4DDN Programming Guide and Man Pages

Document Number 007-1302-020

CONTRIBUTORS

Written by Pam Sogard
Edited by Loraine McCormick
Production by Laura Cooper and Diane Wilford
Engineering contributions by John Ng
Other contributions by Bent Jensen, Ray Niblett, Sam Sengupta and John Talbott
St. Peter's Basilica image courtesy of ENEL SpA and InfoByte SpA. Disk Thrower image courtesy of Xavier Berenguer, Animatica.

This manual has been adapted from the commUnity-UNIX User's Guide by Bell Atlantic Software Systems, Inc. This material may be changed without notice by Bell Atlantic Software Systems, Inc., or by Silicon Graphics, Inc. Bell Atlantic Software Systems, Inc., is not responsible for any errors that may appear herein.

4DDN Programming Guide and Man Pages
Document Number 007-1302-020

Contents

Figures

Tables

Chapter 1

Introduction

Silicon Graphics®' *4DDN™ Programming Guide* explains the software tools available for developing network applications for IRIS®-4D™ workstations running 4DDN software. It describes the library of routines and task-to-task communication that can be used in C programs to access files and establish logical links across a DECnet™ network.

The *4DDN Programming Guide* is one of a three-volume set that documents the 4DDN product. Other members of the set are the *4DDN User's Guide*, which explains how to use the interactive commands that provide user-level services, and the *4DDN Network Management Guide*, which explains how to configure 4DDN software and monitor and control the network from a 4DDN node.

1.1 Using This Guide

The *4DDN Programming Guide* assumes you are experienced with the DECnet environment and with programming in the C language. The information in this guide also requires some familiarity with using IRIX™ input/output operations. If you are not familiar with the IRIX programming environment, refer to the sections entitled "Programming in a UNIX System Environment" and "Programming Basics" in Volume 1 of the *IRIS-4D Programmer's Guide*.

The *4DDN Programming Guide* is organized into these chapters:

Chapter 1 — This Introduction explains the purpose of the *4DDN Programming Guide* and gives information on how to use it. Chapter 1 also gives a brief description of 4DDN user-level services.

Chapter 2	"Notes to New Users" explains the architecture of the DECnet network and the components of 4DDN software. It also gives quick tips on using IRIX, and provides a table of command equivalents for VMS™, IRIX, and 4DDN.
Chapter 3	"Using Network File Access Routines" describes the services provided by the Network File Access Routines (NFARS), the elements from which network applications are constructed. It also explains how to include NFARS in C programs and link them to the object library.
Chapter 4	"Using Task-to-Task Communications" describes how logical links are established between processes running on different network nodes. This chapter provides the background information needed to use the subroutines described in Chapter 5.
Chapter 5	"Task-to-Task Communication Programming Reference" provides a detailed description of each subroutine call used in the 4DDN task-to-task communication service.
Appendix A	"NFARS Error Messages" lists each error message that Network File Acccess Routines can generate and recommends corrective action.
Appendix B	"4DDN Error Codes" describes programming error codes and recommends remedial actions.
Appendix C	"Sample Programs" contains a sample client and server program for exchanging data across the network.
Appendix D	"Glossary" defines terms used in 4DDN documentation.
Appendix E	"Man Pages" contains the IRIX man pages for NFARS routines and task-to-task communication services.

1.2 Conventions

This document uses the standard UNIX™ conventions when referring to entries in the IRIX documentation. The entry name is followed by a section number in parentheses. For example, *cc*(1) refers to the *cc* manual entry in Section 1 of the *IRIS-4D User's Reference Manual, Volume 1*.

In body text, commands and file and directory specifications entered on IRIX systems appear in *italics*, while commands entered on VMS systems appear in *UPPER-CASE ITALICS*. In addition, we use these typographical conventions throughout this guide:

`typewriter font`
: Command syntax descriptions, examples, and screen displays appear in typewriter font.

`bold typewriter`
: User entries are shown in bold typewriter font.

bold
: Command options are shown in bold when they appear in body text. Options do not appear in bold in syntax descriptions or examples.

UPPERCASE
: Error codes and function options appear in uppercase letters.

node-name
: Multi-word variables are hyphenated.

[ ]
: Command arguments that are optional appear in square brackets.

1.3 Related Documentation

4DDN User's Guide
Silicon Graphics, Inc.

4DDN Network Management Guide
Silicon Graphics, Inc.

IRIS-4D Programmer's Guide
Silicon Graphics, Inc.

DECnet Digital Network Architecture (Phase IV) General Description
Digital Equipment Corporation

Guide to Networking on VAX/VMS
Digital Equipment Corporation

1.4 Product Support

Silicon Graphics provides a comprehensive product support and maintenance program for IRIS products. For further information, contact your service organization.

1.5 Overview of 4DDN

4DDN is a Silicon Graphics' software option that connects IRIS-4D series workstations and servers to a DECnet network. It provides DECnet connection and data transfer services to interactive users, file access routines and task-to-task communication service for applications programming, and a suite of Network Control Program (NCP) commands for network management.

4DDN is an implementation of Digital Network Architecture (DNA) protocols and runs on a DECnet Phase IV network. IRIS workstations running 4DDN operate as Ethernet® end nodes in the DECnet network.

In addition to 4DDN, your IRIS workstation might also be running TCP/IP protocols, the standard networking software shipped with the IRIS system. Both TCP/IP and 4DDN software can run simultaneously on an IRIS workstation. Although an IRIS can be configured with multiple Ethernet interfaces, 4DDN can run on only one Ethernet interface at a time.

1.6 4DDN User Services

4DDN user services are provided by a set of commands that you enter at your workstation to connect to other DECnet nodes (computers) or transfer data over the network. 4DDN commands provide these services:

sethost Logs you on to a remote node for a virtual terminal session. During the session, the commands you enter are executed by the remote node, rather than by your workstation.

dnls Lists the contents of a directory on a remote node in the network.

dnmv Moves or renames a file on a remote node in the network.

dnrm Removes a file in a remote directory on the network.

dncp	Transfers files to, from, or between remote nodes on the network.
dnex	Locates a command files on a remote node and submits it for execution.
dnlp	Prints a file to a network printer.
dnMail	Provides mail service between IRIS workstations running 4DDN and VMS nodes on the network.

Chapter 2

Notes to New Users

This chapter acquaints readers with some of the concepts and terms that apply to networking in general, and others that are specific to the DECnet network. It explains how network software is organized, describes the functional components of the DECnet network, and identifies the 4DDN software components that support each network service.

It also gives some tips for using the IRIX user interface to enter 4DDN commands, and a table of equivalent commands for VMS, IRIX, and 4DDN.

2.1 If You Are New to Networks

A network is a configuration of computers that permits the exchange of data and sharing of resources among its members. Incompatibilities in hardware and software components in the network must be resolved in order for an intelligent exchange of information to take place. To resolve these differences, the government and private industry have collaborated to set standards for the development of networking products.

Under the guidance of the International Standards Organization (ISO), a set of standards have been developed in the communications industry that specify a network architecture based on layers. This architecture is known as the ISO model for Open Systems Interconnection (ISO/OSI). Each layer in the model conforms to rules that govern its structure and function. Implementations of the standards are referred to as *protocols*.

2.1.1 Network Protocols

Each protocol layer in the network operates according to specific rules to perform a function and deliver a service to the layer above it. In addition to the interface between the layer above and below, each layer also communicates with a peer layer running in other computers connected to the network. Since layer functions are standardized, peer layers running in different computers can be supplied by different vendors.

Two widely used protocols are Transmission Control Protocol, Internet Protocol (TCP/IP), developed by the Department of Defense, and Digital Network Architecture (DNA), developed by Digital Equipment Corporation (DEC™). The structure and function of TCP/IP and DNA protocols are similar to those specified by the ISO/OSI model.

2.1.2 Message Packets

Information travels through the network in *packets*, small blocks of data prefixed with addressing and control information. Packets originate at the upper network layer and move down succeeding layers until they arrive at the transmission hardware. As each layer receives the packet, it adds header information and passes the packet down to the layer below it.

After a packet is transmitted over network hardware, it is moved up through the layers on the destination computer. Each layer of receiver software strips off header information from the packet and passes it to the layer above it. The end result of this process is a structure that the receiving machine can understand and use.

2.2 The Architecture of DECnet

DECnet, an implementation of DNA that provides connectivity among Digital Equipment systems, is composed of seven layers. In local areas of a DECnet network, the first two layers are frequently an implementation of Ethernet, a specification developed by DEC, Intel Corporation, and Xerox Corporation. DECnet layers perform these functions:

1. **Physical Link Layer** provides the electrical and mechanical support, as well as the software device drivers for network equipment.

2. **Data Link Layer** separates the data from noise coming in over the communication line, frames the data, and corrects transmission errors.

3. **Routing Layer** forwards packets to their destination and controls packet traffic on the network.

4. **End Communications Layer** controls the addressing and timing of data exchanged between communicating processes in different nodes.

5. **Session Control Layer** adapts data received from the End Communications Layer to the specific needs of the local operating system.

6. **Network Application Layer** controls functions required for file transfers, virtual terminal service, and remote access to files and devices on the local system.

7. **User Layer** provides resource sharing, file access and transfers, and the interface to network management tools.

In addition to these layers, DNA also includes a Network Management Layer that spans other layers in the hierarchy. Network Management provides the services needed to plan, control, and maintain the network.

Figure 2.1 illustrates the layers in DNA networks.

Figure 2-1 DNA Network Layers

2.3 If You Are New to DECnet

DECnet is the name given to a collection of hardware and software products that enables computers running DEC operating systems to be members of a network. Since IRIS workstations run IRIX, a Silicon Graphics implementation of the UNIX operating system, they require specialized software, 4DDN, to provide DECnet connectivity. An IRIS workstation running 4DDN is often called a *4DDN node* on the DECnet network.

2.3.1 DECnet Components

A DECnet network contains two kinds of nodes: end nodes and router nodes. An *end node* is one that can send packets to other DECnet nodes and receive packets that bear its address. An IRIS workstation running 4DDN operates as an Ethernet end node on the DECnet.

A *router node* is one that can receive its own packets and forward packets addressed to other nodes. When a particular router is assigned to route messages for end nodes, it is considered the *designated router*.

Nodes attached to the same Ethernet line are considered *adjacent nodes*. Nodes are considered *reachable* when they are available for connections to other nodes. The availability of a node can be controlled by turning its *state* on or off.

The data path between a local node and a remote one is called a *circuit*. The connection between two processes that occurs over a circuit is called a logical *link*. A circuit can support simultaneous logical links between network nodes. The physical media that connects computers and support circuits are called *lines*.

Network software maintains information on events that occur at network nodes, circuits, links, and lines. Event information is saved in *counters*, which are used to track network performance and throughput. This information can be retrieved with the NCP, the user interface to DECnet management.

2.3.2 DECnet Node Identification

A node name and address identify nodes in the DECnet network. If nodes are members of a subnetwork, or *area*, the address includes an area number. Name and address information is stored in a database on each node and used to route connections to other network nodes. Node names and addresses follow these conventions:

Node Address — is a unique number assigned by the network manager; it must conform to the following format:

```
area-number.node-number
```

where:

area-number — is an integer in the 1-63 range. Area numbers must be unique within the network. If an area number is not specified, the area number defaults to the local area.

node-number is an integer in the 1-1023 range. Node numbers must be unique to the specific network area.

Node Name is a string of up to six alphanumeric characters (letters and numbers), and at least one character in the name must be alphabetic. Only one name is allowed per node. Node names are not case-sensitive; lowercase letters are converted to uppercase. However, since IRIX is case sensitive, it is easier to manage 4DDN nodes when all letters in their names are lowercase.

2.3.3 4DDN Software Components

A network connection requires communication between two processes: a *client* process running in the node where the request is entered, and a *server* process running in the responding node. Unless each counterpart is running, logical links cannot be established between network nodes.

The master server process, *dnserver*, is responsible for network connections to 4DDN nodes. *dnserver* is a continuously running IRIX process that invokes other servers to handle requests for service. For each server *dnserver* invokes, a corresponding client process resides on other DECnet nodes. *dnserver* calls on these other 4DDN servers to handle incoming requests:

FAL The File Access Listener (FAL) server handles incoming requests that require access to files on its node. It provides the local file system with network file access functions, such as file copies and directory listings. FAL uses Data Access Protocol (DAP) to communicate with processes on other nodes. User commands such as *dnls* and *dncp* require the services of FAL.

NML The Network Management Listener (NML) server supports NCP, which handles network management tasks. Together with NCP, NML enables users to determine the status of a network, zero lines and circuit counters, and perform other network control operations. NML executes Network Information and Control Exchange (NICE) protocol messages that it receives from NCP. The responses to the protocol messages are returned to the appropriate NCP for display.

sethostd The virtual terminal server *sethostd* enables a user on one node to log into a remote node on the network. The *SET HOST* client and *sethostd* server use the CTERM protocol to communicate during virtual terminal sessions.

dnMaild The mail server, *dnMaild*, handles incoming mail deliveries to the 4DDN node where it runs. It passes the mail messages it receives from remote nodes to the local delivery service. *dnMaild* uses a subset of the VMS MAIL protocol.

2.4 If You Are New to IRIX

IRIX is the operating system supplied with your IRIS workstation. It is a version of the UNIX System V operating system (originally developed at Bell Laboratories) and also includes some BSD UNIX enhancements (features developed by the University of California at Berkeley). The IRIX *kernel* handles system-level tasks such as managing hardware, and the *shell* is the command interpreter for user entries.

The IRIS workstation offers two user interfaces to IRIX: a visual interface called the *WorkSpace*™, which you use by selecting icons; and a shell interface, which you use by typing commands at the IRIX prompt. 4DDN commands must be used from an IRIX shell. (See Appendix A of the *Owner's Guide* for your IRIS for more introductory information on IRIX.)

2.4.1 Opening an IRIX Shell

To open an IRIX shell, follow this procedure:

1. Click on the *Tools* toolbox.
2. Select *Shell* from the menu.
3. Position the shell and click it in place.

2.4.2 Using Command Equivalents

Table 2-1 shows you command equivalents in the VMS, IRIX, and 4DDN environments. IRIX, like other UNIX systems, is case sensitive; your commands must be entered in lower or upper case, as shown.

VMS	IRIX	4DDN
COPY	cp	dncp
DELETE	rm	dnrm
DIRECTORY	ls	dnls
RENAME	mv	dnmv
PRINT	lp	dnlp
SET HOST	rlogin	sethost
SUBMIT	sh	dnex
TYPE	more	(none)
MAIL	mail, Mail	dnMail

Table 2-1 4DDN Command Equivalents

Chapter 3

Using Network File Access Routines

This chapter explains how to use Network File Access Routines (NFARS) in network applications. It provides a functional description of the major routines, explains how NFARS perform wildcard expansion and error handling, and describes how to include and link NFARS routines in C programs. It also provides a reference guide describing each routine you can use for creating network applications.

NFARS provide a programming interface to file systems on remote nodes running 4DDN or DECnet Phase IV. NFARS use DNA's Data Access Protocol for file transfers. The interface to the DAP protocol was designed to closely emulate IRIX basic I/O system calls.

3.1 The NFARS Library

The object-code library of NFARS included with 4DDN software provides the elements for building network applications. The Remote File Access Service (RFAS) programs *dnls*, *dnmv*, *dnrm*, *dncp*, *dnex*, and *dnlp* were built with the NFARS facilities described in this chapter.

NFARS provide the programming interface to RFAS on the DECnet network. NFARS provide these functions:

- Open or create a remote file
- Write data to the currently open file
- Read data from the currently open file

- Append data to the end of a file
- Print a file after it is closed
- Submit a remote command file for execution
- Delete a remote file
- Rename a remote file
- Display the cause of an error encountered in an NFARS call

The NFARS are supplied in an archived object library called */usr/lib/libdn.a*. To link user programs that use the NFARS routines to the NFARS object library, use the *cc*(1) command shown below:

```
cc example.c -o example -ldn
```

Appendix E of this guide contains man pages for the NFARS routines that 4DDN supports. It contains reference information on these NFARS:

net_open *net_delete*
net_read *net_execute*
net_write *net_rename*
net_close *net_find*
net_perror *net_fnext*
net_strerror *net_fstop*

3.2 Accessing Remote Files

The *net_open* function initiates remote file access. Depending on the value of the option argument, it can open an existing remote file for reading or create a new remote file for writing. The *net_open* function returns an integer value that can be either a network file descriptor or a negative value to signal a failure. By default, files opened with *net_open* are in record format; format conversions are performed by the **net_read** or **net_write** function during the transfer of the file.

NFARS uses a network file descriptor to reference an open network stream. The descriptor is a small, non-negative integer value similar to the IRIX local file descriptor returned by the *open* and *create* system calls. However, DO NOT use the local file descriptor instead of the descriptor returned from the network.

3.2.1 Reading and Writing to Opened Files

The *net_read* and *net_write* functions transfer data to or from remote files. Use *net_read* and *net_write* in the same way as you use the IRIX basic I/O *read* and *write* functions. A remote file opened for reading supports only the *net_read* function, whereas a file created or opened for writing supports only the *net_write* function. These restrictions are imposed by the subset of the DAP protocol that implements the NFARS, and by an IRIX restriction on files opened by the *net_open* function.

Do not intermix calls to both *net_read* and *net_write* on the same file descriptor. Unlike the IRIX file system where a file can be opened for both read and write operations, network files cannot.

If you open a file without the RFM_VERBATIM option, data access to the file is line oriented. Data from a call to *net_read* is one or more lines of text. A line is a collection of characters, including control characters, terminated by the local line terminator. In IRIX, the local line terminator is the line-feed known as *newline*.

When *net_read* reads a data block fragment that is not delivered to the user, it stores the fragment. It then combines the fragment with subsequent incoming records and delivers it with them.

net_read provides a sequence of lines resembling the source file to its caller. *net_write* writes the data it receives to the remote file one line at a time. Each line of source data goes into a separate record for transport. Since DAP transport is heavily record oriented, it limits some NFARS capabilities, such as the ability to select file record formats and to seek.

When *net_write* receives line fragments, it combines them with subsequent *net_writes*. Thus, *net_write* can handle partial lines. When it writes to a file that was opened for writing, the resulting file is line oriented and suitable as a user text file on the remote system. On VAX/VMS systems, the file is in Variable Record Format with Carriage Return Carriage Control. On IRIX, the file is simply lines of text.

If you specify the RFM_VERBATIM option to open a file, the line orientation of NFARS is disengaged. Data from a call to *net_read* is read without regard to structure or line orientation; it is merely a byte-for-byte read.

Consider the differences between IRIX and VAX/VMS record formats when writing programs that use the NFARS library. These differences are described under the topic "Converting File Record Formats" in Chapter 6 of the *4DDN User's Guide*.

3.2.2 Specifying Record Length

The default record length for file reads and writes is 512 bytes. However, you can specify a different record length for reads and writes with the external variable *usr_blk_len*. Using *usr_blk_len*, you can specify a record length up to 1024 bytes for read and write operations.

3.2.3 Closing Remote Files

When a network file descriptor is no longer needed, close it immediately with *net_close*. *net_close* releases the network resources the descriptor occupies. Although network files left open when an NFARS application exits (or is terminated) close automatically, use *net_close* to prevent loss of data on remote files that are open for writing when an application exits.

3.2.4 Deleting Remote Files

Use the *net_delete* function to delete remote files. Some remote nodes restrict the data to be deleted, or impose access limitations on files. In addition, some remote nodes are not capable of supporting *net_delete*.

net_delete is analogous to the *dnrm* program. However, *dnrm* supports wildcards, whereas *net_delete* generally does not (some systems are exceptions). See "Deleting Files with *dnrm*" in Chapter 5 of the *4DDN User's Guide* for more information on using *net_delete* in network applications.

The *net_delete* function has no formal relationship with the *net_open* function. Do not use *net_delete* on an opened file.

3.2.5 Executing Remote Files

You can locate remote command files and submit them for execution on the remote node with a call to the *net_execute* function. Some nodes restrict the files that can be executed. In addition, some nodes apply access control rules to the request. For further information on remote file execution, see Chapter 7, "Executing Remote Files," in the *4DDN User's Guide*.

3.2.6 Renaming Remote Files

Calls to *net_rename* rename remote files. The specified files must be on the same node. On some nodes, there are restrictions on renaming files across devices, or when there are differences in access control information. Refer to the section "Moving Files with *dnmv*" in Chapter 5 of the *4DDN User's Guide* for more information. The *dnmv* command supports wildcards, whereas the *net_rename* function does not.

The *net_rename* function has no formal relationship with the *net_open* function. Do not use *net_rename* on an opened file.

3.3 The Wildcard Expansion Utility

The wildcard name expansion facility identifies files that match specified naming requirements. The file specification being expanded may or may not contain wildcard characters. The *net_find* function initiates the search, and the *net_fnext* function finds and retrieves subsequent names. The *net_fstop* function terminates or aborts the wildcard list.

Names resolved by the name expansion facility are stored in its three components: filename, directory, and volume. You can also request that the attributes of the file be collected.

Use this C preprocessor directive to include the attributes structure definition:

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

3.3.1 The File Attributes Structure

If you request file attributes, the attributes are placed in a *file_attr* structure. The values within the structure are filled in with information that the caller of *net_fnext* requests. If date information is necessary, specify RFA_DATE as an attribute when *net_find is called. The structure below applies to VAX/VMS systems:*

```
typedef struct file_att {
    /* Date-time information in UNIX time format  */
    long        fa_cdt;         /* file creation */
    long        fa_rdt;         /* last modification */
    long        fa_adt;         /* last access */

    /* Owner identification */
    char        *fa_owner;      /* user code of file owner */

    /* File protection */
    unsigned char fa_powner;    /* file owner */
    unsigned char fa_pgroup;    /* group */
    unsigned char fa_pworld;    /* world */
    unsigned char fa_psystem;   /* system */

    /* Attributes information */
    short fa_org;     /* file organization */
    short fa_bsz;     /* # of bits per byte */
    short fa_bls;     /* # of bytes per block */
    short fa_rfm;     /* format of records */
    short fa_mrs;     /* length of each file record (bytes) */
    short fa_ffb;     /* first free byte in EOF block */
    short fa_rat;     /* attribute of individual records */
    long  fa_alg;     /* allocation quantity (blocks) */
    long  fa_ebk;     /* # of last block */
} File_attr;
```

3.3.2 File Time Attributes

The times are stored in a format that is identical to the return of a call to the IRIX *time*(2) system call. For this reason, it might require a call to *ctime(3C)* to print the date in a pleasing local format.

3.3.3 File Protection Attributes

The *fa_powner* element is a pointer to a string that contains a sequence of characters, generated by the remote system, that describes the owner of the file. On VMS the sequence is a set of numbers, whereas on UNIX it is a pair of names from the */etc/passwd* file.

The protection attributes are grouped four bits to the attribute byte describing the access rights of a particular entity. File protection is similar to IRIX with the addition of the "system" entity, which is similar to the operator concept on some systems. You can test each byte of permissions with the masks listed below. If the bits are set, they have the following meaning:

`P_READ`	file may be read
`P_WRITE`	file may be written
`P_EXECUTE`	file may be executed
`P_DELETE`	file may be deleted

The meanings are similar to the meanings on IRIX, except that the delete concept is associated with the file rather than the directory.

3.3.4 Other File Attributes

The *fa_org* element contains a code that can be tested to determine file organization. File organizations are specific to VMS. Only sequential files can be transferred using NFARS. Only one of these values is possible. The test symbols are shown below.

`ORG_SEQUENTIAL`	the file is sequential in structure
`ORG_RELATIVE`	the file is relative in structure
`ORG_INDEXED`	the file is indexed in structure

The *fa_bsz* element contains a number that is the number of bits per byte. This is normally 8. It can be considered 8 if it is set to 0. The *fa_bls* element is the number of bytes per block. Its normal value is 512 bytes. If the value in this element is 0, the number of bytes per block can be considered 512.

The *fa_rfm* element contains a code that you can test to determine the record format of the file. Only one of the following symbols can be true:

```
RFM_UNDEFINED    the record format is undefined
RFM_FIXED        fixed length records
RFM_VARIABLE     variable length records
RFM_VARFC        variable with fixed control records
RFM_STREAM       stream format (basically non-record)
RFM_STREAMLF     stream-LF format (lines ending with \n)
RFM_STREAMCR     stream-CR format (lines ending with \r)
```

The *fa_mrs* element contains the maximum number of bytes per record. Its default value is 512 bytes. If the entry is 0, you can assume it is 512 bytes.

The *fa_ffb* element contains the number of the first free byte in the last block. You can use *fa_ffb* to compute the number of bytes in the file with this equation:

*size = (ebk - 1) * bls + ffb*

The *fa_rat* element contains some additional record attributes. You can combine the symbols that follow in a bitwise manner using the OR operator. This field is extremely VMS oriented.

RAT_FORTRAN	Record contains FORTRAN carriage control
RAT_CR	record has an implied LF/CR envelope
RAT_PRN	print file carriage control is in fixed part of VFC
RAT_BLK	records to not span blocks
RAT_EMB	embedded format control
RAT_LSA	line sequenced – ASCII number in fixed part of VFC
RAT_MACY	RSX-11 compatible format

The *fa_alq* element contains the number of blocks allocated to contain the file. This number may be larger than the number of bytes actually involved.

The *fa_ebk* element contains the number of blocks actually involved in the file. More precisely, *fa_ebk* contains the number of the last block.

3.4 NFARS Header Files

C programs using NFARS should add this line before any references to NFARS appear in the program:

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

You can also include the header file *nferror.h* in your programs. *nferror.h* contains the symbolic constants corresponding to the error codes. The comments next to the values in *nferror.h* are similar to the messages printed by *net_perror*. Include this file to check the values of *nfars_errno*. Add this line to include *nferror.h*:

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

You must include the *nfattr.h* file for wildcard processing and to create new files with *net_open*. *nfattr.h* determines the attributes of files discovered with the wildcard expansion system, based on *net_find* and *net_fnext*. Add this line to include *nfattr.h*:

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

For C++ programs, include the header file *nfarsbasic.h*. It contains the function prototypes your program needs:

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

3.5 NFARS Errors

The value returned by an NFARS function always indicates whether it succeeded or failed. In the event of a failure, the external variable *nfars_errno* is set to a code identifying the cause of the failure.

The *net_perror* function prints a descriptive message based on the *nfars_errno* error code. The message contains an NFARS error message listed in Appendix A. The NFARS error message can be preceded by a user-defined string identifying the application program that generated the error.

The *net_strerror* function returns a string of descriptive text, based on the *nfars_errno* error code. Use *net_strerror* to vary the format of an error message from those produced by *net_perror*.

Chapter 4

Using Task-to-Task Communication

This chapter describes task-to-task communications, the process by which 4DDN programs exchange data over network connections. It explains how network applications establish, maintain, and terminate logical links; and it provides a programming reference to IRIX system calls that can be included in user-written applications requiring the task-to-task communication service.

The 4DDN task-to-task communication service enables the exchange of data between processes running on different nodes. It provides reliable, effective communication between tasks, regardless of their location on the network.

The 4DDN virtual terminal service, remote file access, and network management utility use task-to-task communications. You can also use the task-to-task programmatic interface to develop other network applications. The advantage of this program interface is that it protects users from lower-level network details, such as topology, transmission sharing techniques, or communication linkages.

4.1 Client and Server Processes

Task-to-task communication involves two processes, usually (but not necessarily) running on different nodes, and communicating over a logical link. To establish a logical link between two processes, one process must notify the other that it wishes to communicate. The process that requests the connection is called the *client*, and the responding process is called the *server*.

First the server informs the network software that it wants to be a server. The client supplies access control information that the server uses to evaluate the incoming request. The server determines whether to accept or reject the connection request, based on the access control information.

If the request is accepted, a logical link is established. Once the logical link is established, either process can send or receive data; no distinction exists between a client and a server once the link is established.

4.2 Establishing Logical Links

A logical link is a temporary software data path established between two communicating processes in a network. A process can set up more than one logical link. For example, it can set up multiple logical links to communicate with different processes. It can also set up several logical links to communicate with the same process if separate data streams are intended for different purposes. The application using a logical link determines how the link is established.

Logical links are established through the standard IRIX I/O system calls *open*(), *close*(), *read*(), and *write*(), as well as *ioctl*() system calls. The logical link between two processes is comparable to an I/O channel over which both processes can send and receive data. To establish a logical link and transmit data across it, applications must issue calls to the 4DDN logical link device.

4.2.1 Activating the Logical Link Device

To establish a logical link, the client first issues an *open*() to activate the logical link device, */dev/dn_ll*. The logical link device is a virtual I/O device responsible for controlling logical links. The *open*() returns a file descriptor for the logical link. This file descriptor must be used in all subsequent task-

to-task calls over this logical link. Applications must specify an *open*() for each logical link to be established.

Once the client obtains a file descriptor, it makes the connect request by issuing an *ioctl* request and passing access control information to the server. This information identifies the server process and the client. The server process must be available for connection at the time the request is made.

After receiving the request, the server can either accept or reject the connection. A logical link is established only after the server accepts the logical link request.

The *ioctl*() issued by the client returns the status from the server. If the link is rejected, the status indicates a failure and the client must free the file descriptor by issuing a *close*. If the link is accepted, the status indicates success. The logical link is then established and the exchange of data can take place.

4.2.2 Registering 4DDN Processes As Servers

You can code a 4DDN server process to start automatically or explicitly. *Automatically started* servers are those started by *dnserver*, a process that registers itself for the purpose of accepting requests for objects not previously registered as explicitly started servers. *Explicitly started* servers are server processes, started by a command line entry by the user or a shell script, that register to receive a specific or explicit object.

Automatically started servers offers these advantages:

- The server does not need to be pre-started in anticipation of a connection.
- Access control is enforced.

Using Explicitly Started Servers

A server process that is started explicitly by a user or through a shell script must open a logical link and register as a server, either by name or number. To open a link, the server process must first issue an *open*() to activate the logical link device and receive a file descriptor. This file descriptor is used in subsequent task-to-task commands over this logical link.

The server process must then register itself as a server to 4DDN networking software by specifying an object type number or task name to which it will respond. It does this by issuing an *ioctl*() request. Until a 4DDN process is a registered server, no client process can initiate logical connections to it.

After it issues the *ioctl* to be registered, the server process issues an *ioctl*() to wait for a connection request and access control information transmitted by a client. When it receives a connection request from a client process, the server process can use the control information it receives from the client to evaluate the request. The server process then issues an *ioctl* request to accept or reject the link.

When an *ioctl*() to accept a link completes successfully, the link is established and ready for the exchange of data between processes. When the *ioctl*() to reject the link completes successfully, the link must be closed (*close()*) by both processes.

In order to accept new logical links, the server process must re-register itself as a server by repeating the registering process.

Using Automatically Started Servers

dnserver, a continuously running IRIX process, can start servers automatically. Whenever 4DDN receives a connection for an object that is not currently registered by name or number, the connection is given to *dnserver*. *dnserver* performs these actions:

1. *dnserver* checks to see if a username and password are specified in the OpenBlock. If they are specified but not valid according to the */etc/passwd* file or Network Information Services (NIS) password database, the connection is rejected.

2. *dnserver* checks */usr/etc/dn/servers.reg* for an entry for the requested object, by object name or object number (most servers are registered by number). The number 0 is not valid and means that the object is known by name only. The *servers.reg* file consists of entries in this form:

```
obj-number  obj-name  path

17          FAL       /usr/etc/dn/fal
```

3. If no entry is found and the connection was by name, then the login directory of the user specified in *username* is searched for a runnable file named *objectname*.

4. If no server can be found, the link is rejected; if the server is found, the *dnserver* forks a process with group and user privileges associated with the user name and runs the server process. The working directory for the process is the login directory of the user.

   The server process starts with the logical link opened but not yet accepted or rejected, using these file-descriptors:

`0 (stdin)`	The logical link (read only)
`1 (stdout)`	The logical link (write only)
`2 (stderr)`	A log file opened by dnserver
`argv[1]`	The logical link (read and write)

   Note: Writing to *stdout* for automatically started servers writes to the logical link rather than the console.

   dnserver starts the process specified by the path in */usr/etc/dn/servers.reg* or the path in the users's login directory, using a single argument. That argument is the file descriptor number of the logical link. This call starts the process:

   ```
   execl ("/usr/etc/dn/fal","fal","4",0);
   ```

   If main is defined by:

   ```
   main(argc,argv)
       int argc;
       char *argv[];
   ```

   The result is:

   ```
   argc = 2;
   argv[0] = "fal";
   argv[1] = "4";
   ```

   The newly-forked server process may do another *SES_GET_AI ioctl* to get the OpenBlock if desired. The server must perform a *SES_ACCEPT ioctl* before the logical link is really active or a *SES_REJECT* to explicitly reject it.

A server can be coded to run either explicitly or automatically as follows:

```
#include <ctype.h>

if (argc == 2 && isdigit(argv[1][0])) {
   /* automatic start */
} else {
   /* explicit start */
}
```

4.2.3 Transmitting and Receiving Data

While a logical link is in effect, a process can exchange data across the logical link through a series of calls using the assigned file descriptor. It can also transmit interrupt data, special high-priority information that is transmitted immediately.

The process uses *read*() to receive data and *write*() to send data. It transmits and receives interrupt data through *ioctl* requests.

By means of an *ioctl* request, 4DDN allows a process to specify the I/O data format as either record or stream:

- In stream format, data is passed across the network in a buffer. There is no indication whether the buffer contains a complete message. A process receives only the number of bytes sent in the buffer.
- In record format, a process uses a structure to send and receive data. This structure contains the address of the data buffer and a special status field that indicates whether the buffer contains the beginning, middle, or end of a message, or a complete message. Record format allows applications to perform their own data segmentation.

4DDN allows a process to specify the input mode for *read*() operations as either blocking and non-blocking:

- With a blocking read, a process waits until the available data has been written into a user-supplied buffer.
- With a non-blocking read, the number of bytes read is returned or the process is notified that data is unavailable. Optionally, a special signal may be registered to notify the process when data becomes available.

4.2.4 Terminating the Logical Link

At any time, either process can terminate the logical link and, optionally, transmit data explaining the reason for termination. If no optional data is to be sent, *close*() is sufficient to terminate the logical link. *close*() automatically disconnects the link.

If optional data is to be sent to the remote process, a disconnect or abort *ioctl()* request is first issued, followed by *close*(). Disconnecting a logical link guarantees that all data that has been transmitted is delivered before the link is closed.

Note: Successful disconnect indicates that the remote node has received, but not necessarily processed, all transmitted data. An application-level acknowledgment is necessary for assurance.

Aborting a link means that outstanding data is discarded before the link is terminated. The link is terminated whether or not the remote node has acknowledged receipt of previously transmitted data.

Processes should normally disconnect, not abort, a link. A process may choose to abort in response to an error condition.

After the logical link is closed (*close()*), the server process must re-register itself as a server in order to be a server for another logical link connection.

Chapter 5

Task-To-Task Communication Programming Reference

This chapter is a programming reference to the system calls used in 4DDN task-to-task communication. It provides a functional description of each call, illustrates call syntax, describes data structures used in the call, and explains call results. You can include these calls in user-written network applications that require task-to-task communications.

5.1 Header Files and Libraries

The <*dn/defs.h*> header file contains constants and data structure definitions used in the 4DDN task-to-task communication function calls. The <*dn/defs.h*> file should be included when you write networking applications using the 4DDN C program interface.

The <*fcntl.h*> file is a standard IRIX header file that should be included in your source files. The <*fcntl.h*> file is needed only for the *open*() call. It contains the definitions for the different open modes (read only, write only, read and write).

Programs that call the *dn_perror* library routine should link with the library */usr/lib/libdn.a*:

```
cc  example.c -o example -ldn
```

5.2 The *errno* External Variable

If a 4DDN task-to-task function call returns a value of –1, it means the function did not execute successfully. An unsuccessful completions sets the external variable *errno* to the appropriate error code. *errno* is not changed under any other circumstances. (Appendix B contains information on 4DDN error codes and recommended actions.)

IRIX can generate other error codes. Refer to the *intro*(2) man page for IRIX errors.

Calls to the library routine *dn_perror* print a descriptive message based on the *errno* value. The message can include a user-defined string that identifies the application program that generated the message (see "Printing Error Messages" later in this chapter).

5.3 The Calling Sequence

Table 5-1 illustrates the sequence of system calls that client and server processes issue in 4DDN task-to-task communications. The lower region of the table lists calls that either the client or the server process can issue after the logical link is established.

Client	Server
	open() logical link device create logical link fd. *ioctl*() to register as a server (SES_NUM_SERVER or SES_NAME_SERVER) *ioctl*() to wait for connect request (SES_GET_AI)
open() logical link device create logical link fd *ioctl*() to request logical link and pass access control information (SES_LINK_ACCESS)	server waits
client waits for response (accept or reject) from server	receive access control info *ioctl*() to accept link request (SES_ACCEPT) or *ioctl*() to reject link request (SES_REJECT)
Logical link is established.	
ioctl() to define I/O data format and input mode (SES_IO_TYPE) *ioctl*() (SES_GET_PROXY) to determine proxy login data field *ioctl*() to determine the maximum transmit buffer size *read*() and *write*() for normal data *ioctl*() to transmit interrupt data (XMIT_INTERRUPT) *ioctl*() to receive interrupt data (ACCEPT_INT and RCV_INTERRUPT) *ioctl*() for logical link status (SES_STATUS) *ioctl*() to disconnect logical link (SES_DISCONNECT) *ioctl*() to abort logical link (SES_ABORT) *close*() the logical link	

Table 5-1 Sequence of Task-to-Task Communication Commands

5.4 Opening a Logical Link Device

The *open*() call opens the logical link device and returns a unique logical link file descriptor used in subsequent calls associated with this logical link. This is the first step in establishing a logical link. It must be issued by the server and the client for each logical link desired.

Call Usage

```
#include <fcntl.h>
#include <dn/defs.h>

int open_mode;
int link;
link = open(DN_LINK, open_mode);
```

Arguments

link	Logical link file descriptor. *link* is assigned –1 if an error occurred; otherwise it receives the logical link identifier.
DN_LINK	Logical link device name defined in <*dn/defs.h*>.
open_mode	Indicates the open mode for the logical link. These codes are defined in <*fcntl.h*>.

Results

On success, the *open*() call returns a unique logical link file descriptor used for all subsequent I/O function calls pertaining to the associated logical link. If an error occurs, a –1 is returned, and *errno* is set to the appropriate error code. (See Appendix B for recommended actions.)

Error Codes

LOCAL_RESOUR	Local node does not have resources for the link.

5.5 Requesting a Logical Link

The *SES_LINK_ACCESS ioctl* call, issued by a client, transmits information identifying the client and the server to which it wants to connect. The information is passed in a data structure called an OpenBlock.

Call Usage

```
#include <dn/defs.h>
int       link;
OpenBlock ob;
int       ret;
ret = ioctl(link, SES_LINK_ACCESS, &ob);
```

Arguments

OpenBlock	*typedef* defined in <*dn/defs.h*> See the explanation of OpenBlock, below.
link	Logical link file descriptor.
SES_LINK_ACCESS	*ioctl* function code.
ob	Structure containing the information transmitted to the server by the client. See the explanation of the OpenBlock, below.
ret	Value returned by *ioctl*().

Results

On success, this *ioctl* call returns 0, and a logical link is established with the server program. If an error occurs or the server program rejects the logical link request, *ioctl*() returns –1 and the *errno* is set to the appropriate error code. (See Appendix B for recommended actions.) If the server sends data in addition to the connect acceptance or rejection, this data is placed into the *op_opt_data* field of the OpenBlock.

The open_block Structure

```
typedef struct image_16 {
     char im_length;
     char im_data[DATA_LEN];
     char im_rsvd;
} Image16;

/*  Open Block Data
 *
 *   The open_block structure contains the access control
 *   information necessary for establishing a logical link.
 *   This structure must be used in the SES_LINK_ACCESS and
 *   SES_SET_AI_IOCTL function calls.
 */

#define   NODE_LEN       7
#define   TASK_LEN      17
#define   USER_LEN      32
#define   ACCT_LEN      16
#define   PASS_LEN      32

typedef struct open_block {
    short   op_object_nbr;            /* Object number       */
    char    op_node_name[NODE_LEN];   /* Node name           */
    char    op_task_name[TASK_LEN];   /* Task name           */
    char    op_userid[USER_LEN];      /* User name           */
    char    op_account[ACCT_LEN];     /* User account number */
    char    op_password[PASS_LEN];    /* User password       */
    Image16 op_opt_data;              /* Optional data       */
    unsigned short op_proxy_uid;      /* UID for proxy-login */
} OpenBlock;
```

Arguments

op_node_name — is the system name or number of the server system for the connection. It is a null-terminated string. Legal values are:

1. A character string, the name of the remote system, must consist of at least one alphabetic character. It should not end with a colon (":").

2. A character string, in the form *aa.nnn*, representing the system number of the remote system; *aa* is the area number and *nnn* is the system number. If no period is detected in the string, the value is treated as a system number in the same area as the local system.

3. A null string indicates a connection to an object on the local system.

op_task_name — is the name of the server program. It is a null-terminated string. See "Server Addressing Rules," below.

op_object_number — is a binary value of the server object number on the remote system. Legal values range between 1 and 255. See "Server Addressing Rules," below.

op_userid — is the name used by the remote system to identify the connections client (required by some remote systems). It is a null-terminated string.

op_account — is the accounting information (if required) used by the remote system in allowing the connection. It is a null-terminated string.

op_password — is the password (if required) used by the remote system in accepting the connection. It is a null-terminated string.

op_opt_data — is connection-dependent optional data used by the remote application. *im_length* indicates the length of the data and should be set by the requesting program to send optional data to the server program. When a server program returns optional data with the acceptance or rejection, *im_length* is set to the number of bytes received. It should be set to 0 if no data is desired.

im_data — contains the data. *im_rsvd* must be binary zero.

Server Addressing Rules

1. To address a server program, a logical link request specifies either a task name or object number, but not both.

2. To address a server program by task name, *op_object_nbr* must be set to 0 and *op_task_name* set to an ASCII name.

3. To address a target program by object number, *op_object_nbr* must be an integer between 1 and 255 and *op_object_name* must be set to null. User-defined objects must be integers between 128 and 255. Numbers between 1 and 127 are reserved for use by privileged tasks.

Error Codes

ACCESS_CONT	Remote system or program rejected access information
ALREADY	Logical link file descriptor already in use
BAD_OBJECT	Specified remote object does not exist
BY_OBJECT	Local or remote program has closed the link
LOCAL_RESOUR	Local node does not have resources for the link
LOCAL_SHUT	Local node is not accepting new links
MANAGEMENT	Link was disconnected by network
NET_RESOUR	Insufficient network resources
NODE_DOWN	Remote system is not accepting new links
NODE_FAILED	Remote system failed to respond
NODE_NAME	Unrecognized system name
NODE_UNREACH	Remote system is currently inactive
OBJ_BUSY	Insufficient resources at remote system
OBJ_NAME	Specified task name invalid
OUT_OF_SPACE	Temporarily out of kernel buffers
REMOTE_ABORT	Link was aborted by remote program
UNKNOWN_ERR	Status code sent by remote system is undefined at local system

5.6 Registering the Server Program

The SES_NUM_SERVER and SES_NAME_SERVER *ioctl* calls are used when a program wants to register itself as a server. Registration assigns server programs an object number or object name that other programs use to identify the servers.

Call Usage

```
     int    link;
     int    ret;
     short  object_number;
     ret = ioctl(link, SES_NUM_SERVER, &object_number);
or
     char  task_name[TASK_LEN];
     ret = ioctl(link, SES_NAME_SERVER, task_name);
```

Arguments

link	Logical link file descriptor.
SES_NUM_SERVER	Appropriate *ioctl* request codes.
SES_NAME_SERVER	Appropriate *ioctl* request codes.
object_number	Task or object number of the server program. User-defined object numbers must be integers between 128 and 255.
task_name	Null-terminated ASCII string specifying a task or object name of server program.
ret	Value returned by *ioctl*().

Results

SES_NUM_SERVER and *SES_NAME_SERVER* return 0 when the server is registered correctly. The call returns –1 if an error occurs, and the external variable *errno* is set to indicate the appropriate error code. (See Appendix B for recommended actions.)

Error Codes

ALREADY	Logical link file descriptor already in use
LOCAL_RESOUR	Local system does not have resources for the link
LOCAL_SHUT	Local system is not accepting new links
OBJ_NAME	Specified task name is invalid
OUT_OF_SPACE	Temporarily out of kernel buffers
UNKNOWN_ERR	Status code sent by remote system is undefined at local node

5.7 Receiving Access Control Information

The *SES_GET_AI ioctl* call, used to receive access control information, is issued by the server program after it has registered itself. If a logical link request has already been received before this call is issued, then the OpenBlock structure is returned with access control information. If this call is issued before a request has been received, this call blocks and waits until a request is received.

The *SES_GET_AI_NB ioctl* call is used to poll for an incoming connection. It immediately returns a *NOT_CONNECTED* error if no connect request is pending. Otherwise, it returns success and fills in the open block structure.

Call Usage

```
int       link;
OpenBlock ob;
int       ret;
ret = ioctl(link, SES_GET_AI, &ob);
```

Arguments

OpenBlock	typedef defined in <*dn/defs.h*>.
link	Logical link file descriptor.
SES_GET_AI	*ioctl* request code.
SES_GET_AI_NB	*ioctl* non_blocking call.
ob	OpenBlock structure to receive the access control information transmitted by the client to the server.
ret	Value returned by *ioctl*().

Results

When a logical link request is received, the content of the client's OpenBlock is copied into the server's allocated OpenBlock, and the call returns 0. If an error occurs, *ioctl*() returns –1, and the external variable *errno* is set to the appropriate error code. (See Appendix B for recommended actions.)

Error Codes

BY_OBJECT	The remote user disconnected the link
LOCAL_SHUT	The network software was shut off while waiting for the connect request
NOT CONNECTED	The logical link does not exist
OUT_OF_SPACE	Temporarily out of kernel buffers
REMOTE_ABORT	The remote user aborted the link
UNKNOWN_ERR	Error code sent by remote system undefined at local system

5.8 Accepting or Rejecting a Logical Link Request

The server program has the option of accepting or rejecting the logical link request through the *ioctl* calls *SES_ACCEPT* and *SES_REJECT*.

Call Usage

```
typedef struct session_data {
    short   sd_reason;
    Image16 sd_data;
    char    sd_rsvd[4];
} SessionData;

int         link;
SessionData sd;
int         ret;
ret = ioctl(link, SES_ACCEPT, &sd);

             Or

ret = ioctl(link, SES_REJECT, &sd);
```

Arguments

Image16	*typedef* defined in <*dn/defs.h*>.
SessionData	*typedef* defined in <*dn/defs.h*>.
link	Logical link file descriptor.
SES_ACCEPT	*ioctl* request code.
SES_REJECT	*ioctl* request code.

sd	Structure that contains the acceptance or rejection data to be sent to the client. The value of *sd_data.im_data* is application dependent. If no data is to be sent, the *sd_data.im_length* field must be set to 0. *sd_reason* contains a user-defined code sent by the *SES_ACCEPT ioctl* call. *sd_data.im_rsvd* and *sd_rsvd* must be binary zero.
ret	Value returned by *ioctl*().

Description

The acceptance or rejection data passed in the *sd.sd_data* field is sent to the client. The client receives this data in the optional data field (*op_opt_data*) of the OpenBlock structure that was used to request a logical link. Once a logical link request is rejected by the server program, the logical link must be explicitly *close*()'d by the rejecting program in order to free the descriptor for future use. If the logical link is not closed, it remains open and unavailable for any connection requests. To accept a new logical link connection requests, the server program must re-open the logical link device and re-register itself as a server.

Results

On successful completion, *SES_ACCEPT* returns 0, and the link is accepted and prepared for the exchange of data. If an error occurs, SES_ACCEPT returns –1 and the external variable *errno* is set to the appropriate error code. (See Appendix B for recommended actions.)

On successful completion, *SES_REJECT* returns 0 and the link is rejected. (To terminate the logical link, refer to ''Closing the Logical Link'' below.) If an error occurs, SES_REJECT returns –1, and the external variable *errno* is set to the appropriate error code. (See Appendix B for recommended actions.)

If an error occurs for either SES_ACCEPT or SES_REJECT, the logical link file descriptor must be released by a *close()*.

Error Codes

BY_OBJECT	The remote user disconnected the link
LOCAL_SHUT	The local node has been shut down
OUT_OF_SPACE	Temporarily out of kernel buffers
REMOTE_ABORT	The remote user aborted the link
UNKNOWN_ERR	Error code sent by remote node is undefined at local node

5.9 Using the Proxy Ioctl Function

The *proxy ioctl* function allows an application to request the proxy login data field of an incoming connect request. It can be used either as an alternative or as an adjunct to user name and password information. Proxy *ioctl* returns the proxy login field as an Image16 data structure.

The example below illustrates the *proxy ioctl* function.

```
SessionData Sd;
Image16 Proxy;
            .
            :
/*Obtain access control information*/
     if ( ioctl ( fd,SES_GET_AI,&openblock ) <0 )
     {
          dn_perror("?Get Access Info Request Failed - ");
          close( fd );
          Exit( );
     }
/*Obtain proxy field*/
     if ( ioctl ( fd,SES_GET_PROXY,&Proxy ) <0 )
     {
          dn_perror("?Link Accept Failed - ");
          close( fd );
          Exit( );
     }
/*Now accept connection request*/
     if ( ioctl ( fd,SES_ACCEPT,&Sd ) <0 )
     {
          dn_perror("?Link Accept Failed - ");
          close( fd );
          Exit( );
     }
     /*Proxy structure now has proxy data from Connect*/
     /*Initiate Msg*/
```

5.10 Selecting the Data Format and I/O Mode

The *SES_IO_TYPE ioctl* function command is used to select the options for sending and receiving data before issuing a read or write. The options are: I/O data format; stream or record; and the I/O mode,blocking or non-blocking. The data format and I/O mode selected must be used on all subsequent *read*() or *write*() calls.

Note: This *ioctl* request is optional. If omitted, the defaults, stream format and blocking mode, are used.

Data Formats

In stream format, data is passed across the network in a buffer. There is no indication whether the buffer contains a complete or incomplete message. A program receives only the number of bytes sent in the buffer. Stream data format is the default.

In record format, a program uses a structure to send and receive data. This structure contains the address of the data buffer and a special status field that indicates if the buffer contains the beginning, the middle, the end of the message, or a complete message. Record format allows applications to perform their own data segmentation. One end of the link may run in stream format and the other in record format.

Blocking I/O Mode

In blocking I/O mode, a read function returns to the calling program, only after the available data written into a user-supplied buffer. In this mode, the user program is blocked until data becomes available on the link. blocking I/O mode is the default.

In blocking I/O mode, a write function blocks until the data in the user-supplied buffer is copied to a network buffer for transmission. If the link is flow-controlled, a write function does not return (blocks) until the remote program starts reading.

As memory for transmit and receive buffers becomes scarce, flow control is automatically activated by the network software. When flow control is activated, the receiving system notifies the transmitting system to stop sending data messages. After this occurs, the transmitting system must wait for a message from the receiving system before resuming transmission of data messages.

Non-blocking I/O Mode

Non-blocking input mode may be used by polling the logical link for data availability, or with a signal notification indicating that data is available.

In non-blocking input mode, a read returns immediately with one of these results:

1. The number of bytes received and written into the user-supplied data buffer.

2. –1 and the external variable *errno* set to NO_DATA_AVAIL indicating that no data is available.

For non-blocking I/O without signal notification, a program must poll with read commands to determine if data is available. For non-blocking I/O with signal notification, a program receives a signal when data is available for reading. The notification scheme requires you to register the signal with both IRIX and 4DDN, via *SES_IO_TYPE* (see *signal*(2) in the *IRIS-4D Programmer's Reference Manual*).

4DDN signals your process when data becomes available on the link. When all the existing data is received (through reads) and new data becomes available, your process is signaled again.

In non-blocking I/O mode, a write function returns immediately if the logical link is not flow-controlled. If the logical link is flow-controlled, a write function returns immediately with a –1, and the external variable *errno* is set to the error code *FLOW_CONTROL*.

Call Usage

```
typedef struct io_options {
    short io_record;
    short io_nonblocking;
    short io_rsvd[2];
    int   io_signo;
} IoOptions;

int       link;
IoOptions opt;
int       ret;
ret = ioctl(link, SES_IO_TYPE, &opt);
```

Arguments

Io_Options	*typedef* defined in <*dn/defs.h*>.
link	Logical link file descriptor.
SES_IO_TYPE	Appropriate *ioctl* function code.
opt.io_record	Indicates the data format: SES_IO_RECORD_MODE (default) SES_IO_STREAM_MODE.
opt.io_nonblocking	Indicates the input type: SES_IO_BLOCKING (default) SES_IO_NON_BLOCKING.
opt.io_rsvd	Must be binary zero.
opt.io_signo	Number to signal when data becomes available on the link. The signal must be registered with the *signal*(2) system call before this call is issued. This field is used only when the non-blocking I/O mode is chosen.
ret	Value returned by the *ioctl*().

Rules

1. If non-blocking I/O with signal notification is chosen, the signal must be registered with IRIX before this call is issued.

2. This call may be issued only once, before any I/O takes place on the logical link.

Results

Upon successful completion, the *ioctl SES_IO_TYPE* function call returns 0. If it returns –1, an error occurred and the external variable *errno* is set to the appropriate error code. (See Appendix B for recommended actions.)

Error Codes

BAD_COMMAND Invalid *ioctl* command

5.11 Determining the Maximum Transmit Buffer Size

A program can inquire about the maximum number of bytes allowed in a single write request or returned by a single read request by issuing the *SES_MAX_IO ioctl*().

Note: When using record-mode, I/O programs can send and receive messages that are longer than can be specified in a single write or read request.

Call Usage

```
long  length;
int   ret;
ret = ioctl(link, SES_MAX_IO, &length);
```

Arguments

link	Logical link file descriptor.
SES_MAX_IO	Appropriate *ioctl* function code.
length	Variable where the maximum transmit length (in bytes) is returned by *ioctl*().
ret	Value returned by *ioctl*().

Results

On success, the value 0 is returned and the maximum length allowed for single read and write is placed into the length field. If an error occurs, the value -1 is returned and the external variable *errno* is set to the appropriate error code.

5.12 Receiving Data across a Logical Link

Data can be received across a logical link in either stream or record format. The calls described below are used to receive data in each format.

5.12.1 Receiving Data in Stream Format

Use this call to receive data in streams format:

Call Usage

```
int    link;
char   *buf;
int    nbytes;
int    ret;
ret = read(link, buf, nbytes);
```

Arguments

link	Logical link file descriptor.
buf	Character buffer in which data from the link is to be placed.
nbytes	The number of bytes for the read request.
ret	The number of bytes read or –1.
NON_BLOCKING	If set and data is not available, the read returns a –1 and *errno* is set to NO_DATA_AVAILABLE.
	BLOCKING If set, the read blocks until data is available.

Results

The read function call attempts to read (receive) a message that is no longer than the value specified in the *nbytes* parameter.

On successful completion, a non-negative integer is returned indicating the number of byptes of data read. If an error occurs, a –1 is returned and *errno* is set to the appropriate error code.

Error Codes

BY_OBJECT	Local or remote program has closed the link
MANAGEMENT	Link was disconnected by network
NODE_FAILED	Remote node failed to respond
NOT_CONNECTED	Logical link does not exist
OUT_OF_SPACE	Temporarily out of kernel buffers
REMOTE_ABORT	Link was aborted by remote program
UNKNOWN_ERR	Error code sent by remote node is undefined at local node

5.12.2 Receiving Data in Record Format

This call receives data in record format:

Call Usage

```
typedef struct session_record {
    short sr_status;
    char  *sr_buffer;
    char  sr_reserved[6];
} SesRecord;

SesRecord sesrec;
int   link;
int   nbytes;
int   ret;
ret = read(link, &sesrec, nbytes);
```

Arguments

SesRecord	*typedef* defined in <*dn/defs.h*>.
link	Logical link file descriptor.
sesrec	Structure holding address of buffer in which data from the link is placed, and status information about this data. *sr_reserved* must be zero.
nbytes	Requested number of bytes to be read into of *sesrec.sr_buffer*. The *read()* function call attempts to read (receive) a message no longer than this value.
ret	The number of bytes read or –1.
	sesrec.sr_status Buffer allocated to store status of the data. On successful return, contains one of these values.

1. BEG_OF_MESSAGE
2. MID_OF_MESSAGE

3. END_OF_MESSAGE

sesrec.sr_buffer Buffer allocated to store the data to be read.

Results

On success, *read()* returns the number of bytes placed in the allocated buffer. The value of *sesrec.sr_reserved* must be 0. The value in the *sr_status* field of *SesRecord* informs the user if more data is available. If the status returned is BEG_OF_MESSAGE or MID_OF_MESSAGE, then more data is available.

If an error occurs, *read*() returns –1 and the external variable *errno* is set to the appropriate error code.

Example

Assume a remote program issues a *write()* with *sr_status* set to COMPLETE and nbytes = 512. If the local program issues a *read*() with nbytes = 100, then the whole message cannot be read with one *read*() call. The first *read*() returns 100 to ret and BEG_OF_MESSAGE to *sr_status*. In order to read the 512 bytes, several *read*()s must be issued (in a loop) until the END_OF_MESSAGE status is returned.

In this example, the second, third, fourth, and fifth *read*()s return a *MID_OF_MESSAGE* status. The sixth *read*() returns 12 to *ret* and *END_OF_MESSAGE* to *sr_status*.

Error Codes

BY_OBJECT	Local or remote program has closed the link
MANAGEMENT	Link was disconnected by network
NODE_FAILED	Remote node failed to respond
NOT_CONNECTED	Logical link does not exist
OUT_OF_SPACE	Temporarily out of kernel buffers
REMOTE_ABORT	Link was aborted by remote program
UNKNOWN_ERR	Error code sent by remote node is undefined at local node

5.13 Sending Data across a Logical Link

Data can be sent across a logical link in either stream or record format. The system calls described below are used to send data in each format.

5.13.1 Sending Data in Stream Format

In stream format, the *write* function causes the specified number of bytes to be transmitted from the given buffer as a complete data message.

Call Usage

```
int  link;
char *buf;
int  nbytes;
int  ret;
ret = write(link, buf, nbytes);
```

Arguments

link	Logical link file descriptor.
buf	Data buffer from which the data is taken.
nbytes	The length of the buffer (in bytes) to transmit. Length must be less than the value returned by the *ioctl* call *SES_MAX_IO*.
ret	The actual number of bytes that were sent.

Results

The *write*() call returns the number of bytes sent. If –1 is returned, an error has occurred, and the external variable *errno* is set to the appropriate error code. (See Appendix B for recommended actions.)

If the I/O mode is blocking, then the *write*() function blocks until the data buffer is copied to the kernel for transmission. If the I/O mode is non-

blocking, the *write*() function returns immediately. When the logical link is flow-controlled, a –1 is returned, and the external variable *errno* is set to *FLOW_CONTROL*.

Error Codes

BY_OBJECT	Local or remote program closed the link
MANAGEMENT	Link was disconnected by network
NODE_FAILED	Remote node failed to respond
NOT_CONNECTED	Logical link does not exist
OUT_OF_SPACE	Temporarily out of kernel buffers
REMOTE_ABORT	Link was aborted by remote program
UNKNOWN_ERR	Error code sent by remote node is undefined at local node

5.13.2 Sending Data in Record Format

In record format, the *write*() function results in the transmission of a specified number of bytes from the given buffer. The number of bytes is specified in the *sr_status* field. Sending messages in an incorrect order, (for example, *COMPLETE* after *BEG_OF_MESSAGE*) results in unpredictable results.

Call Usage

```
typedef struct session_record {
    short sr_status;
    char *sr_buffer;
    char  sr_reserved[6];
} SesRecord;

SesRecord sesrec;
int   link;
int   nbytes;
int   ret;
ret = write(link, &sesrec, nbytes);
```

Arguments

SesRecord	*typedef* defined in <*dn/defs.h*>.
link	Logical link file descriptor.
sesrec	Structure containing the address of the buffer with the transmission data and status information about this data. The status of the write is placed in the *sr_status* field. The *sesrec.sr_status* field should be set to a value that indicates where this block of data fits within a message. These values can be set by the application: BEG_OF_MESSAGE MID_OF_MESSAGE END_OF_MESSAGE COMPLETE (See Appendix B for more information on these completion codes.) *sesrec.sr_reserved* must be zero.
ret	The actual number of bytes that were sent.
nbytes	The length of the buffer (in bytes) to transmit. The maximum length supported by 4DDN is found by using the *ioctl* call *SES_MAX_IO*.

Results

The *write*() call returns a the number of bytes sent. If this value is –1, then an error was detected, and the external variable *errno* is set to the appropriate error code. (See Appendix B for recommended actions.)

If the I/O mode is blocking, the *write*() function blocks until the data buffer is copied to the controller for transmission.

If the I/O mode is blocking, the *write*() function returns immediately. If the logical link is flow-controlled, then a –1 is returned, and the external variable *errno* is set to *FLOW_CONTROL*.

Note: If the statuses from multiple record format *write*() function calls are sent out of sequence (for example, *MID_OF_MESSAGE* before *BEG_OF_MESSAGE*), results are unpredictable.

Error Codes

BY_OBJECT	Local or remote program closed the link
MANAGEMENT	Link was disconnected by network
NODE_FAILED	Remote node failed to respond
NOT_CONNECTED	Logical link does not exist
OUT_OF_SPACE	Temporarily out of kernel buffers
REMOTE_ABORT	Link was aborted by remote program
UNKNOWN_ERR	Error code sent by remote node is undefined at local node

5.14 Transmitting Interrupt Data

Interrupt data is high-priority information that is immediately transmitted through an *ioctl* call. The *XMIT_INTERRUPT ioctl* returns immediately with a success or failure indication. Because of its importance, the exchange of interrupt data is flow controlled by 4DDN software. For this reason, the *XMIT_INTERRUPT ioctl* returns a –1, and the external variable *errno* is set to *FLOW_CONTROL* providing that the previous *XMIT_INTERRUPT ioctl* has not yet been received by the remote program.

Call Usage

```
typedef struct image_16 {
    char im_length;
    char im_data[DATA_LEN];
    char im_rsvd;
} Image16;

int     link;
Image16 data;
int     ret;
ret = ioctl(link, XMIT_INTERRUPT, &data);
```

Arguments

Image16	*typedef* defined in <*dn/defs.h*>.
link	Logical link file descriptor.
XMIT_INTERRUPT	*ioctl* request code.
data.im_length	Length of data (0-16 bytes).
data.im_data	Interrupt data.
data.im_rsvd	Must be zeroed.
ret	Value returned by *ioctl*().

Results

Upon successful completion, this *ioctl*() call returns 0. If it returns –1, then an error has occurred, and the external variable *errno* is set to the appropriate error code. (See Appendix B for recommended actions).

An *errno* of *FLOW_CONTROL* indicates a non-fatal, temporary condition. In the case of this error, the transmit may be retried.

Error Codes

BY_OBJECT	Local or remote program closed the link
FLOW_CONTROL	Transmit failed; the logical link has been flow controlled
MANAGEMENT	Link was disconnected by network
NODE_FAILED	Remote node failed to respond
NOT_CONNECTED	Logical link does not exist
OUT_OF_SPACE	Temporarily out of kernel buffers
REMOTE_ABORT	Link was aborted by remote program
UNKNOWN_ERR	Error code sent by remote node is undefined at local node

5.15 Receiving Interrupt Data

Receiving interrupt data is a two-step process. The first step is to issue an *ACCEPT_INT ioctl* request. This call is issued only once; it specifies to the device driver the signal to raise when interrupt data is received.

The second step is performed every time data is received. The *RECV_INTERRUPT ioctl* request places interrupt data into the *im_data* field.

5.15.1 ACCEPT_INT ioctl()

Use this system call only once before any interrupt data is passed over the logical link to specify the interrupt data signal.

Call Usage

```
int link;
int ret;
int sig_no;
void func();

signal(sig_no, func);
ret = ioctl(link, ACCEPT_INT, &sig_no);
```

Arguments

link	Logical link file descriptor.
sig_no	Signal sent when interrupt data are received.
func	*func* is the name of the function to be called when interrupt data are received. See below.
ACCEPT_INT	Appropriate *ioctl* function code.
ret	Value returned by *ioctl*().

Results

ioctl() returns 0 when it completes successfully. If it returns –1, an error occurred, and the external variable *errno* is set to the appropriate error code. (See Appendix B for recommended actions.)

5.15.2 RECV_INTERRUPT ioctl

Use this system call to receive interrupt data.

Call Usage

```
typedef struct image_16 {
    char im_length;
    char im_data[DATA_LEN];
    char im_rsvd;
} Image16;

int     link;
Image16 id;
int     ret;
ret = ioctl(link, RECV_INTERRUPT, &id);
```

Arguments

Image16	*typedef* defined in <*dn/defs.h*>.
link	Logical link file descriptor.
RECV_INTERRUPT	Appropriate *ioctl* function code.
id.im_length	Contains the length of the data when the *ioctl* returns.
id.im_data	Buffer used for storing the received interrupt data.
ret	Value returned by *ioctl* call.

Rules

1. The signal number must be registered with the *signal*(2) call before this *ioctl* is issued.

2. The *ACCEPT_INT ioctl* call may be issued only once, before any I/O takes place on the logical link.

Results

The *RECV_INTERRUPT ioctl* request places interrupt data into the *id.im_data* field. The value of *id.im_length* is set to the number of bytes received. If the call takes place successfully, the *ioctl*() returns zero.

If –1 is returned, and the external variable, *errno*, is set to *NO_DATA_AVAIL*, then there was no interrupt data for this link. If –1 is returned, and the external variable *errno* is not set to *NO_DATA_AVAIL*, then an error occurred, and the external variable *errno* contains the appropriate error code. (See Appendix B for recommended actions.)

Error Codes

BY_OBJECT	Local or remote program closed the link
MANAGEMENT	Link was disconnected by network
NO_DATA_AVAIL	No data available (read only)
NODE_FAILED	Remote node failed to respond
NOT_CONNECTED	Logical link does not exist
OUT_OF_SPACE	Temporarily out of kernel buffers
REMOTE_ABORT	Link was aborted by remote program
UNKNOWN_ERR	Error code sent by remote node is undefined at local node

Example

The following program illustrates the two-step process to accept and receive interrupt data.

```
int     link;
int     ret;
Image16 int_data;
void    int_handler();
int     sig_no = SIGNAL_NUMBER;

/* Main routine or subroutine */

routine()
{
     /*
      * STEP 1: register the signal handler, then
      * issue ioctl to accept interrupt data.
      */

     signal(sig_no, int_handler);
     ret = ioctl(link, ACCEPT_INT, &sig_no);
}

/* Interrupt notification routine -
 * Issue ioctl to receive interrupt data and
 * reregister the signal.
 */

void
int_handler()
{
    /*
     * STEP 2: if interrupt data is available, issue the
     * RECV_INTERRUPT ioctl call to get the interrupt data.
     */

     ret = ioctl (link, RECV_INTERRUPT, &int_data)

     /* Re-register the signal handler */

     signal (sig_no, int_handler);
}
```

5.16 Disconnecting a Logical Link

This system call disconnects a logical link. A disconnect operation can be initiated at either end of a logical link connection.

Call Usage

```
typedef struct image_16 {
    char im_length;
    char im_data[DATA_LEN];
    char im_rsvd;
} Image16;

typedef struct session_data {
    short   sd_reason;
    Image16 sd_data;
    char    sd_rsvd[4];
} SessionData;

int         link;
SessionData sd;
int         ret;
ret = ioctl(link, SES_DISCONNECT, &sd);
```

Arguments

SessionData	*typedef* defined in <*dn/defs.h*>.
Image16	*typedef* defined in <*dn/defs.h*>.
link	Logical link file descriptor.
SES_DISCONNECT	Appropriate *ioctl* function code.

sd	Disconnect data sent to the program at the other end of the logical link. *sd_data.im_length* indicates the length of the data. *sd_reason* gives the reason for disconnect and is application dependent. The value of *sd_data.im_data* is application dependent. Disconnect data is optional. If omitted, *sd_rsvd* and *sd_data.im_rsvd* must be binary zero.
ret	Value returned by *ioctl*().

Description

This *ioctl* request is issued by the client or the server. Following a successful disconnect operation, the logical link must be closed to release the descriptor for subsequent use. Then a server program must issue a new *open*() and re-register itself as a server.

Disconnect guarantees the delivery of outstanding data (data that was sent but not acknowledged as received) before the link is terminated. The disconnect *ioctl* blocks until all transmitted data is received by the remote process.

Results

Upon successful completion, 0 is returned. If an error occurs, –1 is returned, and the external variable *errno* is set to the appropriate error code. (See Appendix B for recommended actions.)

Error Codes

BY_OBJECT	Local or remote program closed or rejected the link
MANAGEMENT	Link was disconnected by network
NODE_FAILED	Remote node failed to respond
NOT_CONNECTED	Logical link does not exist
OUT_OF_SPACE	Temporarily out of kernel buffers
REMOTE_ABORT	Link was aborted by remote program
UNKNOWN_ERR	Error code sent by remote node is undefined at local node

5.17 Aborting a Logical Link

This *ioctl* request is issued by the client or server to abort a logical link. It results in an abnormal termination of the link.

Call Usage

```
typedef struct image_16 {
    char im_length;
    char im_data[DATA_LEN];
    char im_rsvd;
} Image16;

typedef struct session_data {
    short   sd_reason;
    Image16 sd_data;
    char    sd_rsvd[4];
} SessionData;

int         link;
SessionData sd;
int         ret;
ret = ioctl(link, SES_ABORT, &sd);
```

Arguments

SessionData	*typedef* defined in <*dn/defs.h*>.
link	Logical link file descriptor.
SES_ABORT	*ioctl* request code.
sd	Abort data sent to the program at the other end of the logical link. Values are application-dependent. *sd.sd_data.im_length* indicates the length of the data. *sd.sd_reason* gives the reason for abort and is application dependent. The value of *sd.sd_data.m_data* is application dependent. Abort data is optional. If omitted, *sd.sd_data.im_length* must be set to 0. *sd.sd_rsvd* and *sd.sd_data.im_rsvd* must be binary zero.
ret	Value returned by *ioctl*().

Results

Upon successful completion, 0 is returned. On error, –1 is returned, and the external variable *errno* is set to the appropriate error code. (See Appendix B for recommended actions.)

Data not yet sent is discarded when the abort is sent to the remote node.

Comments

Following a successful abort operation, the logical link device must be closed to release the descriptor for subsequent use. Then a server program issues a new *open*() and re-registers itself as a server.

An abort constitutes an abnormal termination of the logical link.

Error Codes

BY_OBJECT	Local or remote program closed or rejected the link
MANAGEMENT	Link was disconnected by network
NODE_FAILED	Remote node failed to respond
NOT_CONNECTED	Logical link is not connected
OUT_OF_SPACE	Temporarily out of kernel buffers
REMOTE_ABORT	Link was aborted by remote program
UNKNOWN_ERR	Error code sent by remote node is undefined at local node

5.18 Closing a Logical Link

A logical link must be closed by issuing a *close*() function call. Both the client and the server must issue this command. The *close*() function call terminates the logical link and frees the logical link file descriptor for subsequent use. However, the *close*() function does not permit the transmission of data to the remote node before the logical link termination.

Two other methods can be used to terminate a logical link: it can be disconnected or aborted.

Call Usage

```
int  ret;
ret = close(link);
```

Description

link	Logical link file descriptor.
ret	Value returned by *close*().

Results

Upon successful completion, 0 is returned. On error, –1 is returned, and the external variable *errno* is set to the appropriate error code. (See Appendix B for recommended actions.)

Comment

To obtain optional data, issue a disconnect, or abort *ioctl* before *close*().

The *close*() call always terminates the link. It also frees the logical link identifier without sending optional data.

5.19 Obtaining Link Status

This call returns the status of a logical link. A program can inquire about the status of a logical link at any time.

Call Usage

```
long status;
int  ret;
ret = ioctl(link, SES_STATUS, &status)
```

Description

link	Logical link file descriptor.	
SES_STATUS	Appropriate *ioctl* function code.	
status	Status code for the link. Possible status values are the following:	
	NO_LINK	No logical link for given file descriptor
	LINK_OPEN	Logical link open but link not yet established

LINK_CONNECT	Logical link device open and logical link established
CLOSING	Remote system closed the logical link, waiting for local *close*()
ABORTED	Remote system aborted the logical link, waiting for local *close*()

ret Value returned by *ioctl*().

Results

A successful return of 0 indicates that an error code was placed into the status field. On error, –1 is returned, and the external variable *errno* is set to the appropriate error code.

5.20 Printing Error Messages

The *dn_perror* function prints an informative error message based on the *errno* variable. The function writes one line to the *stderr* (standard error) stream, which is usually the terminal. This line consists of the indicated string followed by a colon and the appropriate error message. To vary the formatting of error messages, use the *dn_strerror* function.

5.20.1 The *dn_perror* Function

This function prints an error message with the format *user-text-string:error-code-string*.

Call Usage

```
dn_perror(string);
   char *string;
```

Description

string A character array or a string constant.

Results

The *dn_perror* function has no return value.

5.20.2 The *dn_strerror* Function

This function returns an informative error message that corresponds to the given error number.

Call Usage

```
char *
dn_strerror(errnum)
   int errnum;
```

Description

errnum An error number resulting from a failed 4DDN *ioctl()* call.

Results

The *dn_strerror* function returns the message string for the given 4DDN error number. Be aware that the string might be a static buffer that can be modified in a subsequent call to this function.

Appendix A

NFARS Error Messages

NFARS error message are listed below with an explanation of the probable cause of the error and a recommended action. (Messages appear in boldface, for clarity.) For some error conditions, you might need to consult your system manager; software errors should be reported. Any messages not shown in this section are task-to-task error messages. (See Appendix B, "4DDN Error Codes" for more information.)

local discovered protocol error
remote discovered protocol error
unknown error
DAP error detected
state table error
unsupported operation
network operation failed at remote
message building failed

Meaning: The above messages explain the general cause of the error. The messages include a set of codes that explain the exact cause of the problem. The message may indicate an incompatibility between the system or a shortcoming in an implementation.

Recommended Action:
Report the error to your service organization. Please include the the 4DDN software version (*dncp* **-r**); the s vendor name and operating system version of the remote node; the exact command line; a printout of the results of the command line (please be precise); and a full directory listing showing the subject file(s). If encountered using an NFARS routine, please include source code.

`operation aborted`

Meaning: The remote host aborted the assigned operation. No explanation is available.

Recommended Action:
Try executing the command again and, if the error persists, report it to your service organization for analysis.

`link was not established`
`cannot alloc NFARS NCB structure`

Meaning: These errors are extremely unlikely to occur.

Recommended Action:
Report the error to your service organization. Please include the 4DDN software version (*dncp* **-r**); the vendor name and operating system version of the remote node; the exact command line; a printout of the results of the command line (please be precise); and a full directory listing showing the subject file(s). If encountered using an NFARS routine, please indicate source code.

`invalid wildcard operation`

Meaning: This message indicates that an invalid wildcard operation was sent to a function that is capable of handling wildcard operation. The system where the function was called cannot perform the operation.

Recommended Action:
Attempt the operation again on a single file.

Note: This message does is not generated by the RFAS programs.

invalid NFD/NWD
inactive DAP link
inconsistent arguments
inappropriate operation

Meaning: The above messages all indicate that the user provided wrong information to one of the follow-on NFARS functions: *net_read*, *net_write*, or *net_fnext*. These messages are not generated by the RFAS programs.

Recommended Action:
Make certain that the functions are called with the proper arguments.

invalid or missing filespec
invalid device or volume
invalid directory
invalid file
invalid version

Meaning: The above messages all indicate that the remote system has found an error in the indicated part of a filespec. It may mean an invalid character, unknown element (such as a directory or device, for example) or an incorrect format.

Recommended Action:
Review the specification rules for the remote system.

no file attributes for dir list
error in reading name for dir list
error in reading attribs; dir list
unable to recover; dir list

Meaning: These messages indicate very rare problems in the creation of parts of a directory list. They may occur in any of the RFAS programs, as well as with the *net_find* and *net_fnext* functions.

Recommended Action:
These errors are unavoidable and have no known work-around, other than to use a less ambiguous filespec and avoid the files that have strong access control.

`no more files` (wildcard expansion)

Meaning: This message is postedeafter the last item of a wildcard expansion has been retrieved. Any further calls to *net_fnext* result in failures.

`error deleting full directory`
`error deleting a locked file`
`error deleting a file`

Meaning: These messages occur when applications encounter problems with calls to *net_delete*. The messages are self-explanatory. These messages occur only with *dnrm* and calls to *net_delete*.

`2 different devices in rename`
`cannot rename old file systems`
`invalid directory rename operation`
`inconsistent nodes for rename`
`rename mismatch Access Control info`
`rename failed; file lost`

Meaning: These messages occur only with *dnmv* and calls to *net_rename*. Renaming a file is valid only on a single node. Restrictions might limit renaming across devices on that node. You can use two sets of access control information for a renaming operation; however, the information in both sets must be identical.

`file not found`

Meaning: This message originates from any NFARS routine that uses a file specification as an argument; it means that the desired file does not exist.

Recommended Action:
Verify the name of the file and try again.

file already exists

Meaning: When a call to *net_open* with RFO_CREATE is executed, the specified file already exists.

Recommended Action:
If this results from a *net_open* with RFO_CREATE, then the caller should try again and use RFO_WRITE instead of RFO_CREATE.

access permission violation
privilege violation(OS denies access)
file is locked by another user

Meaning: These messages occur when file access is denied due to improper access permission or a conflict in a current access.

Recommended Action:
Check the access to the target file(s). If locked by another user, the caller should retry the call later.

error in opening file
error in reading file
error in writing file
device or file are full
error in closing file

Meaning: These messages are the result of problems encountered during the execution of ordinary NFARS functions and they are self-explanatory. These errors might occur in the *dncp* program and the operation has, most likely, failed. However, part of the operation may have been performed, and the file may contain unpredictable data.

end of file

Meaning: This informative message is received by *net_read* when the read reaches the end of the open file.

`bad data format`

Meaning: This error concerns the data that a user provides to *net_write*. Data for the non-verbatim mode, that is, without the RFM_VERBATIM bit masked with RFO_WRITE or RFO_CREATE, must have local line terminations at suitable intervals. The largest number of characters in a line (between terminators) is 510 bytes.

Recommended Action: Use verbatim mode to transfer data byter-for-byte (without record conversion).

Appendix B

4DDN Error Codes

Table B-1 lists the error codes returned in *errno* with the appropriate recommended actions.

Name	Meaning	Recommended Actions
NOT_CONNECTED	The logical link does not exist.	Check program.
PROC_ERROR	Remote node received too much connect data.	Contact your service organization.
BAD_LINK	An invalid logical link ID was specified.	Check the program. Probably trying to access a closed link.
BY_OBJECT	The local or remote process has closed or rejected the link.	This indicates a normal close of the link.

Table B-1 *errno* Error Codes

Name	Meaning	Recommended Actions
NET_RESOUR	Insufficient network resources.	Try again.
NODE_NAME	Unrecognized node name.	Check the NCP database with the "show known nodes" command to make sure the node exists.
NODE_DOWN	The remote node is not accepting new links.	Try again. The remote node is being disconnected from the network.
BAD_OBJECT	The specified remote object does not exist.	Either the object number/task name passed in the OpenBlock is wrong or the requested server has not been registered on the remote node.

Table B-1. *errno* Error Codes (continued)

Name	Meaning	Recommended Actions
OBJ_NAME	The specified task name is invalid.	Change the program to correct the format. Verify that the task name format follows the rules in Chapter 4.
OBJ_BUSY	Insufficient resources at the remote node.	Try again later.
MANAGEMENT	The link was disconnected by the network.	Try again later. The remote node may have become inactive.
REMOTE_ABORT	The link was aborted by the remote process.	Check the remote program. It may have crashed.
BAD_NAME	The node name is invalid.	Verify that the node name in the OpenBlock is valid.
LOCAL_SHUT	The local node is not accepting new links. The STATE of the node is OFF.	Set the node STATE to ON using NCP.
ACCESS_CONT	The remote node or process rejected the access information.	Check the access control information given in the OpenBlock.

Table B-1. *errno* Error Codes (continued)

Name	Meaning	Recommended Actions
LOCAL_RESOUR	The local node does not have resources for a new link.	Too many links are currently open. Kill unneeded programs with open links.
NODE_FAILED	The remote node failed to respond.	Check if the remote node is responding, then retry.
NODE_UNREACH	The remote node is currently inactive.	Use the *shownet* command to determine the status of the remote node and try again when the remote node becomes active.
ALREADY	Logical link identifier is already in use.	Check the program.

Table B-1. *errno* Error Codes (continued)

Name	Meaning	Recommended Actions
USER_ABORT	Program aborted by interactive user at terminal.	This status code will not be returned in the current version.
INV_ACCESS_MODE	Invalid access attempt on read or write.	Reopen the link with the correct address modes.
NO_DATA_AVAIL	No data available in non-blocking input mode.	No action required.
BAD_RECORD_STAT	The status given in record format during a write is invalid.	Modify the status in the sr_status_field.
INVALID_SIZE	Size of transmit buffer is greater than DN_MAX_IO.	Issue the *ioctl SES_MAX_IO* to determine the maximum transmit buffer size. Then reduce the size of the transmit buffer.

Table B-1. *errno* Error Codes (continued)

Name	Meaning	Recommended Actions
OUT_OF_SPACE	No kernel buffer space available.	Retry the program later.
BAD_COMMAND	Invalid *ioctl* command.	Check the program's *ioctl* calls.
FLOW_CONTROL	Transmit failed. Logical link has been flow controlled and I/O mode is non-blocking.	Normal status in non-blocking I/O mode. Reissue the transmit with the same buffer address and length.
CL_DATA_AVAIL	The link was closed by the remote node but data is still available to be read.	Continue reading from the link to receive all available data or just close the link to discard the data.
INT_DATA	Internal 4DDN status. Will not be returned to the user.	No action required.

Table B-1. *errno* Error Codes (continued)

Name	Meaning	Recommended Actions
BEG_OF_MESSAGE	Read returned the first part of a message that is larger than the input buffer specified.	Continue reading until the *END_OF_MESSAGE* status.
MID_OF_MESSAGE	Read returned the next part of a message that is larger than the input.	Continue reading until the *END_OF_MESSAGE* status.
END_OF_MESSAGE	Read returned the last part of a message that is larger than the input buffer specified.	No action is required.
COMPLETE	READ returned a complete message.	No action is required.
UNKNOWN_ERR	Error code sent by remote node is undefined at local system.	Try again. If it does not work, contact your service organization.

Table B-1. *errno* Error Codes (continued)

Name	Meaning	Recommended Actions
DUPE_NODE_NAME	Duplicate node name detected.	Check the NCP database. If this node had a different number when 4DDN was initialized, change its name with the NCP *SET NODE* and *DEFINE NODE* commands.
DUPE_NODE_NUM	Duplicate node number detected.	Check the NCP database. If this node had a different number when 4DDN was initialized, change its name with the NCP *SET NODE* and *DEFINE NODE* commands.
NODE_NUM_REQUIRED	Node records require the node numbers.	Check the contents of the NCP database.
NOT_SUPPORTED	Function not yet supported.	Should not appear.

Table B-1. *errno* Error Codes

Appendix C

Sample Programs

These sample test programs illustrate task-to-task communication by showing an exchange of data using 4DDN. These programs exist in the directory */usr/etc/dn/examples*.

C.1 *client.c*

```
/*
 * Module:  CLIENT.C  -  Example DECnet Client Program
 *
 ******************************************************************************
 *                                                                            *
 *       COPYRIGHT 1985, 1986  BY TECHNOLOGY CONCEPTS INC.                    *
 *                                SUDBURY, MASSACHUSETTS 1776                 *
 *       COPYRIGHT 1988 SILICON GRAPHICS, INC.                                *
 *                                -- ALL RIGHTS RESERVED --                   *
 *                                                                            *
 *    THIS SOFTWARE IS FURNISHED UNDER LICENSE AND MAY BE USED AND COPIED     *
 *    ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE AND WITH THE INCLUSION*
 *    OF THE ABOVE COPYRIGHT NOTICE. THIS SOFTWARE OR ANY OTHER COPIES THEREOF*
 *    MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY OTHER PERSON. NO *
 *    TITLE TO AND OWNERSHIP OF THE SOFTWARE IS HEREBY TRANSFERRED.           *
 *                                                                            *
 *    THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE AND*
 *    SHOULD NOT BE CONSTRUED AS A COMMITMENT BY TECHNOLOGY CONCEPTS INC. AND *
 *    SILICON GRAPHICS INC.                                                   *
 *                                                                            *
 *    DECnet is a trademark of Digital Equipment Corporation                  *
 *                                                                            *
 ******************************************************************************
 *
 * Version:     1        Revision: 1
 *
 * Facility:    Example client program
 *
```

```
 * Abstract:    This program demonstrates how to exchange messages with a
 *              remote node using IRIS-DN.  This operation is called
 *              task-to-task communication and is performed by issuing commands
 *              to the IRIS-DN network software.
 *
 *              This program demonstrates how to:
 *
 *              1) Establish a logical link as the client
 *
 *                 A logical link must be established between this host and
 *                 the remote node before messages can be exchanged.  To
 *                 establish a logical link, one node must initiate the link
 *                 request. The initiating node, or program, is called the
 *                 client and the receiving program is called the server.
 *                 This program is an example of the client. It requests
 *                 a logical link, or connection, to a server.
 *
 *              2) Exchange messages over the logical link
 *
 *                 Once the logical link is established, no distinction is
 *                 made between the client and the server. Both programs
 *                 can receive and send messages across the logical link using
 *                 the read() and write() functions respectively.
 *
 *              3) Terminate the logical link
 *
 *                 Before a client program terminates, it must close
 *                 the logical link.
 *
 ****************************************************************************/

/*  Include files  */
#include <stdio.h>
#include <fcntl.h>
#include <dn/defs.h>

/*  Constant definitions  */
#define  NUM_BYTES  100                /* Maximum number of bytes to read */


/*   Global data definitions  */
int             ll;                    /* Logical link identifier         */
char            buffer[NUM_BYTES+1];   /* Character buffer                */
OpenBlock       opblk;                 /* OpenBlock typedef is defined in
                                          <dn/defs.h>             */



/*  Program description
 *
 *  In this example, our node name will be "CLIENT".  We will make a logical
 *  link request to the task name "EXAMPLE" on a remote node specified
```

```
 *  on the command line.  If the server accepts our logical link request, we
 *  will send it the message "This is an example".  We will then wait for the
 *  reply message, "Got it".  After we receive this message we will terminate
 *  the connection and exit the program successfully.  If an error is returned
 *  from any IRIS-DN function call, error() or the library routine dn_perror()
 *  is called to display the error message.
 */

main(argc, argv)
    int  argc;
    char **argv;        /* argv[1] is the server's node name */
{
    int ret;
    int len;


    /*  Before establishing a logical link, we must first open the
     *  logical link device, DN_LINK.
     */

    if ((ll = open(DN_LINK, O_RDWR)) < 0) {
        dn_perror("Open Fail: ");
        exit(1);
    }


    /*  Next, we must make the logical link request to the server.
     *  To do this, we must specify the remote node and the server task
     *  we want to connect to. In addition, we must identify ourself
     *  so the server knows who is making the request.
     *  This information is contained in a data structure called the
     *  OpenBlock. We will fill in an OpenBlock with the necessary
     *  data, then issue the SES_LINK_ACCESS ioctl() function to make
     *  the logical link request to the server.  The ioctl() function
     *  will return a 0 if the link is established to the server.
     *  If it returns a -1, the link is not open. The reason or error
     *  number is contained in the external variable errno.
     */

    bzero((char *) &opblk, sizeof(opblk)); /* Any field not used must be zero */
    if (argc == 2) {
        strcpy(opblk.op_node_name, argv[1]);    /* Remote node name */
    } else {
        strcpy(opblk.op_node_name, "SERVER");   /* default if not given */
    }
    strcpy(opblk.op_task_name, "EXAMPLE");      /* Server task name */
    strcpy(opblk.op_userid,    "CLIENT");       /* Our ID */

    if (ioctl(ll, SES_LINK_ACCESS, &opblk) < 0) {
        error("link");
    }
```

```
    /*  The logical link is established once our connect request is
     *  accepted by the server.  We may now proceed to send and receive
     *  data across the link using the read() and write() functions.
     *  We will now send the message "This is an example" to the server.
     *  We will then wait to receive the response message before terminating
     *  the connection. Note that we are using the default I/O options
     *  (stream data format and blocking reads).
     */


    /*  First, copy the message to send into the character buffer allocated
     *  The copied string is NULL-terminated so we must add 1 to the
     *  string length for the NULL byte. Then send the message.
     */

    strcpy(buffer, "This is an example");
    len = strlen(buffer) + 1;

    if ((ret = write(ll, buffer, len)) < 0) {
        error("write");
    }


    /*  Wait to receive the response message. */

    if ((ret = read(ll, buffer, NUM_BYTES)) < 0) {
        error("read");
    }


    /*  If the read was successful, display the message. Note that ret
     *  contains the actual number of bytes received.
     */

    display_msg(buffer, ret);


    /*  Terminate the connection before successfully exiting the program.
     *  This example chooses not to send the optional disconnect data
     *  Therefore, only close() is needed.
     */

    close(ll);
}

/*
 *  Display message routine
 */

display_msg(buf, count)
    char *buf;
    int  count;
```

```
{
    buf[count] = ' ';
    printf("Received reply '%s'\n", buf);
}


/*
 *  Error handler routine
 */

error(where)
    char *where;
{
    /*  An error has occurred. Dn_perror displays the appropriate
     *  message based on the external variable errno. The close()
     *  system call will disconnect the logical link.
     */

    dn_perror(where);
    close(ll);
    exit(1);
}
```

C.2 *server.c*

```
/*
 * Module:  SERVER.C  -  Example DECnet Server Program
 *
 ******************************************************************************
 *                                                                            *
 *       COPYRIGHT (C) 1985, 1986  BY TECHNOLOGY CONCEPTS INC.                *
 *                                 SUDBURY, MASSACHUSETTS 1776                *
 *       COPYRIGHT 1988 SILICON GRAPHICS, INC.                                *
 *                                  -- ALL RIGHTS RESERVED --                 *
 *                                                                            *
 *    THIS SOFTWARE IS FURNISHED UNDER LICENSE AND MAY BE USED AND COPIED     *
 *    ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE AND WITH THE INCLUSION*
 *    OF THE ABOVE COPYRIGHT NOTICE. THIS SOFTWARE OR ANY OTHER COPIES THEREOF*
 *    MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY OTHER PERSON. NO *
 *    TITLE TO AND OWNERSHIP OF THE SOFTWARE IS HEREBY TRANSFERRED.           *
 *                                                                            *
 *    THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE AND*
 *    SHOULD NOT BE CONSTRUED AS A COMMITMENT BY TECHNOLOGY CONCEPTS INC. AND *
 *    SILICON GRAPHICS INC.                                                   *
 *                                                                            *
 *    DECnet is a trademark of Digital Equipment Corporation                  *
 *                                                                            *
 ******************************************************************************
 *
 * Version:     1        Revision: 1
```

```
 *
 * Facility:    Example server program
 *
 * Abstract:    This program demonstrates how to exchange messages with
 *              a remote node using IRIS-DN. This operation is called
 *              task-to-task communication and is performed by calling
 *              system routines to access the IRIS-DN network software.
 *
 *              This program demonstrates how to:
 *
 *              1) Establish a logical link as a server
 *
 *                 A logical link must be established between this host and
 *                 the remote node before messages can be exchanged.  To
 *                 establish a logical link, one node must initiate the
 *                 logical link request.  The initiating program is called the
 *                 client and the receiving program is called the server.
 *                 This program is an example of the server.  It demonstrates
 *                 how a server program registers itself and waits for a
 *                 logical link request.  It also demonstrates how a server
 *                 may use the access control information received with a
 *                 request to decide whether to accept or reject the logical
 *                 link request.
 *
 *              2) Exchange messages over the logical link
 *
 *                 Once the logical link is established, no distinction is
 *                 made between the client and the server.  Both programs
 *                 can receive and send messages across the logical link using
 *                 the read() and write() functions respectively.
 *
 *              3) Terminate the logical link
 *
 *                 Before a server program terminates,  it must close() the
 *                 logical link.  Before it can receive additional logical
 *                 link requests, a server program must reopen the logical
 *                 link device and reregister itself.
 *
 *
 ****************************************************************************/


/*  Include files  */
#include <stdio.h>
#include <fcntl.h>
#include <dn/defs.h>


/*  Constant definitions  */
#define  NUM_BYTES  100               /* Maximum number of bytes to read */
```

```
/*  Global data definitions  */

int             ll;                     /* Logical link identifier */
char            buffer[NUM_BYTES+1];    /* Character buffer */
OpenBlock       opblk;                  /* OpenBlock typedef is defined
                                           in <dn/defs.h> */



/*  Program description
 *
 *  In this example, we will register ourself as a server for the task
 *  name "EXAMPLE" and wait for a logical link request.  If the request
 *  received is for the user name "CLIENT", we will accept the logical
 *  link request, otherwise we will reject it.  Once a logical link is
 *  established, we will wait to receive the message "This is an example".
 *  Upon receiving it, we will display it and send back the reply message,
 *  "Got it".   Then we will terminate the connection and exit the program
 *  successfully.  If an error is returned from any IRIS-DN function call,
 *  error() or the IRIS-DN error message routine dn_perror() is called to
 *  display the error message.
 */

main()
{
    int len;
    int ret;
    SessionData sd;


    /*  Before establishing a logical link, we must first open the
     *  logical link device, DN_LINK.
     */
    if ((ll = open(DN_LINK, O_RDWR)) < 0) {
        dn_perror("open");
        exit(1);
    }


    /*  Next, we must register ourself as a server for the task name "EXAMPLE".
     */

    if (ioctl(ll, SES_NAME_SERVER, "EXAMPLE") < 0) {
        error("name server");
    }


    /*  Once registered as a server, we must wait for the access
     *  control information from a client with the SES_GET_AI ioctl()
     *  function. When a link request comes in, the client's access
     *  control information will be copied into opblk. Note that this
     *  ioctl() function will block until a request is made for this
```

```
 *  server or an error occurs.
 */

if (ioctl(ll, SES_GET_AI, &opblk) < 0) {
    error("Get AI");
} else {

    /*  We received a logical link request and OpenBlock.
     *  We must now determine whether or not we want to accept or
     *  reject the request. This determination is application
     *  dependent.  We will use the access control information in the
     *  OpenBlock just received to make this determination.  In this
     *  example, we will accept the request if it is from the user
     *  CLIENT, otherwise we will reject it. In this example, we do
     *  not send any return codes in the Session Data block with the
     *  SES_ACCEPT or SES_REJECT.
     */

    bzero((char *) &sd, sizeof(sd));
    if (strcmp(opblk.op_userid, "CLIENT") == 0) {
        if (ioctl(ll, SES_ACCEPT, &sd) < 0) {
            error("accept");
        }
    } else {
        if (ioctl(ll, SES_REJECT, &sd) < 0) {
            error("reject");
        }

        /*  A close() must always be issued after a SES_REJECT. */
        close(ll);

        /* Return an error to the shell. */
        exit(1);
    }

}

/*  The logical link is established once we (the server) accept the
 *  link request. We may now proceed to send and receive data across
 *  the link using the read() and write() functions.  We will first
 *  wait to receive the message "This is an example" from the remote
 *  node. Upon receiving it, we will display it and send back the
 *  message "Got it".  Then we will terminate the connection.  Note
 *  that we are using the default I/O data format and Input mode. They
 *  are stream data format and blocking reads.
 */

/*  Wait to receive a message from the remote node */

if ((ret = read(ll, buffer, NUM_BYTES)) < 0) {
    error("read");
}
```

```
    /*  If no error occurred, display the message. Note that ret
     *  contains the actual number of bytes received.
     */

    display_msg(buffer, ret);


    /*  Copy the response message into the allocated character buffer.
     *  The copied string is NULL-terminated, so we must add 1 to the
     *  string length for the NULL byte. Then send the response message.
     */

    strcpy(buffer, "Got it");
    len = strlen(buffer) + 1;

    if ((ret = write(ll, buffer, len)) < 0) {
        error("write");
    }


    /*  Terminate the connection before successfully exiting the program.
     *  In this example, we not to send optional disconnect data.
     *  Therefore, only the close() function is needed.
     */

    close(ll);

}


/*
 *  Display message routine
 */

display_msg(buf, count)
    char *buf;
    int  count;
{
    buf[count] = ' ';
    printf("Received message '%s'\n", buf);
}


/*
 *  Error handler routine
 */

error(where)
    char *where;
{
    /*  An error has occurred. Dn_perror displays the appropriate
     *  message based on the external variable errno. The close()
```

```
     *  system call will disconnect the logical link.
     */

    dn_perror(where);
    close(ll);
    exit(1);
}
```

Appendix D

Glossary

A

access control information
Information contained in the OpenBlock structure that is needed to access a remote node. This information includes username, password, and account.

active node
A node that is currently communicating or ready to communicate with another node.

adjacent node
See *active node*.

application-dependent
Fields of a data structure that can be filled in with user-defined data at the option of the application programmer.

area number
A number assigned to a group of nodes in the network to identify it. The area number must be an integer in the 1-63 range.

B

blocking I/O
A method of reading data in which a process waits to do the read operation until the data becomes available. See also *non-blocking I/O*.

C

client
A local process that requests a logical link connection in task-to-task communication. See also *server*.

collision
An event that results from simultaneous transmissions by two or more nodes on an Ethernet network.

congestion
A condition that occurs when too many packets are to be queued.

counters
Performance variables providing network management information. These variables can be displayed by using the *SHOW COUNTERS* command. They can be zeroed by using the *ZERO COUNTERS* command.

D

datagram
The portion of an Ethernet packet that remains after routing control information is removed.

Digital Network Architecture (DNA)
The Digital Network Architecture developed by Digital Equipment Corporation as the networking architecture for DEC systems.

DNA
See *Digital Network Architecture*.

E

Ethernet
A local area network using a Carrier-Sense Multiple Access with Collision Detect scheme to arbitrate the use of a 10-megabit-per-second baseband coaxial cable.

F

flow control
The function performed by a receiving node to limit the amount or rate of data that is sent by a transmitting node. Flow control is automatically activated by the network software as memory for transmit and receive buffers becomes scarce. When activated, the receiving node notifies the transmitting node to stop sending data messages. After this occurs, the transmitting node must wait for a message from the receiving node to resume the transmission of data messages.

frame
A synonym for packet in Ethernet terminology.

I

inactive node
A node that is not currently communicating or ready to communicate with another system on the Ethernet. See also *active node*.

interrupt data
Special high-priority control information that is transmitted immediately.

L

logical link
A virtual circuit between two application programs.

logical link device
A virtual I/O device responsible for controlling logical links.

N

NCP
See *Network Control Program*.

Network Control Program (NCP)
A utility at the user level that interfaces with lower level modules. It provides a set of interactive commands that the user enters at the terminal.

non-blocking I/O
A method of reading data in which a process does not wait until data is available before performing a read operation. If a special interrupt signal is registered, the process is notified when data becomes available. See also *blocking I/O*.

null-terminated string
A string that ends with zero.

O

object number
A number used instead of a name for addressing a process in task-to-task communication.

OpenBlock structure

The data structure created by a 4DDN client process containing the information needed to establish a DECnet connection. This information includes the node name, object type or name, user name, and the password.

optional data

A special data field that is generally used by the application program to explain the reason for terminating a logical link.

P

packet

A unit of data to be routed from a source node to a destination node. When its routing header is removed and the packet is passed to the End Communication Layer, it becomes a datagram.

R

record format

A method of exchanging data in which the transmitted message contains a structure indicating whether it is complete. If the message is incomplete, a special status field indicates whether it is the beginning, the middle, or the end of the message. See also *stream format*.

S

server
A remote process that accepts or rejects a logical link connection when a process is attempting to establish connection in task-to-task communication stream I/O data format. See also *client*.

stream format
A method of exchanging data in which a process receives data as it appears across the logical link without distinguishing where messages begin and end. See also *record format*.

T

task-to-task communication
The exchange of data between two processes over a logical link.

Appendix E

IRIX Manual Pages

This appendix contains the IRIX manual pages that pertain to programming with 4DDN software.

Index

E

F

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-1302-020.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-932-0801
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389