contains the node address and the length of the returned node address. This information is required for the **sockaddr_dn** data structure.

RETURN VALUE

If the call succeeds, a pointer to a **dn_naddr** structure is returned. If an error occurs, a value of 0 is returned.

RESTRICTION

If you plan to call this function again before you finish using the data, you must copy the data into a local structure.

dnet_conn (3dn)

NAME

dnet_conn — connect to target network object

SYNTAX

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdnet/dn.h>

int
dnet_conn (node,object,type,opt_out,opt_outl,opt_in,opt_inl)

char *node;
char *object;
u_char *opt_out,*opt_in,
int opt_outl,*opt_inl;
```

where

s	is a returned socket descriptor over which a connection has been established.
node	specifies the address of the string that contains the remote node name and any optional access data. The node string can have one of the following formats: "*nodename*[*/username/password/account*]" or "*a.n*[*/username/password/account*]" where *a* is the area number and *n* is the node number. Node names are matched without regard to case, and access-control information is passed as supplied. (Case is preserved.)

NOTE

Programs that use **dnet_conn** prompt you for a password if you omit the *password* field in an access-control string. The password that you type after the prompt does not echo, which provides account security.

object	specifies the address of the string that contains the target network object. You can specify the object by name or number. If the object number is zero, you must specify the object by name. If the object number is something other than zero, you can specify the object by name or number, but for remote non-ULTRIX nodes, it is recommended that you specify them by number. (To specify the object by name on a remote non-ULTRIX node, see the documentation for that operating system.) Specify object name and number strings as follows: By object name: "*objectname*" (For example, "test".) By object number: "#*objectnumber*" (For example, "#17") Case conversion is not performed on object names before they are sent to a destination program.
type	is the socket type. The DECnet domain currently supports the following socket types: • SOCK_STREAM (stream socket). • SOCK_SEQPACKET (sequenced-packet socket). A value of 0 defaults to SOCK_SEQPACKET.
opt_out	specifies the address of the outgoing optional user data buffer. The message can be up to 16 bytes long. If this argument is not required, supply a NULL pointer.
opt_outl	specifies the size of the optional outgoing message. The message can be up to 16 bytes long. If this argument is not required, supply a NULL value.
opt_in	specifies the address of the buffer that will store the optional incoming message. The message can be up to 16 bytes long. If this argument is not required, supply a NULL pointer.
opt_inl	specifies the size of the buffer that will store the optional incoming message. On return, this argument contains the actual size of the optional incoming message. If this argument is not required, supply a NULL pointer.

DESCRIPTION

The **dnet_conn** subroutine establishes a connection to a specified network object on a remote node. Default access-control information is used to validate access privileges. You can override the default access control by supplying optional access-control information. You can also supply optional connection data.

In addition or instead of supplying access-control information, you can request proxy access on the remote node. The *DECnet–ULTRIX Network Management* manual contains a full description of proxy access.

The **dnet_conn** subroutine requests proxy by default. If you do not want outgoing requests to ask for proxy access at the remote systems, your program should clear the global variable *proxy_requested*.

The **dnet_conn** subroutine creates a socket in the DECnet domain and binds a name to the socket. The bound socket name is the respective user's log-in name converted to uppercase. This bound name is used as the source name for the outgoing connection only when a program is running as superuser; otherwise, the user ID in ASCII is used as the source name.

When you write a program using **dnet_conn**, you must subsequently call **nerror** in order to return any DECnet system errors that occur. For example, if **dnet_conn** returns an error, use the **nerror** subroutine to display the DECnet system error.

RETURN VALUE

If the subroutine completes successfully, the socket descriptor is returned. If an error occurs, a value of -1 is returned. Additional error detail will appear in the external variable **errno**.

RESTRICTION

Currently, **dnet_conn** creates a socket in the DECnet domain and binds a name to the socket. The bound socket name is the respective user's log-in name converted to uppercase. This bound name is used as the source name for the outgoing connection only when a program is running as superuser; otherwise the user ID in ASCII is used as the source name.

DIAGNOSTICS

Use **nerror** to map the following standard ULTRIX errors to equivalent DECnet error messages. The DECnet error text is written to a standard error message.

ULTRIX Error	Equivalent DECnet Error Message
[EACCES]	Connect failed, access control rejected
[EADDRINUSE]	Connect failed, insufficient network resources
[EADDRNOTAVAIL]	Connect failed, unrecognized node name
[ECONNREFUSED]	Connect failed, connection rejected by object
[EHOSTDOWN]	Connect failed, local node shutting down
[EHOSTUNREACH]	Connect failed, node unreachable
[EINVAL]	Connect failed, invalid object name format
[EISCONN]	Connect failed, insufficient network resources
[ENAMETOOLONG]	Connect failed, invalid node name format
[ENETDOWN]	Connect failed, remote node shutting down
[ENOBUFS]	Connect failed, insufficient network resources
[ENOSPC]	Connect failed, insufficient network resources
[ESRCH]	Connect failed, unrecognized object
[ETIMEDOUT]	Connect failed, no response from object
[ETOOMANYREFS]	Connect failed, object too busy

SEE ALSO

errors(2)

dnet_eof (3dn)

NAME

dnet_eof — test for end-of-file on a DECnet socket

SYNTAX

```
int
dnet_eof (s)

int s;
```

where

s	specifies a DECnet socket.

DESCRIPTION

The **dnet_eof** subroutine tests a DECnet socket to determine if an end-of-file (EOF) condition exists. An EOF on a DECnet socket indicates that it is impossible to read any more data because no more data exists for a **read** operation and the socket is no longer connected.

This subroutine is useful for determining if an EOF condition exists on a DECnet sequenced-packet socket when a **read** operation has returned zero bytes. ULTRIX uses a returned value of zero on a **read** operation to indicate EOF. Since it is always possible to read a zero-length packet on a DECnet sequenced-packet socket, you cannot determine whether you have just read a zero-length packet or reached EOF without using **dnet_eof**.

RETURN VALUE

If **dnet_eof** determines a connection to be in an active state, a value of 0 is returned. If **dnet_eof** determines a connection to be in an inactive state, a nonzero value is returned.

RESTRICTION

Even though zero-length packets may be available for a **read** operation, **dnet_eof** will indicate an EOF condition if the DECnet socket is no longer connected.

SEE ALSO

read(2dn)

dnet_getalias (3dn)

NAME

dnet_getalias — get extended node information

SYNTAX

```
char *
dnet_getalias (alias)
char *alias;
```

where

alias	is a character pointer to an alias.

DESCRIPTION

The **dnet_getalias** subroutine searches for a **.nodes** file in your home directory and returns any alias definitions found in that file. The **dnet_getalias** subroutine returns a node name and any default access-control information associated with the node name.

RETURN VALUE

If a node has default access-control information associated with it, the node name, followed by the access data, is returned in the following format:

nodename/username/password/account

If you have a **.nodes** file in your home directory that defines aliases, any alias definition for the specified alias name is returned.

If a matching alias entry is not found, a NULL pointer is returned.

RESTRICTION

If you plan to call this function again before you finish using the data, you must copy the data into a local structure.

dnet_htoa (3dn)

NAME

dnet_htoa — return ASCII node name or node address

SYNTAX

```
#include <netdnet/dn.h>

char *
dnet_htoa (add)

struct dn_naddr *add
```

where

Input Argument:

add	specifies a pointer to a structure of the type **dn_naddr**, which contains the node address.

Return Argument:

dn_naddr specifies the node address structure. The following fields are filled in by this subroutine:

a_len	specifies the length of the returned node address.
a_addr	specifies the node address.

DESCRIPTION

The **dnet_htoa** subroutine searches the node database for a node by address. If the node is found, the ASCII node name string is returned. If the node is not found, the 16-bit binary node address is converted to the ASCII string representation (*area.number*).

RETURN VALUE

If the node name is found, a pointer to the ASCII node name string is returned. Otherwise, the ASCII node address string is returned.

RESTRICTION

If you plan to call this function again before you finish using the data, you must copy the data into a local structure.

dnet_ntoa (3dn)

NAME

dnet_ntoa — convert binary node address to ASCII

SYNTAX

```
#include <netdnet/dn.h>

char *
dnet_ntoa (add)

struct dn_naddr *add;
```

where

Input Argument:

add	specifies a pointer to a structure of the type **dn_naddr**, which contains the node address.

Return Argument:

dn_naddr	specifies the node address structure. The following fields are filled in by this subroutine:

a_len	specifies the length of the returned node address.
a_addr	specifies the node address.

DESCRIPTION

The **dnet_ntoa** subroutine converts a 16-bit binary node address to its ASCII string representation (*area.number*).

RETURN VALUE

A pointer to the ASCII string representation of the DECnet node address is returned.

RESTRICTION

If you plan to call this function again before you finish using the data, you must copy the data into a local structure.

dnet_otoa (3dn)

NAME

dnet_otoa — convert DECnet object name or number to ASCII

SYNTAX

```
#include <netdnet/dn.h>

char *
dnet_otoa (dn)

struct  sockaddr_dn *dn;
```

where

dn   is the address of a structure **sockaddr_dn**.

DESCRIPTION

Given a **sockaddr_dn** data structure, **dnet_otoa** converts a DECnet object name or number to its ASCII string representation.

RETURN VALUE

A character pointer to the ASCII string representation of the DECnet object name or number is returned.

RESTRICTION

If you plan to call this function again before you finish using the data, you must copy the data into a local structure.

getnodeadd (3dn)

NAME

getnodeadd — return local node address

SYNTAX

```
#include <netdnet/dn.h>

struct dn_naddr *
getnodeadd();
```

where

dn_naddr	specifies the node address structure. The following fields are filled in by this subroutine:

a_len	specifies the length of the returned node address.
a_addr	specifies the local node address.

DESCRIPTION

The **getnodeadd** subroutine returns a pointer to a structure of the type **dn_ naddr**, which contains the DECnet node address of your local DECnet–ULTRIX node.

RETURN VALUE

If the subroutine is successful, a pointer to a **dn_naddr** structure is returned. If an error occurs, a value of 0 is returned.

RESTRICTION

If you plan to call this function again before you finish using the data, you must copy the data into a local structure.

getnodeent (3dn)

NAME

getnodeent — get node information from network node database

SYNTAX

```
#include <netdnet/dnetdb.h>

struct  nodeent  *
getnodeent()

struct nodeent *
getnodebyname (name)
char *name

struct nodeent *
getnodebyaddr (addr,len,type)
char *addr;
int len,type;

int setnodeent ()

endnodeent()
```

where

name	specifies the address of the buffer containing the node name string.
addr	specifies the address of the buffer containing the node address string in the form returned by the **dnet_addr** subroutine.
len	is the length of the node's address string.
type	specifies the address type. This must be specified as AF_DECnet.

getnodeent returns a pointer to a structure of type **nodeent** with the following members:

```
struct nodeent {

  char           *n_name;            /* name of node */
  int            n_addrtype;         /* node address type */
  int            n_length;           /* length of address */
  char           *n_addr;            /* address */
};
```

nodeent specifies the node-address database structure. The following data fields are filled in by this subroutine:

n_name	specifies the name of the node.
n_addrtype	specifies the returned address type as AF_DECnet.
n_length	specifies the length of the address.
n_addr	specifies the network address for the node.

DESCRIPTION

Given a node name or node address, the **getnodebyname** and **getnodebyaddr** subroutines, respectively, access the network node database and return node information. Both return a pointer to a **nodeent** structure. This structure contains an entry from the network node database. The returned **nodeent** structure is stored in static memory allocated in the **getnodeent** subroutine. Therefore, to save it, you must copy it to user memory.

The **getnodebyname** and **getnodebyaddr** subroutines search sequentially from the beginning of the database until a matching host name or host address is found, or until the end of the database is found. Node addresses are always arranged in ascending numeric order.

The **setnodeent**, **getnodeent**, and **endnodeent** functions are similar to the **sethostent**, **gethostent**, and **endhostent** functions. They read through the node database and perform functions in the following order:

1. **setnodeent** sets the pointer to the beginning of the database.
2. **getnodeent** reads the next entry in the database.
3. **endnodeent** closes the database.

RETURN VALUE

setnodeent returns a value of 0 if the subroutine completes successfully; if it fails, a value of -1 is returned.

If **getnodeent** completes successfully, the address for the **nodeent** structure is returned. If an error or an EOF occurs, a value of 0 is returned.

DIAGNOSTICS

NULL pointer (0) is returned on EOF or error.

The following error messages can be returned by the database routines **getnodebyname**, **getnodebyaddr**, **getnodeent**, and **setnodeent**:

[EADDRNOTAVAIL]	No such node name in database.
[EFAULT]	Incompatible database version number.
[ENAMETOOLONG]	The node name is too long.
[ENOBUFS]	Insufficient resources to complete request.
[EPROTONSUPPORT]	Type not supported or length not a valid length.

getnodename (3dn)

NAME

getnodename — return local node name

SYNTAX

```
char *
getnodename()
```

DESCRIPTION

The **getnodename** subroutine returns the ASCII string representation of your local DECnet–ULTRIX node name.

RETURN VALUE

If the subroutine is successful, your local DECnet node name is returned. If an error occurs, a value of 0 is returned.

nerror (3dn)

NAME

nerror — produce DECnet error messages

SYNTAX

```
void
nerror (s)
char *s;
```

where

s	is the program name, such as **dlogin**, since the value of the *s* argument is the name of the program that incurred the error (as it is with **perror**).

DESCRIPTION

The **nerror** subroutine produces **dnet_conn** error messages by mapping standard ULTRIX errors to the appropriate DECnet error message. The resulting DECnet error text is written to the standard error file. The error number is taken from the external variable **errno**, which is set when errors occur, but is not cleared when nonerroneous calls are made.

The DECnet error text is output to the standard error file.

RETURN VALUE

This function returns no value.

SEE ALSO

dnet_conn (3dn)

Appendix A

DECnet–ULTRIX Data Structures

This appendix shows the DECnet–ULTRIX data structures. For guidelines on specifying these data structures, see the relevant system calls and the header file **/sys/netdnet/dn.h**.

A.1 Access-Control Information Data Structure

```
struct accessdata_dn {
     unsigned short  acc_accl;       /* length of account string */
     unsigned char   acc_acc[40];    /* account string */
     unsigned short  acc_passl;      /* length of password string */
     unsigned char   acc_pass[40];   /* password string */
     unsigned short  acc_userl;      /* length of user string */
     unsigned char   acc_user[40];   /* user string */
};
```

A.2 DECnet Node Address Data Structure

```
# define DN_MAXADDL    2

    struct dn_naddr {
            unsigned short a_len;           /* length of address */
            unsigned char a_addr[DN_MAXADDL]; /* address as bytes */
        };
```

NOTE

The structure member *a_addr* represents the DECnet Phase IV node address. It is a 16-bit unsigned value, where bits 0-9 are the node address and bits 10-15 are the area number.

A.3 Logical Link Information Data Structure

```
struct linkinfo_dn {
       unsigned short  idn_segsize;    /* segment size for link */
       unsigned char   idn_linkstate;  /* logical link state */
};
```

A.4 Optional User Data Structure

```
struct optdata_dn {
        unsigned short  opt_status;     /* extended status return */
        unsigned short  opt_optl;       /* length of user data */
        unsigned char   opt_data[16];   /* user data */
};
```

A.5 Socket Address Data Structure

```
struct sockaddr_dn {
        unsigned short  sdn_family;     /* AF_DECnet */
        unsigned char   sdn_flags;      /* flags */
        unsigned char   sdn_objnum;     /* object number */
        unsigned short  sdn_objnamel;   /* size of object name */
                char    sdn_objname[16];        /* object name */
        struct dn_naddr  sdn_add;       /* node address */
};
```

Appendix B

DECnet–ULTRIX Programming Examples

This appendix presents the following types of programming examples:

- A sample client program using **dnet_conn**
- A sample server program using the **dnet_spawner**
- A sample client program using system calls
- A sample server program using system calls
- A sample gateway program

These programming examples are also available on line in **/usr/examples/decnet**.

B.1 Sample Client Program Using dnet_conn

```
#ifndef lint
static  char  *sccsid = "@(#)dnet_echo1.c       1.5  7/27/90";
#endif lint

/*
 *
 * d e c n e t   e x a m p l e : d n e t   e c h o 1
 *
 * Description: This client program connects to the partner server
 *              program "dnet_echo1d" on the node specified on the
 *              command line.  Once connected, lines are read from
 *              standard input and shipped over the network to
 *              dnet_echo1d, which then echoes the lines back to
 *              this program, which then prints them on standard
 *              output.
 *
 * Input:       Name of the node on which dnet_echo1d will run.
 *
 * Output:      none
 *
 * To compile:  cc dnet_echo1.c -ldnet -o dnet_echo1
 *
 * Examples:    dnet_echo1 boston
 *              dnet_echo1 boston/jones/topsecret
 *
 * Comments:    This example illustrates the use of dnet_conn to
 *              establish a network connection.   Compare the
 *              simplicity of using dnet_conn with doing the connect
 *              with system calls, as shown in example dnet_echo2.c.
 *              Whether access control is required depends on how the
 *              companion program (dnet_echo1d) was set up.
 */

/*
 * Digital Equipment Corporation supplies this software example on
 * an "as-is" basis for general customer use.  Note that Digital
 * does not offer any support for it, nor is it covered under any
 * of Digital's support contracts.
 */

#include <stdio.h>

#define BUFSIZE 1024        /* size of buffer for read/write */

main(argc, argv)
int argc;
char *argv[];
{
    int sock;               /* socket for connection */
    int length;             /* length of data        */
    char buff[BUFSIZE];     /* buffer for data       */

    /*
     * Make sure the node name was given on the command line
     */
    if( argc < 2 ) {
        fprintf(stderr, "Usage: %s nodename\n", argv[0]);
        exit();
    }
```

```
        /*
         * connect to our partner "dnet_echold" on the specified node
         */
        sock = dnet_conn(argv[1], "dnet_echold", 0, 0, 0, 0, 0);
        if( sock < 0 )
        {
            /* print DECnet specific connect error */
            nerror (argv[0]);
                    exit();
        }

        puts("Connected!");

        /*
         * read lines from standard input, send them over the network,
         * then read and print the echoed line from our partner
         */
        while( gets(buff) != NULL )
        {
            length = strlen(buff);

            /* Only send nonempty lines */
            if( length > 0 ) {
                if( write(sock, buff, length) < 0 ) {
                    perror("couldn't send line over network");
                    break;
                }

                if( (length = read(sock, buff, BUFSIZE)) < 0 ) {
                    perror("couldn't read line over network");
                    break;
                }

                buff[length] = '\0';
                puts(buff);
            }
        }

        /*
         * finished - close network connection and exit
         */
        puts("Exiting...");
        close(sock);
}
```

B.2 Sample Server Program Using the dnet_spawner

```
#ifndef lint
static  char  *sccsid = "@(#)dnet_echo1d.c      1.5  7/27/90";
#endif lint

/*
 *
 * d e c n e t   e x a m p l e : d n e t   e c h o 1 d
 *
 * Description: This server program reads messages from  the network
 *              connection and then writes(echoes) them back to the
 *              network connection, until the network connection is
 *              broken.
 *
 * Input:       none
 *
 * Output:      none
 *
 * To compile:  cc dnet_echo1d.c -ldnet -o dnet_echo1d
 *
 * Comments:    This example illustrates the use of the dnet_spawner
 *              to listen for incoming connect requests on behalf of
 *              other server programs.  Note that standard in and
 *              standard out are directed to the network connection
 *              (socket) by the dnet_spawner before executing this
 *              program. Compare the simplicity of using the spawner
 *              with the complexity of using system calls as is done
 *              in example dnet_echo2d.c.
 *
 *              To work with the spawner the decnet object database
 *              must be properly configured. There are two
 *              alternatives:
 *                1)  A person with superuser privileges must define
 *                    this object using ncp, for example:
 *                    ncp set object dnet_echo1d \
 *                        file /usr/examples/dnet_echo1d
 *                2)  Object "DEFAULT" must be defined in the object
 *                    database (DECnet comes with DEFAULT defined),
 *                    and this program must be moved to a directory
 *                    searched by the spawner (for example, /usr/bin).
 *              In either case, if no default user is defined for the
 *              object, access control information must be specified
 *              by the client (dnet_echo1) when attempting to
 *              connect.  If a default user is defined, and such an
 *              account actually exists, then access control is not
 *              required (although it can still be specified if
 *              desired).
 */

/*
 * Digital Equipment Corporation supplies this software example on
 * an "as-is" basis for general customer use.  Note that Digital
 * does not offer any support for it, nor is it covered under any
 * of Digital's support contracts.
 */

#include <stdio.h>

#define BUFSIZE 1024   /* size of buffer for read/write */

#define TRUE    1
#define FALSE   0

main()
{
    char buff[BUFSIZE];
    int length;
```

```
        while( TRUE )
    {
        /*
         * read messages from standard input,
         * write them to standard output
         */
        length = read(0, buff, sizeof(buff));
        if( length <= 0 )
             /* if at "end-of-file" (connection broken) */
            if( dnet_eof(0) )
                break;

        write(1, buff, length);
    }
}
```

B.3 Sample Client Program Using System Calls

```
#ifndef lint
static  char  *sccsid = "@(#)dnet_echo2.c      1.5  7/27/90";
#endif lint

/*
 *
 * d e c n e t   e x a m p l e : d n e t   e c h o 2
 *
 * Description: This client program connects to the partner server
 *              program "dnet_echo2d" on the node specified on the
 *              command line.  Once connected, lines are read from
 *              standard input and shipped over the network to
 *              dnet_echo2d, which then echoes the lines back to
 *              this program, which then prints them on standard
 *              output.
 *
 * Input:       Name of the node on which dnet_echo2d will run.
 *
 * Output:      none
 *
 * To compile:  cc dnet_echo2.c -ldnet -o dnet_echo2
 *
 * Example:     dnet_echo2 boston
 *
 * Comments:    This example illustrates the use of system calls
 *              to establish a network connection.  Compare this
 *              method with that of using dnet_conn as is done in
 *              dnet_echo1.c.  Also, the connect request is by
 *              object number, rather than by object name as was
 *              done in dnet_echo1.c.
 */

/*
 * Digital Equipment Corporation supplies this software example on
 * an "as-is" basis for general customer use.  Note that Digital
 * does not offer any support for it, nor is it covered under any
 * of Digital's support contracts.
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdnet/dn.h>
#include <netdnet/dnetdb.h>

#define BUFSIZE 1024        /* size of buffer for read/write */

main(argc, argv)
int argc;
char *argv[];
{
    int sock, length;
    char buff[BUFSIZE];
    struct sockaddr_dn address;
    struct dn_naddr *node_addr;
    struct nodeent *nodep;

    if( argc < 2 ) {
        fprintf(stderr, "Usage: %s nodename\n", argv[0]);
        exit();
    }

    /* Create a socket in DECnet domain */
    if( (sock = socket(AF_DECnet, SOCK_SEQPACKET, 0)) < 0 ) {
        perror(argv[0]);
        exit();
    }
```

```
        /* Specify target object number for connection */
        bzero(&address, sizeof(address));
        address.sdn_family = AF_DECnet;
        address.sdn_objnum = 128;

        if( (node_addr = dnet_addr(argv[1])) == NULL ) {
            if( (nodep = getnodebyname(argv[1])) == NULL ) {
                fprintf(stderr, "%s: Node unknown\n", argv[1]);
                exit();
            }
            else {
                bcopy(nodep->n_addr,address.sdn_nodeaddr,nodep->n_length);
                address.sdn_nodeaddrl = nodep->n_length;
            }
        }
        else
            address.sdn_add = *node_addr;

        /* Connect to partner on specified node */
        if ( connect(sock, &address, sizeof(address)) < 0 ) {
            perror(argv[0]);
            exit();
        }

        puts("Connected...");

        /* Read lines from standard input, send them over */
        /* the network connection, and print the response */

          while( gets(buff) != NULL )
        {
            length = strlen(buff);

            /* Only send non-empty lines */
            if( length > 0 ) {
                if( write(sock, buff, length) < 0 ) {
                    perror("couldn't send line over network");
                    break;
                }

                if( (length = read(sock, buff, BUFSIZE)) < 0 ) {
                    perror("couldn't read line over network");
                    break;
                }

                buff[length] = '\0';
                puts(buff);
            }
        }

        printf("Exiting...\n");     /* Close link and exit */
        close(sock);
}
```

B.4 Sample Server Program Using System Calls

```
#ifndef lint
static  char  *sccsid = "@(#)dnet_echo2d.c      1.5  7/27/90";
#endif lint

/*
 *
 * d e c n e t  e x a m p l e : d n e t  e c h o 2 d
 *
 * Description: This server program is designed to run as a daemon.
 *              When a network connection is created, it forks and
 *              executes a child, which then reads messages from the
 *              network connection and then writes(echoes) them back
 *              to the network connection, until the network
 *              connection is broken.
 *
 * Input:       none
 *
 * Output:      none
 *
 * To compile:  cc dnet_echo2d.c -ldnet -o dnet_echo2d
 *
 * Example:     dnet_echo2d &
 *
 * Comments:    This example illustrates the programming of a server
 *              that does not use the dnet_spawner.  This program
 *              must be started manually before attempting to connect
 *              to it from the client program (dnet_echo2).  Note
 *              that no access control verification is done in this
 *              example, but a "real" server program would
 *              need to do some form of access control verification
 *              (in example dnet_echo1d, access control verification
 *              is done automatically by the dnet_spawner).
 */

/*
 * Digital Equipment Corporation supplies this software example on
 * an "as-is" basis for general customer use.  Note that Digital
 * does not offer any support for it, nor is it covered under any
 * of Digital's support contracts.
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdnet/dn.h>
#include <sys/wait.h>
#include <signal.h>
#include <errno.h>

#define BUFSIZE 1024        /* size of buffer for read/write */

main(argc, argv)
int argc;
char *argv[];
{
    int s, ns, acclen, rdlen;
    char buf[BUFSIZ];
    struct sockaddr_dn sockaddr, accsockaddr;

    /* Create socket in DECnet address family */
    /* of type sequenced packet.              */
    if ( (s = socket(AF_DECnet, SOCK_SEQPACKET, 0)) < 0 ) {
        perror(argv[0]);
        exit();
    }
```

```
        /* The socket address indicates the DECnet address */
        /* and object number of 128.                       */
        bzero(&sockaddr, sizeof(struct sockaddr_dn));
        sockaddr.sdn_family = AF_DECnet;
        sockaddr.sdn_objnum = 128;

        /* Bind the socket to a DECnet socket address and */
        /* listen for a connection.                       */
        if( bind(s, &sockaddr, sizeof(sockaddr)) < 0 ) {
            perror(argv[0]);
            exit();
        }

        if( listen(s, SOMAXCONN) < 0 ) {
            perror(argv[0]);
            exit();
        }

        /* Accept an incoming connection */
        for( ; ; ) {
            do {
                acclen = sizeof(accsockaddr);
                ns = accept(s, &accsockaddr, &acclen);
            } while( ns == -1  &&  errno == EINTR );

        /* Fork child to handle the new connection */
        if( fork() == 0 )
            break;
        close(ns);
        }

        /* Redirect standard input and output to new socket */
        dup2(ns, 0); dup2(ns, 1);
        close(ns); close(s);

        for( ; ; ) {
        if( (rdlen = read(0, buf, sizeof(buf))) <= 0 )
            if( dnet_eof(0) )
                break;

        write(1, buf, rdlen);
        }
}
```

B.5 Sample Application Gateway Program

```
#ifndef lint
static  char  *sccsid = "@(#)gatewayd.c        1.4  5/27/90";
#endif lint

/*
 * DESCRIPTION
 *
 *   This program illustrates how an ULTRIX system can be used as a
 *   gateway to swap transports for an application protocol.  A brief
 *   description of how the program is used is given below.  For more
 *   details on usage, see file /usr/examples/decnet/gatethru/README
 *
 * USAGE
 *
 *   gatewayd
 *   gatewayd -inet desthost destservice
 *   gatewayd -dnet destnode destobject
 *
 *   In the first case (with no arguments specified), three
 *   lines should be sent initially:
 *
 *           protocol      ("inet" or "dnet" without the quotes)
 *           destsystem    (host or node name)
 *           destentity    (service or object name)
 *
 *           These must be delimited by one of the following:
 *
 *              <LF>       (linefeed)
 *              <CR><LF>   (carriage-return linefeed)
 *
 *           A response will be returned as:
 *
 *              Connected to destsystem (destentity) via protocol
 *
 *           for success, and
 *
 *              Not-Connected [further explanation]
 *
 *           for failure.  These responses will be delimited by the
 *           same delimiter that had delimited the protocol.
 *
 *   In the other cases, there will be no exchange.  If the
 *   connection couldn't complete to the destsystem/destentity,
 *   the connection to the client is simply disconnected.
 *
 *   In all cases, "service" must be defined in /etc/services on the
 *   gateway system, and "host" must be in /etc/hosts.
 */

/*
 * Digital Equipment Corporation supplies this software example on
 * an "as-is" basis for general customer use.  Note that Digital
 * does not offer any support for it, nor is it covered under any
 * of Digital's support contracts.
 */

#include <stdio.h>
#include <strings.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <netdnet/dn.h>
#include <netinet/in.h>
#include <netdb.h>

#include <signal.h>
```

```
#include <errno.h>
#include <sysexits.h>

#define     STREQL(a, b)        (strcmp(a, b) == 0)
#define     NIL                 (0)

char      DestProto[40];    /* Protocol family to connect by       */
char      DestHost[256];    /* Remote system to connect to         */
char      DestObj[256];     /* Remote object/service to connect to */

char      LineDelim[10] = "\n"; /* Default line delim in smart mode */

int      SmartMode;         /* Get destination info from client mode */

/* Signal handler for SIGPIPE (write on a disconnected socket) */
abort()
{
    /* We still had data to transfer and the side who should have
       received it has gone away.  We will consider it an I/O error */
    exit(EX_IOERR);
}

main(argc, argv)
int    argc;            /* # of command line arguments */
char   *argv[];         /* the command line arguments  */
{
    int      client,        /* Socket connected to client  */
             server;        /* Socket to use for server    */

    /* Check usage */
    if( !(argc == 4  ||  argc == 1) )
      exit(EX_USAGE);

    /* If no arguments, operate in Smart Mode */
    SmartMode = argc == 1;

    if( !SmartMode ) {
      /* Fetch connect info from command line */
      strcpy(DestProto, &argv[1][1]);
      strcpy(DestHost,   argv[2]);
      strcpy(DestObj,    argv[3]);
    }
    else {
      char *p;        /* Temp */

      /* Get connect info from client */
      fgets(DestProto, sizeof(DestProto), stdin);
      fgets(DestHost,  sizeof(DestProto), stdin);
      fgets(DestObj,   sizeof(DestProto), stdin);

      p = strpbrk(DestProto, "\r\n");
      strcpy(LineDelim, p);
      *p = '\0';

      strpbrk(DestHost, "\r\n")[0] = '\0';
      strpbrk(DestObj,  "\r\n")[0] = '\0';
    }

    /* Time to attempt the connection */

    if( STREQL(DestProto, "dnet") ) {
      server = dnet_conn(DestHost, DestObj, SOCK_STREAM,
                         (u_char *)0, 0, (u_char *)0, (int *)0);

      if( server < 0 ) {
          /* Return failure indication back to client in Smart Mode */
          if( SmartMode ) {
              printf("Not-Connected%s", LineDelim);
              fflush(stdout);
          }
```

```
                /* Some spawners (DECnet) log children's exit codes */
                exit(EX_CANTCREAT);
            }
        }
        else if( STREQL(DestProto, "inet") ) {
            server = inet_conn(DestHost, DestObj);

            if( server < 0 ) {
                /* Return failure indication back to client in Smart Mode */
                if( SmartMode ) {
                    printf("Not-Connected%s", LineDelim);
                    fflush(stdout);
                }

                /* Some spawners (DECnet) log children's exit codes */
                exit(EX_CANTCREAT);
            }
        }
        else {
            /* Error; Request to connect via an unknown protocol */

            /* Return failure indication back to client in Smart Mode */
            if( SmartMode ) {
                printf("Not-Connected Unknown protocol %s%s",
                        DestProto, LineDelim);
                fflush(stdout);
            }

            /* Some spawners (DECnet) log children's exit codes */
            exit(EX_PROTOCOL);
        }

        /* Return success indication back to client in Smart Mode */
        if( SmartMode ) {
            printf("Connected to %s (%s) via %s%s",
                DestHost, DestObj, DestProto, LineDelim);
            fflush(stdout);
        }

        /* Just to make the code more readable */
        client = 0;

        /* We will abort gracefully when the client or remote system
           goes away */
        signal(SIGPIPE, abort);

        /* Now just go and move raw data between client and
           remote system */
        dowork(client, server);
        /* ... NEVER RETURNS ... */
}

dowork(client, server)
int     client, server;
{

/* select(2) masks for client and remote */
int       ClientMask, ServerMask;

/* Combined ClientMask and ServerMask */
int       ReadMask;

        /* Initialize select(2) masks */
        ClientMask = 1<<client;
        ServerMask = 1<<server;

        ReadMask = ClientMask | ServerMask;

        /* Now move raw data for the rest of our life between
           client and remote */
        for( ; ; ) {
            /* Local Variables */
            int  SelectReadMask;/* select(2) mask modifiable by select(2) */
            int  nready;        /* status return from select(2)           */
        do {
            /* Intialize select(2) mask every time
               as select(2) always modifies it */
            SelectReadMask = ReadMask;

            /* Wait for data to be present to be moved */
            nready = select(32,&SelectReadMask,(int *)0,(int *)0,NIL);
        } while( nready < 0  &&  errno == EINTR );

        /* select(2) failed, should not happen.  Exit abnormally */
        if( nready < 0 )
            exit(EX_SOFTWARE);

        /* Favor the client (unspecified reason)
           if s/he has data */
        if( SelectReadMask & ClientMask )
            xfer(client, server);

        /* Then check on the other operation*/
        if( SelectReadMask & ServerMask )
            xfer(server, client);
    }

    /* NEVER REACHED */
}

#define      BUFSIZE        256 /* Max bytes to move at a time */

xfer(from, to)
int      from, to;        /* Move data from "from" to "to" */
{
static char buf[BUFSIZE];      /* Buffer data to be moved      */
int      nready;               /* # bytes readable             */
int      got;                  /* # bytes actually being moved */

    /* Query the system how many bytes are ready to be read */
    ioctl(from, FIONREAD, &nready);

    /* Only try to get the smaller of nready and BUFSIZE */
    got = read(from, buf, nready < BUFSIZE ? nready : BUFSIZE);

    /* Zero bytes returned indicates end of stream, exit gracefully */
    if( got == 0 )
      exit(EX_OK);

    /* Now send it across to the other side */
    write(to, buf, got);
}

int
inet_conn(host, port)
    char *host;
    char *port;
{
/* Local Vars */
int                sock;      /* Socket to use for the connection */
struct hostent     *hostent;  /* Destination host entry           */
struct servent     *servent;  /* Destination service entry        */
struct sockaddr_in addr;      /* Formatted destination for connect */

    /* Fetch the requested host and service entries */
    hostent = gethostbyname(host);
    servent = getservbyname(port, "tcp");

    /* No host entry, no service entry, or host is not
       Internet, error! */
    if( servant == NIL ||
      hostent == NIL ||
      hostent->h_addrtype != AF_INET )
      return -1;

    /* Get a socket from the system to use for the connection */
    if( (sock = socket(AF_INET, SOCK_STREAM, 0)) < 0 )
      return -1;
```

```
        /* Make sure we start with a clean address structure ... */
        bzero(&addr, sizeof(addr));

        /* ... then fill in the required fields */
        addr.sin_family = AF_INET;
        addr.sin_port   = servant->s_port;
        bcopy(hostent->h_addr, &addr.sin_addr, hostent->h_length);

        /* Now try connection to the destination */
        if( connect(sock, &addr, sizeof(addr)) < 0 ) {
          /* No go, release the socket, and then return error! */
          close(sock);
          return -1;
        }

        /* Success.  Return the connected socket descriptor */
        return sock;
}
```

Glossary

Accept-Deferred Mode

A mode for accepting incoming connections. **Deferred mode** lets the server program store, examine, and process any access control information or optional data that is supplied as part of a connection request.

Accept-Immediate Mode

A mode for accepting incoming connections. **Immediate mode** makes it possible for the server program to send and receive data as soon as the accept call operation completes.

Access Control Information

Identification information used to screen inbound connect requests and verify them against a system account. In the DECnet domain, access control information consists of a specified user name, password, and account string.

Blocking Input/Output (I/O)

An I/O mode that causes a calling process to wait for an input/output operation. Blocking prevents an input/output system call from returning control to a calling procedure until the operation completes. See also **Nonblocking Input/Output**.

Client Application

Any application that initiates a connection and requests services from the server application.

Client-Server Communication

Task-to-task communication between applications through a socket interface.

Communication Domain

A set of protocols that have common communication properties. For example, the Internet domain supports applications that communicate through the Defense Advanced Research Projects Agency (DARPA) standard communication protocols, and the DECnet domain supports applications that communicate through the Digital Network Architecture.

Digital Data Communications Message Protocol (DDCMP)

A set of conventions used for data transmission over physical links.

Interprocessor Communication (IPC)

Communication between two independent processes, such as client and server programs. These processes use system calls to establish connections and communicate with each other through sockets.

Network Object

A task or program (for example, fal or nml) that provides generic services across a network. In the DECnet–ULTRIX programming environment, a network object is a server application that can be accessed from other Internet or DECnet nodes on a network.

Nonblocking Input/Output (I/O)

A mode that causes a calling process to not wait for an I/O operation. The nonblocking input/output mode returns control to the calling procedure immediately with an error message if there are not enough resources available to complete the operation. See also **Blocking Input/Output**.

Optional data

In the DECnet domain, a string of up to 16 bytes that clients and servers can exchange on either a connect or disconnect sequence. This data is interpreted differently according to the application.

Out-of-Band Message

An unsolicited, high-priority message that one application sends to another outside of the normal data channel. In most cases, it informs the receiving application of an unusual or abnormal event in the sending application.

Proxy Access

A method of screening client application access to a server application without using a password. The supplied name of the user making the access request must correspond with an entry listed in the target node's proxy access file.

Sequenced-Packet Socket

A socket type that preserves record boundaries and supplies a bidirectional, reliable, ordered, first-in, first-out (FIFO), unduplicated flow of data.

Server Application

Any application that either accepts or rejects a request from a client application and provides services to client applications.

Stream Socket

A socket type that provides a byte stream without using message boundaries. It also supplies a bidirectional, reliable, ordered, first-in, first-out (FIFO), unduplicated flow of data.

Socket

An addressable endpoint for communication. Client and server applications each create a socket that acts as a handle for sending and receiving data.

Index

DECnet–ULTRIX
Programming
AA–EA88D–TE

READER'S COMMENTS

What do you think of this manual? Your comments and suggestions will help us to improve the quality and usefulness of our publications.

Please rate this manual:

	Poor				Excellent
Accuracy	1	2	3	4	5
Readability	1	2	3	4	5
Examples	1	2	3	4	5
Organization	1	2	3	4	5
Completeness	1	2	3	4	5

Did you find errors in this manual? If so, please specify the error(s) and page number(s).

____
____
____
____

General comments:

____
____
____
____

Suggestions for improvement:

____
____
____
____

Name ______________________ Date ______________
Title ______________________ Department ______________
Company ______________________ Street ______________
City ______________________ State/Country ________ Zip ________

DO NOT CUT – FOLD HERE AND TAPE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY LABEL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

digital™

Networks and
Communications Publications
550 King Street
Littleton, MA 01460–1289

DO NOT CUT – FOLD HERE

ON THIS LINE