4DDN User's Guide

IRIS-4D Series

SiliconGraphics
Computer Systems

Document number: 007-0820-010

4DDN User's Guide

Version 1.0

Document Number 007-0820-010

9/90

Technical Publications:

Kevin Walsh
Diane Wilford

Engineering:

Vernon Schryver
Andrew Cherenson

4DDN User's Guide
Version 1.0
Document Number 007-0820-010

Silicon Graphics, Inc.
Mountain View, California

Contents

1. Introduction

IRIS-4DDN is a software communications product that enables your IRIS-4D workstation to communicate on a DECnet Phase IV network as an Ethernet end node. 4DDN is an implementation of the Digital Network Architecture (DNA) protocols.

This document provides an overview of the IRIS-4DDN user software, an explanation of Network Virtual Terminal, the Network File Access System, and task-to-task communication. It also provides guidelines on how to write network application programs. A glossary is provided at the end to define some technical terms.

1.1 Services Provided by 4DDN

IRIS-4DDN enables you to:

- Transfer files to, from, or between remote nodes on the network using the *dncp* (copy) command.
- List the contents of directories on a remote node on the network using the *dnls* (list) command.
- Remove a specified file in a remote directory on the network using the *dnrm* (remove) command.
- Move a specified file to a remote directory on the network using the *dnmv* (rename) command.
- Log on to a remote node on the network using the *sethost* command. Your terminal acts as though it were connected to that computer, with that computer executing commands you type at your terminal.

- Control and monitor the network using the Network Control Program (ncp).
- Write programs that exchange data with programs on other computers.

1.2 Structure of the Guide

This manual consists of five chapters and three appendices.

Chapter 1: This chapter contains general information about IRIS-4DDN.

Chapter 2: This chapter describes the Network Virtual Terminal utility, sethost.

Chapter 3: This chapter describes the Network File Access commands: dncp (copy), dnls (directory), dnrm (delete), and dnmv (rename). Error messages generated by these commands are explained.

Chapter 4: This chapter describes the programmer's interface to the Network File Access Routines (NFARS). It is intended for experienced programmers.

Chapter 5: This chapter provides a detailed explanation of task-to-task communication (session layer) for the experienced programmer. It also contains the programmer's reference guide with the specifications required for writing task-to-task network applications in the C programming language.

Appendix A: This appendix contains the 4DDN programming error codes and the corresponding recommended actions.

Appendix B: This appendix contains two sample programs: one program is provided for the server; and a second program is provided for the client.

Appendix C: This appendix contains a glossary of terms that are used throughout this manual.

1.3 Conventions

This document uses the standard IRIX™ convention 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 command syntax descriptions and examples, square brackets surrounding an argument indicate that the argument is optional. Variable parameters are in *italics*. You replace these variables with the appropriate string or value.

In text descriptions, filenames and IRIX commands are also in *italics*. Names of variables and error codes used in the VMS environment are in uppercase.

Other conventions used in the DECnet environment are listed below:

CAPITAL	Words that are capitalized and in bold type are commands that must be entered by the user. Press Return after making the entry.
node-name	Multi-word variables are hyphenated.
[ ]	Square brackets indicate an optional argument.
( )	Function calls are denoted by a pair of parentheses immediately following the name of the function.

1.4 Related Documentation

Silicon Graphics, Inc. *4DDN Network Manager's Guide*	007-0821-010
Digital Equipment Corporation *DECnet Digital Network Architecture (Phase IV) General Description*	DEC part #: *AA-N149A-TC*
Digital Equipment Corporation *Guide to Networking on VAX/VMS*	DEC part #: AA-Y512A-TE

1.5 Product Support

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

2. Network Virtual Terminal Utility

This chapter explains how to connect to a remote node (IRIS-4DDN or VAX/VMS) and log on to that node using 4DDN or DECnet software.

The following operations are possible:

- Logging on to a remote VAX/ULTRIX/RSX node running DECnet from a 4DDN node
- Logging on to a remote 4DDN node from a VAX/ULTRIX/RSX node

2.1 Establishing a Virtual Terminal Session

Before logging on to a remote system, first establish a connection between your node and a process called the virtual terminal server at the remote node. The sethost command establishes this connection using 4DDN's task-to-task interface.

When the virtual terminal session is established, the remote system prompts you for the necessary log on information. Once you are logged on to a remote node, you can issue any commands that a user on that node can issue. The operating system at the remote node interprets the commands and performs the desired operations.

For example, you can log on to a remote node and edit a file residing there, direct a file to be printed on the remote system's printer, or compile a program using the remote system's compiler.

2.2 Software Prerequisites

In order to use the *sethost* command, the remote node you want to connect to must be running a virtual terminal server.

2.3 The *sethost* Command

The *sethost* command is used to log on to a remote node from your 4DDN system. The command resides in the directory /usr/bin/dn.

Command Syntax

```
sethost [-r] node-name
sethost [-r] node-address
```

Where

–r	The –r option displays the release and revision numbers of sethost and the 4DDN product.
node-name	is one of the remote node names. A node name is a group of not more than 6 letters and/or numbers. The first character must be a letter. The node name must be unique across the network.
node-address	is the address of an active remote node. It must conform to the <area-number.node-number> format. If the remote node is in the same network area as the local node, you can simply use the node number. The area number identifies a group of nodes in the network. The area number must be an integer in the 1-63 range. The node number must be an integer in the 1-1023 range. Node numbers must be unique across your network area.

If *node-name* or *node-address* is omitted, you are prompted to enter the information.

Next, you are prompted to supply the log on information required by the remote node.

2.4 Terminating a Virtual Terminal Session

You can normally terminate a virtual terminal session by returning to the remote operating system prompt and then logging out. Also, you can abort the virtual terminal session in the middle of a program.

2.4.1 Logging out of the Remote Node

To log out of a remote node, issue the log out command for that remote node (e.g., LOGOUT for VAX/VMS).

2.4.2 Aborting the Virtual Terminal Session

To abort the virtual terminal session from your terminal, type the <CTRL>-Y key combination twice. The following prompt is displayed:

```
Do you wish to abort the network virtual terminal session ?
```

A Yes (Y) answer returns you to the local system's command level.

A No (N) answer terminates the currently executing program and returns to the remote host system's command level.

2.5 Comments

1. If your terminal is a VT100 or compatible, in the terminal set-up options, WRAP needs to be set to ON, and NEWLINE needs to be set to OFF.
2. If you are logging into a VMS system, you can put a SET TERM/INQUIRE command in your *LOGIN.COM* file and VMS automatically determines your terminal type and sets the appropriate terminal characteristics.

3. Network File Access Commands

This chapter discusses the following 4DDN file access commands:

- *dncp* transfers files to, from, and between remote systems;
- *dnls* provides the listing of the contents of a directory on a remote system;
- *dnrm* removes specified file(s) on a remote directory;
- *dnmv* renames specified file(s) on a remote system.

The commands follow the IRIX conventions for command line syntax. Thus, each command line consists of a command name followed by command options and file specifications for the local and/or remote files. These commands reside in */usr/bin/dn*, which should be added to your shell's path environment variable.

3.1 Software Requirement at the Remote Node

All of the Network File Access Commands may be used with remote nodes that provide the File Access Listener (FAL) object. The *dncp* and *dnls* commands are the most commonly-used commands. They communicate with the subset of FAL operations supported by most remote nodes.

The *dncp* and *dnls* can be used with almost any node. The *dnrm* and *dnmv* commands require a facility that not all remote FALs possess. If the *dnrm* or *dnmv* program prints a message about unsupported operations, it is not likely that the remote node's FAL is capable of the requested operation. This should be verified with the system administrator of the remote node.

3.2 Remote File Specifications

A complete remote file specification consists of a node identifier (name or address), access control information and a file specification that must conform to the rules that apply on the remote system. The syntax is:

```
node"access-information"::file-spec
node::file-spec
```

Node

The node identifier must be a DECnet node name or address. The node name can be up to six characters long, and the node address must conform to the area-number.system-number format The node identifier must be followed by a double colon (::) if no access control information is specified.

Access-information

The access control information syntax is:

```
"user password account-id"
"user password"   (The account-id is optional.)
"user"            (The user is prompted for the password.)
```

Access control information establishes the user's privileges or the reading, writing, or executing files on a remote system. The account ID is not currently used by 4DDN, but can be used on some DECnet systems to indicate the party to be billed for network access time. The user name identifies the user on the remote system in whose name the access is to be performed. Password identifies the password that the indicated user would use to log in.

The access control information must be enclosed in double quotes with each access control word separated by a space. The remote file specification must appear directly after the double colon following the node identifiers. If the remote file specification's password is missing and this is an argument to a RFAS program, the program prompts the user for the password.

File-spec

The Remote File Specification must be entered in the same way that it would be entered by a user on the remote system. Also, the Remote File Specification must conform to the default environment for the user specified in the access control information. Wildcard characters can be used in the commands the same way as they would be used on the remote system.

EXAMPLE

```
dncp  'ginsu"rob boston"::/usr/rob/myfile1'  myfile1

        node      access          remote         local
        name      information     file-spec      destination
```

CAUTION: Single quotes (') must be used to enforce the literal interpretation of the file specification. If quotes are not used, then the shell might improperly interpret the file specification text. To ensure proper identification of embedded access control information, it must be surrounded by double quotes ("). (See the examples included in the description of the *dncp* command.)

3.2.1 Supplying Access Control Information

You can supply access control information in one of three ways:

1. You can include the access information within the remote file specification, as previously discussed.

   ```
   'node"access_info"::remote_file_specification'
   ```

   Access control information specified using this method takes precedence over the information specified through the other two methods.

2. This method enables you to specify the user, password, and account ID as command line options. The syntax for this method is:

   ```
   -u user -p password -a account 'node::remote_file_description'
   ```

   –u User: The argument following –u is the default username used to access all remote files on the command line.

   –p Password: The argument following –p is used as the default password for the specified user name. If –p is omitted, the user is prompted to enter the password with echoing characters turned off for security. A null password can be specified with –p " ".

   –a Account-ID: The argument following –a is the account string used to access all the files specified on the command line. Use –a "" to return to a no-account string later in the command line.

3. Set the environment variables NET_USER and NET_ACCOUNT using the appropriate shell command(s). These commands are used to preset the username and account string for all remote file accesses. For security reasons, a password cannot be specified with this method. The values of these environment variables are overridden when different access control information is included in the command line.

   NOTE: Passwords given on the command line are visible to the IRIX *ps*(1) command. Do not specify a password on the command line if you are concerned about security.

3.3 The *dncp* Command

The dncp command is used to:

- Copy a local file or group of files to a remote system;
- Copy a remote file or group of files to a local system;
- Copy a remote file or group of files to a remote system.

Command Syntax

```
dncp [-options] file1 file2
dncp [-options] file ... directory
```

where

options — Can be one or more of the following:

v	Verbatim
i	Interactive
l	Logging
n	Noisy
r	Release
t	Total

file1, file — Local or remote file specification for the source file to be copied. File1 is the source file specification to be copied. This file can contain wildcards and can refer to a local or remote file. File1 can not be a directory specification.

file2 — This file is the target file specification. This file specification must not include wildcard specifications. File2 can refer to a local or remote file.

directory — Local or remote target directory specification into which files are copied. A remote directory specification must include all trailing punctuation (everything up to the first letter of the filename that would normally follow the directory).

In addition, access control information can be supplied in any one of the three methods discussed in the section "Supplying Access Control Information." Specific examples are presented below.

3.3.1 Description

The *dncp* copies files between DECnet and 4DDN nodes. The file and pathname arguments can be either simple file specifications of local files or the more lengthy remote file specifications. (See the "Remote File Specifications" section.)

The standard input device, such as the keyboard, can be used in place of the source input file by substituting a – (hyphen) symbol. A standard output device, such as the display screen, can be used in place of the target file or target directory by substituting a – (hyphen) symbol.

When specifying multiple source input files, the target destination must be a remote or local directory or the standard output. The output files retain much of their original names. Some reduction in length by the destination node may be performed.

When remote files are copied to a target directory on a 4DDN node, their names are converted, if necessary, to names that are suitable for use on the IRIX system. Files are stripped of version numbers, and, if the files are from a non-case-sensitive system like VAX/VMS, they are converted to lower case, e.g., "SYS$SYSDEVICE:[LEE.PROJ1]MYFIL.RNO" becomes "myfil.rno" on IRIX. The names of files from another UNIX system (such as ULTRIX), or case-sensitive system, are unchanged.

Command options are used to modify commands. You can place the option characters anywhere on the command line and in any order. The –u, –a, and –p command options are exceptions as they affect filespecs appearing only to their right. The options can be grouped as several letters following a minus sign. The options are:

v Verbatim: All the input files are transferred byte for byte, without record format conversion and with no bytes lost, altered, or inserted. Output files are created with a record format appropriate to their byte-stream nature. On VMS, the output files always have FIXED RECORD format and NO RECORD attributes. When copying from one UNIX system to another UNIX system, the Verbatim mode increases copying speed.

i Interactive: Before each input file is copied, the user is prompted to confirm the operation by entering Y, y (yes), T, t, or 1 (true). Any other response skips the current file and proceeds to the next file. The interactive option is particularly useful in a selective transfer with wildcard specification.

l Logging Information: Logging information is printed on the standard output to indicate the start of data transfer. The logging message format is the following:

```
source-file =>  destination-file
```

n Noisy: A message is printed on the standard error stream indicating when there is an attempt to connect to FAL, the remote file transfer server. This often takes several seconds, and the message provides a way to monitor the operation.

r Release Number: This option is used to specify the release and revision numbers of *dncp* and its components. If the Release Number switch is the sole argument to *dncp*, *dncp* prints the release information and terminates.

t Total: The total number of bytes and files transferred are printed.

3.3.2 Diagnostics

Whenever *dncp* fails to complete the requested operation, diagnostic messages are printed on the standard output device. Generally, these messages are self-explanatory.

Also, when *dncp* is executed, a status value is returned to the shell. A status value equal to 0 means that the *dncp* command was successfully completed. Any other value indicates a failure.

3.3.3 Remarks

1. If you are copying a file to or from a system that supports global, or wildcard characters (e.g., * and ? on IRIX and ULTRIX), these characters can be used in your file specification. See the examples with wildcard specifications.
2. A new file is always created for the output file of a copy. On a VAX/VMS system, when a specified target file name already exists, a new version of the same file is created. On a UNIX system, when a specified target file name already exists, and is not write-protected, a new file is created and the existing file is deleted.
3. The protection mode of the output files is the default protection mode which is in effect for the accessing user, i.e., the remote user for remote files.
4. The creation and modification times are not copied from the input file. The creation and modification times for the output file are the time of the creation of the new file.
5. Do not try to copy multiple input files to a single output file. Multiple input files can only be copied to a directory or to the standard output device.
6. If a remote file specification cannot be opened, the file specification and a diagnostic message are output. If a password was specified, in the access control information, it is not displayed.
7. When copying from one UNIX-based system (e.g., IRIX and ULTRIX) to another UNIX-based system, use the verbatim option (–v). In a UNIX-to-UNIX copy, there is no need for the record format conversions that are performed in a non-verbatim copy. Eliminating the record format conversions increases the copying speed.
8. When copying files from a remote node to a remote node, the resulting files will have the same record attributes as if they had originated on the local system.

3.3.4 Sample *dncp* Usage

EXAMPLE

Copying a local IRIX file to a remote VAX/VMS file with embedded access control information:

Input:

```
dncp -ln /usr/max/bow.c  'gorgo"lee wiz"::[lee]test1.c'

            Local IRIX      Remote          Remote VAX
            filespec        node name       filespec
                            with user
                            and password
```

Output:

```
Waiting for response from FAL on node gorgo ... ok
/usr/max/bow.c => gorgo"lee"::[lee]test1.c;1
```

EXAMPLE

Copying a local IRIX file to a remote VAX/VMS file with access control information specified as options:

Input:

```
dncp -ln /usr/max/bow.c -u lee -p wiz 'gorgo::[lee]test1.c'

            Local IRIX      User and          Remote VAX
            filespec        password          filespec
```

Output:

```
Waiting for response from FAL on node gorgo ... ok
/usr/max/bow.c => gorgo"lee"::[lee]test1.c;1
```

EXAMPLE

Copying a local IRIX file to a remote VAX/VMS file with the user name specified by the environment variable:

Input:

```
csh: setenv NET_USER LEE
sh:  NET_USER=LEE ; export NET_USER

dncp -ln /usr/max/bow.c  'gorgo::[lee]test1.c'

          Local IRIX          Remote VAX
          filespec            filespec
```

Prompt:

```
Password [gorgo::lee]:    <User enters password with no echo>
```

Output:

```
Waiting for response from FAL on node gorgo ... ok
/usr/max/bow.c => gorgo"lee"::[lee]test1.c;1
```

EXAMPLE

Copying a VMS file to a local file:

Input:

```
dncp -u lee -p wiz   'gorgo::[lee]mad.exe'  /usr/lee/test -ln

       Access Info    Remote VAX filespec    UNIX target
                      specification
```

Output:

```
Waiting for response from FAL on node gorgo ... ok
gorgo"lee"::[lee]mad.exe => /usr/lee/test
```

EXAMPLE

Local wildcard to remote directory:

Input:

```
dncp /usr/max/*.c   'gorgo"lee wiz"::user:[lee]' -ln

      Local IRIX        Remote VAX/VMS Directory
      filespec          specification with access
                        information
```

Output:

```
Waiting for response from FAL on node gorgo ... ok
/usr/max/bow.c => gorgo::user:[lee]bow.c
/usr/max/yow.c => gorgo::user:[lee]yow.c
/usr/max/zow.c => gorgo::user:[lee]zow.c
```

EXAMPLE

Local wildcards to remote directory:

Input:

```
dncp -l   *.h  *.c      'gorgo"lee wiz"::user:[lee]'

         Local IRIX     Remote VAX/VMS Directory
         filespec       specification with access
                        information
```

Output:

```
foo.h =>  gorgo"lee"::[lee]foo.h
bar.h =>  gorgo"lee"::[lee]bar.h
bow.c =>  gorgo"lee"::[lee]bow.c
yow.c =>  gorgo"lee"::[lee]yow.c
```

EXAMPLE

Remote file to local directory (current directory):

Input:

```
dncp 'gorgo::[max]hello.c'       .        -lnr

        Remote filespec       Target specification:
                              current working dir.
```

Output:

```
dncp: Version 3.028 date 20-Jan-88 16:27:34
      NFARS: Version 3.064 date 20-Jan-88 16:30:00
      NFTL:  Version 3.040 date  11-Jan-88 16:33:11
      Build SGI 4DDN Release 1.0

Waiting for response from FAL on node gorgo ... ok
gorgo::[max]hello.c => ./hello.c
```

EXAMPLE

Remote file to redirected output piped to "more":

Input:

```
dncp -u lee 'gorgo::[max]hello.c'  - |  more

                Remote filespec   Standard output
                                  as destination
```

Prompt:

```
Password [gorgo::lee]:
```

Output:

```
main ( )
{
                printer ("Hello world\n");
}
```

EXAMPLE

Printing files on GORGO's line printer:
dncp uses redirected input from a pipe with *pr*:

Input:

```
pr -160 *.h *.c | dncp - gorgo::lca0:source.list -lt

                      Standard    Remote VAX/VMS
                      Input       Printer with Named
                                  Listing
```

Output:

```
STDIO => gorgo::lca0:source.list
1 file (32596 bytes) copied.
```

EXAMPLE

Remote file to local directory with password prompt:

Input:

```
dncp -u lee -ln 'gorgo::[lee]mad.exe'     bin

      Access      Remote VAX              Target
      Control     filespec                directory
                                          info
```

Prompt:

```
Password [gorgo::lee]:     <User enters password>
```

Output:

```
Waiting for response from FAL on node gorgo ... ok
gorgo"lee"::[lee]mad.exe => bin/mad.exe
```

EXAMPLE

Local file to remote file (IRIX -> RSX):

Input:

```
dncp bow.c 'gdzila"sqa test"::dr0:[210,201]test.c' -l

  Local IRIX          Remote RSX filespec with
  filespec            access information
```

Output:

```
bow.c  => gdzila"sqa"::dr0:[210,201]test.c
```

EXAMPLE

Remote file to local file (RSX -> IRIX):

Input:

```
dncp  'gdzila"sqa test"::dro:[210,201]test.c' .
```

EXAMPLE

Local file to remote file (IRIX -> IRIX or ULTRIX):

Input:

```
dncp -ln bow.c 'mutant"sqa test"::/u0/usr/sqa/test1.c'

     Local IRIX       Remote UNIX filespec
     filespec         with access information
```

Output:

```
Waiting for response from FAL on node mutant ... ok
bow.c => mutant"sqa"::/u0/usr/sqa/test1.c
```

EXAMPLE

Remote file to local file (IRIX or ULTRIX -> IRIX):

Input:

```
dncp -ln 'mutant::/u0/usr/sqa/test1.c' .
```

Output:

```
Waiting for response from FAL on node mutant ... ok
mutant::/u0/usr/sqa/test1.c => ./test1.c
```

3.3.5 *dncp* Error Messages

When an error occurs during a file transfer operation, a *dncp*, NFARS, or task-to-task error message is printed.

The *dncp* error messages are listed below. Each message includes an explanation and recommended action. For some error conditions, it may be necessary to consult the system manager. Other messages indicate software errors and these errors should be reported. If an error message is not listed in this section, then the error is an NFARS or a task-to-task communication error message. (See "NFARS Error Messages" in Chapter 4 and Appendix A, "4DDN ERROR CODES", for more information.)

There is one other source of *dncp* error messages. Operations on local files may cause local error messages to be printed. These messages are printed by calls to "perror". A complete explanation can be found in *intro*(2) manual page in the *IRIS-4D Programmer's Reference Manual.*

```
operation LOCAL to LOCAL not supported
operation STDIO to LOCAL not supported
operation LOCAL to STDIO not supported
operation STDIO to STDIO not supported
```

Meaning: Copying from the specified file type to the specified file type is not supported.

Recommended Action: Use *cp*(1), *cat*(1) or an editor.

```
source filespec cannot be a directory
```

Meaning: *dncp* determined that the source filespec is a directory and this is not permitted. The source of any transfer must be a file specification that results in a list of files. The destination specification may be a directory.

Recommended Action: Respecify the source as a list of files.

```
remote wildcards require a destination directory
```

Meaning: A destination directory is required when either multiple source files, or a source file specification including wildcards is specified. This restriction includes a wildcard file specification that results in a single file.

Recommended Action: Specify a directory as the destination.

```
Failed: <source> => <destination>
```

Meaning: This is a notification that the copying operation for the indicated file failed. The complete reason is explained by an earlier message.

```
can't delete old destination file
can't rename destination file
```

Meaning: The destination file already exists. A temporary working file is created to protect the existing data. Upon completing the transfer, the temporary file replaces the original file. These messages indicate that the original file cannot be deleted or that the temporary file cannot be renamed.

```
dncp: std-input and interactive are inconsistent
```

Meaning: The *dncp* command determined that the given command requires both interactive mode and input from standard input to be used at the same time. The *dncp* command does not simultaneously support these operations.

Recommended Action: Execute the command again without interactive mode or without std-input as a file. The most logical action is to omit the interactive mode.

```
dncp: stdin to stdout is not permitted
```

Meaning: The *dncp* command determined that both stdin and stdout would be used as the source and destination for a transfer. The *dncp* command does not support this operation.

Recommended Action: Use one of the operating systems text editors or similar facilities.

3.3.6 File Record Formats

The IRIX file format is often different from the file formats of non-UNIX systems on a DECnet Phase IV network. The *dncp* command recognizes these differences when copying a file to/from a IRIX system and a non-IRIX system.

IRIX files contain record structured character data, i.e., lines of text or source code. The end-of-record is delineated with a newline character (usually a line-feed or ASCII 10).

Files on VAM/VMS or other DEC operating systems can be stored in a variety of file formats, including several stream formats, variable and fixed record formats. (See Digital's *Guide to VAX/VMS File Applications* for more information on file formats.) Unlike UNIX files, DEC files are stored with information indicating the file's format and its record attributes (e.g., carriage control).

To compensate for these differences in file record format, the *dncp* command has two distinct modes of operation: the default conversion mode and the verbatim (or image mode, as it is often referred to in the VAX/VMS world.) The verbatim mode can be selected through an option (–v) in the command line.

When copying a file (in the default conversion mode) from the local IRIX system to a VAX/VMS system, the file created on VAX/VMS is always in the Variable Record format, with the Carriage Return Carriage Control record attribute.

In the default mode, *dncp* transfers data in line per record mode. This works for source code, etc., but causes certain types of files to become useless. The contents of each record (stripped of original record delimiting information) are written to the IRIX file with IRIX-style line delimiters.

In Verbatim mode, *dncp* copies the source file byte for byte to the destination. When copying a file verbatim mode from a IRIX system to a VAX/VMS system, the created VAX/VMS file is in Fixed Record Format with NO RECORD attributes. It is recommended that the Verbatim option be used for copying binary image files, such as object files or executable program images. The Verbatim option is not recommended for user text.

In reading under verbatim mode, a VMS Variable source file would be reproduced on the IRIX destination with the record delineation bytes of the VMS Variable type such as the 2 byte length and possibly a pad byte.

In this release, there is no capability of writing records of any type other than fixed-length 512-byte records (for verbatim) to all systems and Variable with carriage control to VMS systems or Stream-LF to other systems.

3.4 The *dnls* Command

The *dnls* command lists the contents of each remote directory name specified. If the name is a file specification, the files matching it are listed. The file specification must conform to the wildcard rules of the remote system. Both the remote file specification format and the methods of specifying access control information are the same as for *dncp*.

Command Syntax

```
dnls [-options] name ...
```

Where

options — Can be one or several of the following:

h	headers not printed
l	list in long format
s	print out size of file
c	time of creation
1	print out in one column
C	Print out in multi-column format
r	Print out release and version number
t	Sort by time
U	Time of last access

name — represents one or more directories or file specifications on the remote system.

3.4.1 Description

The *dnls* command lists the files for each remote directory specified. Optionally, you can include a file name, rather than a directory name, in the command line. In this case, *dnls* lists all the files that match the name you specify. The file specification must conform to the wildcard rules of the remote system.

The files are sorted alphabetically by name, as a default condition. The –t switch causes the sort to be performed by the time of last modification, with the latest files appearing first. The *dnls* command understands the following options:

h Headers that describe the directory that the following files belong to, are not printed (by default, they are printed).

l List files in long format. The protection mode, owner, size in bytes and time of last modification are listed along with the name of the file.

When the long output format is requested, the 12 character protection mode is printed showing 4 protection levels:

1. System, for the system user (a VMS concept)
2. Owner, for the owner of the file
3. Group, for users in the owner's (or the file's, on IRIX) group
4. World, for all other users

Each set of three characters specifies the privileges for the designated users to read, write and execute (or list, for a directory) a file. The privileges are as follows:

r if the file may be read.

w if the file may be written to

x if the file may be executed (or listed if a directory).

– for a permission not granted.

s The size of the file in 512-byte blocks is printed before the rest of each file's information.

c The time of creation is listed instead of the last modification time. When used together with the –t option, files will be sorted according to the time of creation.

U The time of last access is listed instead of the modification time. when used with the –t (Sort by time) option, files are sorted according to the time of last access.

1 Only one file entry is printed on each line; this is the default mode for long format or when the standard output is not a terminal.

C Multi-column listing format is used; this is the default when the output is to a terminal.

When dnls generates multi-column output, it checks the environment variable COLUMNS for the number of columns that can be displayed on the standard output device. (The default value is 80.) Then, *dnls* formats each line of the listing accordingly. (The *ls*(1) command also uses COLUMNS for this purpose.)

r Displays *dnls*' release and version levels.

t Sorting is performed by time instead of alphabetically.

3.4.2 Remarks

1. The file listing in each separate directory are preceded by the line "Directory NODE::remote-directory" to identify the resolved pathname of the listed directory. A blank line separates directory groups in one listing.
2. By default, file names are sorted alphabetically, in ASCII order. When files containing embedded numbers such as version or generation numbers, then the alphabetical listing can be different from the natural listing order on the original system.
3. All output is written to the standard output device except for some diagnostic messages, which are sent to the standard error stream.
4. The "system" protection bits are only meaningful if the system, which the directory list comes from, maintains system protections. If the remote system is UNIX-based, then the protection bits have whatever value the UNIX FAL assigns to system protection.

5. The value shown for the size of a file on a remote system is the number of bytes allocated to store it in its own record format. If the file is then copied to a UNIX system with *dncp*, the size of the resulting file may differ from the size shown by *dnls*, if a record format conversion is executed.

3.4.3 Sample *dnls* Usage

EXAMPLE

Remote directory listing of a VAX/VMS node:

Input:

```
dnls 'gorgo"lee wiz"::[lee]'
```

Output:

```
DIRECTORY GORGO::DUA1:[LEE]

TEST1.C;2     JFEP.TXT;11     DQU.RNO;1
TEST1.C;1     TEST2.C;1
```

EXAMPLE

Remote directory listing of a VAX/VMS node using the –l (long format) option:

Input:

```
dnls -l -u lee 'gorgo::[lee]'
```

Prompt:

```
PASSWORD: [gorgo::lee]:
```

Output:

```
DIRECTORY GORGO::DUA1:[LEE]

rwxrwxr-xr-- [200,134]        3190  03-MAR-86  14:22  CNTRCHART.RNO;1
rwxrwxr-xr-- [200,134]        7728  03-MAR-86  14:17  DUMPCH3.XXX;3
rwxrwxr-xr-- [200,134]        4090  03-MAR-86  14:17  DUMPCH4.XXX;4

Protections      UIC          Bytes   Modification       Filename
             (User ID Code)           Date     Time
```

EXAMPLE

Remote directory using wildcard and release, filesize and sort by time options of users login directory:

Input:

```
dnls -srt 'gorgo"lee" wiz"::n*'
```

Output:

```
dnls: version 3.005 date 01-Dec-87 11:09:00
    NFARS version 3.064 date 20-Jan-88 16:30:00
    NFTL  version 3.040 date 11-Jan-88 16:33:11
    Build SGI 4DDN Release 1.0

DIRECTORY GORGO::DUA1:[LEE]

    2 NETSERVER.LOG;48
    3 NETSERVER.LOG;47
    2 NETSERVER.LOG;46
    4 NFARS_INTRO.DOC;2
```

EXAMPLE

dnls command of remote ULTRIX directory:

Input:

```
dnls 'mutant"sqa test"::/u0/usr/sqa/'
```

EXAMPLE

dnls command of remote RSX directory:

Input:

```
dnls 'gdzila"sqa test"::dr0:[210,201]'
```

3.5 The *dnrm* Command

The *dnrm* command is used to delete specified remote files. Both the remote file specification and the methods of specifying access control information are the same as for *dncp*.

Command Syntax

```
dnrm [-options] filespec
```

Where

options — Can be one or several of the following:

- i — interactive
- l — logging
- n — print progress
- r — print version information

filespec — Remote file specification of file(s) to be deleted

3.5.1 Description

i — Interactive - The user is prompted to confirm each file deletion by entering one of the following responses:

- Y or y — Delete the file and continue the interactive file deletion mode.
- N or n — Do not delete the file and continue the interactive file deletion mode.
- R or r — Delete the file and all the remaining files. This terminates the interactive file deletion mode.
- Q or q — Do not delete the file and terminate the interactive deletion mode.

l Logging information: Following the deletion of a remote file, an acknowledgement is printed on the standard output terminal in the form:

```
<node>::<filespec> removed
```

n Noisy: A message is printed on the standard error stream indicating when there is an attempt to connect to FAL, the remote file transfer server. This often takes several seconds, and the message provides a way to monitor the operation.

r Release Number: This option is used to specify the release and revision numbers of *dnrm* and its components. The release option, by itself, simply displays the revision numbers.

3.5.2 Error Messages

When an error occurs during a *dnrm* command execution an error message in the form shown below is displayed:

```
node::<file spec> not removed: <error description>
```

The error message is displayed, even if the logging option was not selected.

NOTE: Support for this command may not be provided by the remote node's FAL. This would be indicated by a message that indicates the words "unsupported operation". This does not constitute a problem with *dnrm*.

3.5.3 Sample *dnrm* Usage

EXAMPLE

Deleting a file on a remote VAX/VMS system:

Input:

```
dnrm 'gorgo"sqa"::[sqa]test.obj'
```

Prompt:

```
Password [gorgo::sqa]
```

EXAMPLE

Deleting files on a VAX/VMS remote system using wildcard notation and specifying access control information:

Input:

```
dnrm -u sqa -p test 'gorgo::[sqa]*.obj'
```

EXAMPLE

Deleting files on a VAX/VMS remote system using wildcard notation with interaction and logging options:

Input:

```
dnrm -l -i 'gorgo"sqa test"::[sqa]*.obj'
```

Interaction

```
remove gorgo::[sqa]test1.obj (y/n/r/q) ? y
gorgo::[sqa]test1.obj removed
remove gorgo::[sqa]test2.obj (y/n/r/q) ? n
gorgo::[sqa]test1.obj not removed
remove gorgo::[sqa]test3.obj (y/n/r/q) ? r
gorgo::[sqa]test4.obj removed
gorgo::[sqa]test5.obj removed
```

EXAMPLE

Deleting multiple files on two different VAX/VMS nodes:

Input:

```
dnrm   'gorgo"sqa1 test1"::[sqa1]*.obj'
                -u sqa2 -p test2  'galaxy::[sqa2]test.*'
```

EXAMPLE

Deleting a file from an IRIX or ULTRIX node:

Input:

```
dnrm 'mutant"sqa test"::/u0/usr/sqa/hello.o' -l
```

Output:

```
mutant"sqa"::/u0/usr/sqa/hello.o removed
```

EXAMPLE

Deleting a file from a RSX node:

Input:

```
dnrm  -l 'gdzila"sqa test"::dr0:[210,201]hello.obj'
```

Output:

```
gdzila"sqa"::dr0:[210,201]hello.obj removed
```

3.6 The *dnmv* Command

The *dnmv* command is used to move files from one directory to another directory. This command also renames the moved file or a group of files. Both the remote file specification and the methods of specifying access control information are the same as for *dncp*.

Command Syntax

```
dnmv [-options] source-filespec destination-filespec
dnmv [-options] file1 file2 ... destination-filespec
```

Where

options Can be one or several of the following:

- i interactive
- l logging of activity
- n print progress
- r print release number

3.6.1 Description

The *dnmv* command moves a file or a group of files. When moving a file, two filespecs are required: the source and the destination. There is a choice of moving the filename, directory or both. When moving a group of files, specify a filespec for each of the source files and then specify the destination filespec. A wildcard option can be used in the filespec to move more than one file to a new directory. Each of the files is moved to the new directory listed in the destination filespec.

The *dnmv* command understands the following options:

i Interactive: Before each input file is copied, the user is prompted to confirm the operation by entering Y, y (yes), T, t, or 1 (true). Any other response skips the current file and proceeds to the next file. The interactive option is particularly useful in a selective move with wildcard specification.

l Logging: When the move is successful, an acknowledgement is displayed on the screen for each file moved.

n Noisy: A message is printed on the standard error stream indicating when there is an attempt to connect to FAL, the remote file transfer server. This often takes several seconds, and this message provides a way to monitor the operation.

r Release Number: This option is used to specify the release and revision numbers of dnmv and its components. The release number option, by itself, simply displays the revision numbers.

source-filespec
This is the complete file specification for the file that is going to be moved.

destination-filespec
This is the complete file specification for the destination file.

3.6.2 Error Messages

When an error occurs during a dnmv command execution an error message in the form shown below is displayed:

```
node::<file spec> <error description>
```

The error message is displayed, even if the logging option was not selected.

NOTE: Support for this command may not be provided by the remote node's FAL. This would be indicated by a message that indicates the words "unsupported operation". This does not constitute a problem with *dnmv*.

3.6.3 Sample *dnmv* Usage

EXAMPLE

Using the VAX node name in the first filespec:

Input:

```
dnmv 'gorgo::dual:[dir1]file1.tst'  'dual:[dir1]file2.tst'
```

EXAMPLE

Using the VAX node name in both filespecs:

Input:

```
dnmv 'gorgo::dual:[dir1]file1.tst'
     'gorgo::dual:[dir1]file2.tst'
```

EXAMPLE

Renaming files from two directories (dir1) and (dir2) to another directory (dir3):

Input:

```
dnmv 'gorgo::dual:[dir1]file1.tst'
     'dual:[dir2]file2.tst' '[dir3]'
```

EXAMPLE

Using the username option with a remote UNIX node:

Input:

```
dnmv -u sqa quaser::/u0/user/file1 /u0/user2
```

EXAMPLE

Renaming files on a UNIX node:

Input:

```
dnmv mutant::/u0/user1/file1 /u0/user2/file2 -l
```

Output:

```
moved mutant::/u0/user1/file1 to /u0/user2/file2
```

EXAMPLE

Wildcard renaming on a VMS node:

Input:

```
dnmv  'gorgo::dual:[dir1]file*.tst' 'dual:[dir2]'
```

3.7 Remote File Access from a VAX/VMS Node

The VAX/VMS COPY command is used on a VAX/VMS node to copy a file locally on the VAX/VMS node or remotely across the network.

The VAX/VMS DIRECTORY command is used on a VAX/VMS node to list the contents of a directory on the VAX/VMS node or on a remote node on the network.

When copying to or from a 4DDN node, you must substitute the VAX/VMS file specification format for the IRIX file specification format whenever you use wildcards in the file specification. For example, when copying a file from a remote 4DDN node named quasar to a VAX/VMS node, the recommended file specification format is as follows:

```
$ COPY quasar::usr[sam.sub]*.*  (note VMS format)
```

rather than the usual UNIX format:

```
$ COPY quasar::"/usr/sam/sub/*"
```

When listing the contents of a directory named */usr/sam/memos* on a remote 4DDN node named quasar from a VAX/VMS node, and wildcards are used, it is recommended that you use VAX/VMS syntax, e.g.,

```
$ DIR quasar::usr:[sam.memos]
```

Otherwise, you must explicitly specify the wildcards, e.g.,

```
$ DIR quasar::"/usr/sam/memos/*"
```

NOTE: When indicating a file that resides in the root directory (/) such as */unix*, the 'device' part of the name is root so */unix* is *root:[]unix*. Other examples are:

IRIX	VMS
/etc/passwd	etc:[]passwd
/usr/bin/more	usr:[bin]more
/d/baa/src/test.c	d:[baa.src]test.c
/unix	root:[]unix

LIMITATIONS

1. No case distinction can be realized with VMS.
2. No dots may appear in any directory component. For example, */u0/baa/rfas.doc/doc.n* is illegal.
3. Only one dot may appear in the filename; two are rarely possible. For example, "a.b.c.d" is illegal.

The limitations are deeply rooted within the 'design' of VMS. They cannot be fixed by altering 4DDN.

4. Network File Access for User Programs

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

The material presented in the chapter is for the experienced user who is comfortable with the C programming language and can utilize the IRIX primitive I/O operations: open, read, write, and close.

4.1 Network File Access Routines (NFARS) - Description

The programmer's interface to the Remote File Access System is through a set of routines called the Network File Access Routines (NFARS). The NFARS are included in an object-code library, with which application programs can be built. (The RFAS programs *dncp*, *dnls*, *dnmv*, and *dnrm* are all built using the NFARS facilities described in this chapter.)

The NFARS provides the following functions:

- Open or create a remote file
- Write data to the currently open file
- Read data from the currently open file
- Delete a remote file
- Rename a remote file
- Display the cause of an error encountered in an NFARS call

The following subsections provide a description of the NFARS facility. The last two subsections of this section describe the include files and libraries and how to involve them. These sections are followed by reference materials for the individual routines.

4.1.1 Opening and Creating Files

Remote file access is initiated with the *net_open* function. 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.

A network file descriptor is a small, non-negative integer value that is used by the NFARS to reference an open network stream. This file descriptor is similar to the IRIX local file descriptor returned by the open and create system calls, and should NEVER be used in place of it.

4.1.2 Reading and Writing to Opened Files

Data is transferred to or from remote files with the *net_read* and *net_write* functions. These functions are used the same way as the IRIX basic I/O *read* and *write* functions. A remote file opened for reading only supports the *net_read* function, and a file created or opened for writing supports only the *net_write* function; this is a restriction imposed by the subset of the DAP protocol that is used to implement the NFARS, and is also an IRIX restriction on files opened by the *net_open* function. Calls to *net_read* and *net_write* functions may not be intermixed on the same file descriptor as they may in certain cases on the IRIX file system when a file is opened for both read and write.

Data access to a file opened without the RFM_VERBATIM option is line oriented. The data obtained with a call to *net_read* is one or more lines of text. A line is merely a collection of characters (including control characters) terminated by the local line terminator. On IRIX, this line terminator is the line-feed known in IRIX parlance as a 'newline'. Any fragments from a block of data read from the remote file but not delivered to the user are stored for combination with later incoming records and subsequent delivery to the user.

The caller of *net_read* receives a sequence of lines resembling the source file. Data given to *net_write* is written to the remote file a line at a time. Each line of source data is placed into a separate record for transport. The DAP transport is heavily record oriented. This forms the basic reason for the limitations of the capabilities of NFARS. These limitations include file record format selection, no seeking, etc.

Any fragments of lines given to *net_write* are left for combination with subsequent *net_writes*. Thus, partial lines may be given to *net_write*. If the file was opened for writing, the resultant file is proper for a file of user text on the remote system. The file is line oriented. On VMS, this means Variable record with Carriage Return Carriage Control. On IRIX, this concept is basically meaningless except that the file is a set of lines of text.

If the RFM_VERBATIM option is specified when a file is opened, the line orientation of NFARS is disengaged. Data may be written to *net_write* much as it is to write, without regard to structure or line orientation. Data obtained from *net_read* is merely data; it is not line oriented.

NFARS reads and writes (*net_read* and *net_write*) should generally be of smaller sizes than those that IRIX may be capable of dealing with. It is recommended that the largest size of a read or write be restricted to 512 bytes. Larger reads or writes are strongly discouraged. The *dncp* program uses ONLY reads and writes of 512 bytes.

When writing programs that use NFARS library, be aware of the differences between IRIX and VAX/VMS record formats. These differences are discussed in "The *dncp* Command" section in Chapter 3.

4.1.3 Closing Remote Files

When a network file descriptor is no longer needed, it should be closed immediately, with *net_close*, to release the network resources it occupies. All of the network files left open when an NFARS application exits (or is terminated) automatically close, but it is desirable to use *net_close* on remote files opened for writing to prevent the loss of data. If a file opened for writing is not closed with *net_close*, data is lost.

4.1.4 Deleting Remote Files

Remote files may be deleted with a call to the *net_delete* function. It should be noted that there may be restrictions on what can be deleted applied by the remote node. Also certain access rules will be applied by the REMOTE NODE. Some remote nodes are not capable of supporting this feature. The user might consult the section of Network File Access Commands that covers the *dnrm* program. It should be noted that the *dnrm* command supports wildcards while the *net_delete* function is not represented as supporting wildcards. A wildcard call of *net_delete* may work.

This function has no formal relationship with the *net_open* function. A file does NOT have to be *net_open*'ed before deleting. In fact this function should never be called on an opened file.

4.1.5 Renaming Remote Files

Remote files may be renamed with a call to *net_rename*. It should be noted that the two filespecs MUST be on the same node. There may be restrictions about the renaming of files on the same node. The restrictions deal with renaming across devices and access control. The user may consult the section of the Network File Access Commands chapter that describes the *dnmv* program. It should be noted that the *dnmv* command supports wildcards while the *net_rename* function is not stated as supporting wildcards.

This function has no formal relationship with the *net_open* function. A file does NOT have to have been opened before renaming. It is better not to have the file open when it is renamed.

4.1.6 Wildcard Name Expansion

The wildcard name expansion facility provides the identification of files that match specified naming requirements. The names are specified by a wildcard specification, i.e., a filespec with or without wildcards. This facility is initiated by a call to the *net_find* routine. Each name is retrieved using repeated calls of the *net_fnext* routine. If the wildcard list must be aborted or terminated, the *net_fstop* call should be issued.

The name is stored in its three components filename, directory, and volume. If requested, the user may also request file attributes to be collected. The attributes are placed in a "File_attr" structure, which is described at the end of the reference section.

4.1.7 NFARS Error Handling

It is possible that an NFARS routine may fail to perform its operation. The value returned by the 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* routine can be called to print a descriptive message based on this error code. It may also include a user-defined string that can identify the application program that generated the message. The NFARS error messages are listed at the end of this chapter.

4.1.8 Header Files

Any C program making use of the NFARS should contain the following line before any NFARS references:

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

There is another header file that may be included: the *nferror.h* file. The *nferror.h* file contains the symbolic constants corresponding to the possible error codes. The comments near the values are similar to the messages printed by *net_perror*. Include this file to check the values of *nfars_errno*. It is included with:

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

A third file is required to understand the attributes of files discovered by using the wildcard expansion system based upon *net_find* and *net_fnext*. It may be included with:

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

4.1.9 Linking to the NFARS Library

The NFARS routines 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:

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

4.2 Network File Access Routines (NFARS) – Reference

This section is a reference for the supported NFARS routines. At the end of this section, the wildcard file attribute structure is explained.

4.2.1 *net_open* – Opening a Remote File

The *net_open* function opens a file on a remote DECnet node.

Call Synopsis

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

int
net_open(filespec, options, attrib)
    char        *filespec;
    int         options;
    File_attr   *attrib;
```

Description

The *net_open* function initiates a logical link to the remote node and opens the file at the remote node. The logical link remains connected for subsequent NFARS calls until the file is closed. At which time, the logical link is marked idle, to be reused during the session. (In a later version of 4DDN, a mechanism may be supported that permits complete closure of a logical link.) This call can be used to open existing files or create new ones on remote systems.

File Spec

The filespec argument points to a null-terminated string, which is the complete network file specification of the remote file to be opened. The format is described in the "Remote File Specifications" in Chapter 3. If access controls are used, they must be embedded in double quotes after the node name. NFARS routines have no ability to ask the user for the password nor to incorporate environment variables. The filespec must provide ALL access information.

Options

The options argument determines if the remote file is to be opened for reading, writing, or creation. Possible values are:

RFO_READ — opens an existing file on a remote system for read access only.

RFO_WRITE — opens an existing file on a remote system for writing only and overwrites the existing data.

RFO_CREATE — creates a new file on the remote system and opens it for writing only.

These values are defined in the *<dn/nfars.h>* header file. Only one of these values should be assigned to options. They may NOT be ORed. If they are ORed, the resultant operation is most likely either write or create.

Additionally, the RFM_VERBATIM option may be ORed in using the bitwise operator (|). RFM_VERBATIM performs a byte-for-byte file transfer on subsequent read or write, operations. If RFM_VERBATIM is ORed with RFO_CREATE, the file created has a record format appropriate for byte-stream access. On VAX/VMS systems a file created in Verbatim Mode has Fixed Record Format, No Record Attributes, and 512 byte-records.

Attrib

Attrib is a pointer to a structure describing the file. It is only used when the RFO_CREATE function is selected. If the file is open for a read or write, specify *(File_attr *) 0*. (NOTE: This release of NFARS does not use mode to determine the protection of remote files. It uses the default protection of the remote user, so mode may always be specified as *(File_attr *) 0*. This is provided for upward compatibility.)

Results

Upon successful completion, *net_open* returns a valid network file descriptor, that is, a small, non-negative number. The network file descriptor is used for subsequent *net_read*, *net_write*, and *net_close* operations on the specified file. The value is analogous to the file description returned by a call to *open*. Beware as they are not interchangeable nor are they directly related; DO NOT mix them.

If the *net_open* call fails, a negative value is returned and *nfars_errno* is set to indicate the cause of the failure.

4.2.2 *net_read* – Reading From a Remote File

The *net_read* function reads from an open remote file.

Call Synopsis

```
int
net_read(nfd, buffer, size)
    int  nfd;
    char *buffer;
    int  size;
```

Description

The *net_read* function attempts to read a number of bytes of data (specified by the size argument) from the remote file identified by *nfd*, the network file descriptor.

It returns the number of bytes actually read, which may be less than the value specified by size. A return of 0 indicates that the end of the remote file has been reached. A return of less than 0 indicates that an error occurred and *nfars_errno* is set to the appropriate error code.

nfd is an NFARS network file descriptor returned from a successful call to *net_open.*

Buffer is a pointer to the area of memory where the data read from the remote file will be placed. It should have at least the number of bytes specified by the size argument.

Size specifies the maximum number of bytes to read. It is recommended that all reads be less than or equal to 512 bytes in size.

Unless the RFM_VERBATIM bit was set in the options word when the file was opened, record-formatted data is returned by *net_read* as UNIX stream format data, with newlines (\n) marking the end of each record. Note that *net_read* returns an arbitrary segment of stream data that is not related to the record structure of the remote file. If RFM_VERBATIM was set, *net_read* returns the bytes of the remote file in the original record format, without interpretation. Certain files read without verbatim result in error and *nfars_errno* being set to NFEBADDATA. These files may be read with verbatim.

Results

There is no guarantee that the number of bytes specified in size are read. Upon successful completion, *net_read* returns the number of characters actually read, which is no more than the value of size, but may be less.

net_read always returns 0, when the end of file has been reached.

If there is an error, –1 is returned and *nfars_errno* is set to the appropriate error code. When an error occurs or end-of-file is reached, the stream should be closed with *net_close*.

4.2.3 *net_write* – Writing to a Remote File

The *net_write* function is used to write to a remote file.

Call Synopsis

```
int
net_write(nfd, buffer, size)
    int  nfd;
    char *buffer;
    int  size;
```

Description

The *net_write* function writes the number of bytes of data specified in size from the specified buffer to the remote file. *net_write* returns the number of characters actually written.

nfd is an NFARS file descriptor returned from a successful *net_open.*

Buffer points to the beginning of the area of memory containing the data to be written.

Size is the number of bytes to be written. It should be greater than zero. It is recommended that all *net_writes* be less than or equal to 512 bytes in size.

Unless RFM_VERBATIM is set, data being written is assumed to be in the UNIX canonical record format and is converted into a format suitable for the remote system. Care should be taken to give data to *net_write* that is correct for the mode of file operation with respect to RFM_VERBATIM. If non line oriented data is written and RFM_VERBATIM is NOT set, an error occurs and *nfars_errno* is set to NFEBADDATA.

Results

The *net_write* function returns the number of characters actually written. This may not be the same as the value of size. If there is an error, −1 is returned. The variable *nfars_errno* is set to the appropriate error code to indicate the cause of the failure. When an error occurs, the stream should then be closed with *net_close*.

4.2.4 *net_close* – Closing a Remote File

The *net_close* function is used to close a remote file.

Call Synopsis

```
int
net_close(nfd)
    int nfd;
```

Description

The *net_close* function is used to close a remote file. It forces the completion of all pending write operations and frees the logical link initiated by *net_open*. Explicitly close an opened remote file using *net_close*. Failure to do so could result in a loss of data written to the file.

nfd is any NFARS network file descriptor returned by *net_open*.

Results

Upon successful completion, *net_close* returns 0. If there is an error, a non-zero value is returned and the variable *nfars_errno* is set to indicate the cause of the failure. After a failure, the descriptor can be presumed closed and should not be used with any other NFARS calls.

The *net_close* function should be called only on OPEN network file descriptors. Hence, *net_close* should never be called more than once on the same descriptor.

4.2.5 *net_perror* – Printing Error Messages

The *net_perror* function prints an informative error message.

Call Synopsis

```
net_perror(string)
    char *string;
```

Description

The *net_perror* function displays a meaningful message explaining the last error that occurred in an NFARS call. The *net_perror* function writes one line to the standard error file, which is usually the terminal. This line consists of the indicated string followed by a colon and the appropriate NFARS error message. The message is written to the standard error stream, *stderr*.

This function makes no effort to deal with long strings that may cause the line to 'wrap the screen'.

Results

The *net_perror* function has no return value.

EXAMPLE:

```
net_perror("util");
```

DISPLAY:

```
util: access permission violation
```

4.2.6 *net_delete* – Deleting Remote Files

The *net_delete* function deletes remote file(s).

Call Synopsis

```
int
net_delete(filespec)
    char *filespec;
```

Description

The filespec is identical to the filespec in the *net_open* function. There are no other arguments. The caller must be aware that not all remote FALs support the DAP delete file command. This function should not be invoked with wildcards.

Arguments

```
filespec - remote filespec to be deleted
```

Return Values

```
0         - if the file is removed
non-zero - if the file is not removed (see nfars_errno)
```

4.2.7 *net_rename* – Renaming a Remote File

The *net_rename* function renames a remote file.

Call Synopsis

```
int
net_rename (old_filespec, new_filespec)
    char *old_filespec;
    char *new_filespec;
```

Description

The *net_rename* function renames remote files. It takes two filespecs. The first filespec must be a remote filespec; the second filespec may be remote, but it specifies the new filename. The caller must be aware that not all FALs support the DAP rename file command. The two remote filespecs MUST be on the SAME system. This function should not be invoked with wildcards.

Arguments

```
old_filespec - original remote filespec
new_filespec - the new name of the remote file
```

Return Values

```
0         - if the value is renamed
non-zero - if the file is not renamed (see nfars_errno)
```

Limitations

This function may fail for a number of different non-protocol reasons. On some systems that communicate using DAP, the rules concerning the 'rename' of a file may differ substantially. For example, it may be impossible to 'rename' a file to another directory. It may also be impossible to 'rename' a file from one device to another device.

4.2.8 *net_find* – Wildcard Name Expansion

This function is used to initiate the wildcard name expansion.

Call Synopsis

```
int
net_find (filespec, attributes)
    char *filespec;
    int  attributes;
```

Description

This function MUST be sent to start the wildcard name expansion. Its primary mission is to send the remote wildcard name. The wildcard filespec is the first argument and conforms to the format of the 'filespecs' section. The second argument is the specification of the set of information about each file, called 'attributes' such as file size, protection, and access dates that the remote host is requested to provide. If successful, this function returns a "network wildcard descriptor", which is a descriptor similar to the *nfd*. It is used by *net_fnext* and *net_fstop* to refer to this wildcard expansion.

Arguments

```
filespec   - remote wildcard spec
attributes - attributes desired (see below)
```

The attributes field is a bitmask that is used to request information of the remote host about each file. It MUST be provided at the time of the *net_find* call.

```
RFA_NOATTRIBUTES - fetch no file attributes
RFA_ATTRIBUTES   - file type and size
RFA_PROTECTION   - file protection
RFA_DATE         - file access, modification dates
```

NOTE: RFA_NOATTRIBUTES must be used alone while the other three may be ORed to produce the desired information. The number of attributes affects directly the information about each file. It also affects the speed of wildcard expansion.

Return Values

```
nwd   - network wildcard descriptor (like nfd)
-1    - some error occurred (see nfars_errno)
```

4.2.9 *net_fnext* – Wildcard Name Retriever

This function retrieves the wildcard name generated in response to a previous *net_find* call.

Call Synopsis

```
int
net_fnext (nwd, name_p, vol_p, dir_p, attrib_p)
    int  nwd;
    char **name_p;
    char **vol_p;
    char **dir_p;
    File_attr  **attrib_p;
```

Description

This function is called to retrieve the name of an individual file from a wildcard expansion. It can be called ONLY AFTER a *net_find* was called successfully. This function allocates space to contain the various components of the generated filespec and the attributes structure. The information requested by the attributes field of *net_find* is placed in a structure allocated for the user.

Arguments

```
nwd      - network wildcard descriptor
name_p   - address of a pointer to a buffer for the name.
vol_p    - address of a pointer to a buffer for the volume.
dir_p    - address of a pointer to a buffer for the directory.
attrib_p - address of a pointer to a buffer for the file
           attribute structure.
```

Return Values

```
0         - the name is present.
non-zero  - if the name is not present, or an error occurred, or
            end of list.
```

4.2.10 *net_fstop* – Aborting a Wildcard Expansion

Call Synopsis

```
int
net_fstop(nwd)
    int nwd;
```

Description

If a wildcard expansion must be aborted for any programmatic reason, the *net_fstop* call should be issued. It aborts the current expansion and renders the DAP link to a neutral state.

Arguments

```
nwd - network wildcard descriptor (from net_find)
```

Return Value

The *net_fstop* function has no return value.

4.2.11 Wildcard Expansion Structure

The wildcard expansion operation may be directed to produce an attributes structure about each file generated by wildcard expansion. The following C preprocessor directive includes the definition for the attributes structure.

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

It must be noted that the values within the structure are filled in with information the caller of *net_find* requests. Therefore, if the date information is necessary, RFA_DATE must be specified, etc. This section deals with material that is oriented to VMS. The structure follows:

```
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;
```

The times are stored in a format that is identical to the return of a call to the IRIX *time*(2) system call. Thus it may be involved in a call to ctime to print the date in a pleasing local format.

The *fa_owner* element is a pointer to a string that contains a sequence of characters, generated by the remote system, that describe the owner of the file. On VMS this is a set of numbers, 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 that particular entity. It is similar to IRIX with the addition of the "system" entity, which is similar to the operator concept on some systems. Each byte of permissions may be tested with the following masks to test for access. 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 and not the directory.

The *fa_org* element contains a code that may be tested to determine the files organization. The organizations are specific to VMS. Only Sequential files may be transferred using NFARS. Only one of these values is possible. The test symbols are:

```
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 may be considered to be 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 may be considered to be 512.

The *fa_rfm* element contains a code that may be tested to determine the record format of the file. Only one of the following symbols may 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 it may be assumed to be 512 bytes.

The *fa_ffb* element contains the number of the first free byte in the last block. The number of bytes in the file is computed using this element by the following: $size = (ebk - 1) * bls + ffb$.

The *fa_rat* element contains some additional record attributes. The symbols that follow may be combined 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 it is the number of the last block.

4.3 NFARS Error Messages

Network File Access Routines (NFARS) are the building blocks used in the construction of RFAS programs such as *dncp*, *dnls*, etc. This section lists the error messages observed by NFARS.

Each NFARS error message listed below includes an explanation and recommended action. For some error conditions, it may be necessary to consult the system manager. Other messages indicate software errors and these errors should be reported. If an error message is not included in this section, then the error message is a task-to-task error message. (See Appendix A, 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 following: 4DDN software product version (*dncp* -r); remote node's vendor name and operating system version, i.e., DEC VAX/VMS 4.6 etc.; 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 with use of an NFARS routine, please include source code for sufficient analysis.

```
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 following: 4DDN software product version (*dncp* -r); remote node's vendor name and operating system version, i.e., DEC VAX/VMS 4.6 etc.; 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 with use of NFARS, please indicate source code for sufficient analysis.

```
invalid wildcard operation
```

Meaning: This message indicates that a call to a function capable of performing wildcard operations was given an invalid wildcard operation. The system on which the function was called cannot perform the wildcard operation.

Recommended Action: Attempt the operation again using a single file operation. Note: this message does not occur from 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 do not occur from the RFAS commands *dncp*, *dnls*, *dnrm*, or *dnmv*.

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 (directory, device, etc.), 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 obtained after 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 problems are encountered 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. There may be restrictions about renaming across devices on that node. Since renaming may be specified with two sets of access control information for a given operation, these sets must be identical.

```
file not found
```

Meaning: This message originates from any NFARS routine that uses a filespec as an argument and 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 a 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 are the result of an access to a file being denied with either 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 may 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 is an informative message 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, i.e., 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 the verbatim mode to transfer the data since it is binary information and not line oriented.

5. Task-To-Task Communication

IRIS-4DDN's task-to-task communication services support the exchange of data between processes on different nodes. The same program interface used by 4DDN's own network services, virtual terminal, remote file access, and network management, is made available to user applications. The advantage of this program interface is that the users need not be concerned with, or even aware of, lower-level network details such as topology, transmission sharing techniques, or type of communication linkage (e.g., local versus long-distance). 4DDN provides appropriate mechanisms to ensure reliable, effective communication between tasks, regardless of their location in the network.

This chapter contains a description of the logical link, explains the procedure to establish, maintain, and terminate a logical link.

5.1 Logical Links

Processes that run on different nodes and exchange data are connected by logical links. Logical links are temporary software data paths established between two communicating processes in a 4DDN network. The exchange of data between two processes over a logical link is called task-to-task communication.

5.1.1 Client and Server

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 inform the other process that it wishes to communicate with it.

The process that requests the connection is called the client. The other process is called the server.

The server must first inform the network software that it wants to be a server. The client supplies the server with access control information so it can decide whether or not to accept the logical link. (See "Establishing a Logical Link: Client" below.) The server can accept or reject the request.

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

5.1.2 Exchange of Data

After establishing the logical link connection, the two processes can exchange data. In addition, they can transmit interrupt data. Interrupt data is special high-priority information that is transmitted immediately.

Prior to the exchange of data, the I/O data format and the input mode are selected.

4DDN supports two I/O data formats, STREAM and RECORD.

- 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 only receives 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 if the buffer contains the beginning, the middle, the end of a message, or a complete message. RECORD format allows applications to perform their own data segmentation.

The two input modes are 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.

5.1.3 Multiple Concurrent Logical Links

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. These decisions are application dependent.

5.2 Establishing, Using, and Terminating a Logical Link

Logical links are established through the standard IRIX I/O calls (*open*(), *close*(), *read*(), and *write*()) and *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, calls must be issued to the 4DDN logical link device.

5.2.1 Establishing a Logical Link: Client

The client must first issue an *open*(), which activates the logical link device (called */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. The *open*() must be specified for each logical link that is to be established.

Once a file descriptor has been obtained by the client, the connect request is made by passing access control information to the server. This information identifies the server process and the client. It is sent by issuing an *ioctl*() request. The server process must be available for connection at the time the request is made.

After reception, the server may either accept or reject the link request. 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 was rejected, the status indicates a failure and the client must free the file descriptor by issuing a close. If the link was accepted, the status indicates success. The logical link is then established and the exchange of data can take place.

5.2.2 Registering 4DDN Processes as Servers

A 4DDN server process can be coded to start automatically or explicitly. Automatically started servers are those started by the dnserver process, a process that registers itself for the purpose of accepting all requests for objects not previously explicitly registered.

Explicitly started servers are those server processes that register to receive a specific or explicit object.

The benefits of starting a server automatically are:

- Since it is automatic, there is no need to "pre-start" the server in anticipation of a connection.
- Access control is enforced.

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).

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 to the 4DDN networking software as a server by specifying an object type number or task name to which it will respond. This is accomplished through an *ioctl*() request.

Once this call has been issued, the server process issues an *ioctl*() request to wait for a connection request and to receive the access control information transmitted by the client.

Registering 4DDN processes as servers must be performed before the client initiates a logical connection to a server. The server process must have opened a link and have registered as a server before any clients can connect to it.

The server process may use the control information received to decide whether or not to accept the request and then issues an ioctl request to accept or reject the link.

When the *ioctl*() to accept the 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 *close*()'d by both processes. In order for new logical links to be accepted by the server, the server process must re-register itself as a server. The process must then repeat the procedure of registering as a server.

Automatically Started Servers

Servers can be started automatically by the dnserver, a continuously running IRIX process. Whenever 4DDN receives a connection for an object (by name or number) that is not currently registered, the connection is given to the dnserver. The dnserver performs the following actions:

1. The 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 Yellow Pages passwd database, the connection is rejected.

2. The dnserver checks in */usr/etc/dn/servers.reg* for an entry for the requested object. It checks for an 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 of the 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 username and runs the server process. The process's working directory is that of the login directory of the user.

In this case, the server is started with the logical link opened but not yet accepted or rejected. The server is started with the following 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)
```

The process specified by the path found in */usr/etc/dn/servers.reg* or found in the users's login directory is started with a single argument. That argument is the file descriptor number of the logical link. The process is started as follows:

```
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 server 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 */
}
```

5.2.3 Transmitting and Receiving Data

After the link has been established, normal or interrupt data can be sent or received across the logical link through a series of calls using the assigned file descriptor.

An *ioctl* request allows a process to specify the I/O data format (RECORD or STREAM), and input mode (BLOCKING or NON-BLOCKING) for the *read*()'s.

read() is used by the process to receive data. *write*() is used by the process to send data.

Interrupt data is transmitted and received through *ioctl* requests.

5.2.4 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 acknowledgement 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 has been *close*()'d, the server process must re-register itself as a server in order to be a server for another logical link connection.

5.3 Task-To-Task Communication – Reference

This remainder of this chapter is a programmer's guide explaining each subroutine call in task-to-task communication. It is intended for C language programmers who require use of the task-to-task communications facilities provided within IRIS-4DDN.

5.3.1 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 only needed 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
```

5.3.2 The *errno* External Variable

If a 4DDN task-to-task function call returns a value of –1, it indicates that the function did not execute successfully. When this occurs, the external variable *errno* is set to the appropriate error code. *errno* is not changed under any other circumstances. Refer to Appendix A for more information on 4DDN error codes and recommended actions. Other error codes may be generated by IRIX. Refer to *intro*(2) in the *IRIS-4D Programmer's Reference Manual* for those errors. The library routine *dn_perror* can be called to print a descriptive message based on the errno value. It may also include a user-defined string that can identify the application program that generated the message. (This routine is described below.)

5.4 Opening a Logical Link Device

The *open*() call opens the logical link device and returns a unique logical link file descriptor to be 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

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

Description

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

Upon successful completion, the *open*() call returns a unique logical link file descriptor. This file descriptor is used for all subsequent I/O function calls pertaining to the associated logical link. In case of error, *open*() returns –1, and the external variable errno is set to the appropriate error code (See Appendix A for recommended actions.)

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

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

Description

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

Results

Upon successful completion, this *ioctl* call returns 0, and the logical link is established with the server program. If there is an error or the server program rejected the logical link request, *ioctl*() returns –1 and the external variable errno is set to the appropriate error code. (See Appendix A for recommended actions.) Additional data sent by the server and accompanying the acceptance or rejection of the connection 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;
```

Description

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 begin with an 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 Special 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 Special 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. *im_length* should be set by the requesting program when sending optional data to

the server program. When a server program returns optional data with the acceptance or rejection, *im_length* will be 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_NAME	Specified task name invalid
OBJ_BUSY	Insufficient resources at remote system
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. A program can be a server for either an object number or object name.

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);
```

Description

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

This call returns 0 when the server is registered correctly. It returns –1 if there is an error and the external variable *errno* is set to indicate the appropriate error code. (See Appendix A 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 can be used to poll for an incoming connection. It 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);
```

Description

OpenBlock	typedef defined in *<dn/defs.h>*.
link	Logical link file descriptor.
SES_GET_AI	*ioctl* request code.
SES_GET_AI_NB	*ioctl* polling 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 A for recommended actions.)

Error Codes

BY_OBJECT	The remote user disconnected the link
LOCAL_SHUT	The network software has been 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);
```

Description

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 are to be sent, the *sd_data.im_length* field must 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*().

Results

If the SES_ACCEPT call returns 0, the link is accepted and readied for the exchange of data. If it returns –1, an error occurred and the external variable *errno* is set to the appropriate error code. (See Appendix A for recommended actions.)

Upon successful completion, the SES_REJECT call returns 0 and the link is rejected. (To terminate the logical link, refer to "Closing the Logical Link" below) If the call returns –1, an error occurred and the external variable, *errno*, is set to the appropriate error code. (See Appendix A for recommended actions.) Also, the logical link file descriptor must be released by a close() if an error occurs.

Comment

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 new logical link connection requests, the server program must re-open the logical link device and re-register itself as a server.

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 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 prior to 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 only receives 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. Note that one end of the link may run in STREAM format and the other in RECORD format.

I/O Modes

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 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 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 either:

1. The number of bytes received and written into the user-supplied data buffer OR
2. −1 and the external variable errno set to NO_DATA_AVAIL indicating that no data is available.

In NON-BLOCKING I/O mode with polling, a program must issue read commands to determine if data is available. In NON-BLOCKING I/O mode with signal notification, a program receives a notification of data availability prior to issuing a read. If you register a signal number with 4DDN, you must register a signal handler with IRIX (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);
```

Description

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_STREAM_MODE, SES_IO_RECORD_MODE.
opt.io_nonblocking	Indicates the input type: SES_IO_BLOCKING, 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 only used when the non-blocking I/O mode is chosen.
ret	Value returned by the *ioctl*()

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 A for recommended actions.)

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.

Error Codes

BAD_COMMAND	Invalid ioctl command

5.10 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);
```

Description

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

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

5.11 Receiving Data Across a Logical Link

5.11.1 Stream Format

Call Usage

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

Description

link	Logical link file descriptor.
buf	Character buffer in which data from the link is to be placed.
nbytes	The size in bytes of *buf*.
ret	The number of bytes read or –1.

Results

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

Upon successful completion, the number of bytes placed in the allocated buffer is returned. If the number of bytes requested (nbytes) is smaller than the size of the data available to be read, the buffer is filled with the amount of data requested in the nbytes field. In this mode, no attempt is made to inform the user that more data is available. Subsequent *read*()'s will return any further data.

If *read*() returns –1, an error has occurred. In this case, the external variable, *errno*, is set to the appropriate error code.

If the I/O mode is SES_IO_BLOCKING, this *read*() will block until there is data available, or until an error occurs. If the I/O mode is SES_IO_NON_BLOCKING, and there is no data available, the *read*() returns –1, and the external variable *errno* is set to NO_DATA_AVAIL.

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

5.11.2 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);
```

Description

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	The size in bytes of *sesrec.sr_buffer*.
ret	The number of bytes read or –1.

Results

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

Upon successful completion, the number of bytes placed in the allocated buffer is returned. If the number of bytes requested is smaller than the size of the data available to be read, the buffer is filled with the amount of data requested in the *nbytes* field, and the user is informed that more data is available by the value in the *sr_status* field of the SesRecord.

If *read*() returns –1, then an error has occurred, and the external variable, *errno*, is set to the appropriate error code. (See Appendix A, "4DDN Error Codes," for recommended actions.)

If the I/O mode is BLOCKING, the *read*() function call blocks until there is data available, or until an error occurs. If the I/O mode is NON-BLOCKING, and there is no data available, the *read*() returns –1 and *errno* is set to NO_DATA_AVAIL.

This function call places the status of the data into *sr_status*. If the status returned is BEG_OF_MESSAGE or MID_OF_MESSAGE, one or more subsequent *read*()s must be issued to receive the rest of the data sent.

Example

Let a remote program issue 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 particular example, the second, third, fourth and fifth *read*()s returns 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

5.12 Sending Data Across a Logical Link

5.12.1 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);
```

Description

link	Logical link file descriptor.
buf	Data buffer from which the data is taken.
nbytes	The length of the buffer (in bytes) to transmit. The maximum length supported by 4DDN is found through the *ioctl* call SES_MAX_IO.
ret	The actual number of bytes that were sent.

Results

The *write*() call returns a value representing 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 A 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, then the *write*() function returns immediately. If 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

5.12.2 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, (e.g., 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);
```

Description

SesRecord	typedef defined in *<dn/defs.h>*
link	Logical link file descriptor
sesrec	Structure containing: the address of the buffer containing transmission data; and which data to transmit is taken, and status information about this data. The status of the read is placed in the *sr_status* field. The *sesrec.sr_status* field is set to a value that indicates where this block of data fits within a message. Possible values are: BEG_OF_MESSAGE MID_OF_MESSAGE END_OF_MESSAGE COMPLETE (See Appendix A 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

Write() returns a value representing 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 A for recommended actions.)

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

If the I/O mode is NON-BLOCKING, then 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.

Comment

If the statuses from multiple record format *write*() function calls are sent out of sequence (e.g., 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

5.13 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 the importance of interrupt data, 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);
```

Description

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 A 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

5.14 Accepting and Receiving Interrupt Data

Receiving interrupt data is a two-step process. First, an ACCEPT_INT ioctl request must be issued. This call is issued only once. It instructs the logical link device driver when the program interrupt data is received. This is accomplished through the specified signal.

The second step is performed every time data is received. The RECV_INTERRUPT *ioctl* request places interrupt data into the *im_data* field. See the second part of this section for more information and an example of this 2-step process.

5.14.1 ACCEPT_INT ioctl

Call Usage

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

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

Description

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 A for recommended actions.)

5.14.2 RECV_INTERRUPT ioctl

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);
```

Description

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.

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 A for recommended actions.)

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
NO_DATA_AVAIL	No data available (read only)
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

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.

Example

The following program illustrates the 2-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);
}
```

5.15 Disconnecting a Logical Link

A disconnect operation is 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);
```

Description

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*().

Results

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

Comments

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 whose receipt has not been acknowledged) before the link is terminated. The disconnect *ioctl* blocks until all transmitted data is received by the remote process.

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

5.16 Aborting a Logical Link

This *ioctl* request is issued by the client or the server.

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);
```

Description

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 A 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

5.17 Closing the 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 ink 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.

There are two other methods for terminating 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 A for recommended actions.)

Comment

If optional data is desired, a disconnect, or abort ioctl, must be issued before *close*().

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

5.18 Obtaining Link Status

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.

5.19 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.

Call Usage

```
char *string;

dn_perror(string);
```

Description

string	A character array or a string constant.

Results

The *dn_perror* function has no return value.

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 has been established and the following calls may be issued by server and client *ioctl*() to define I/O data format and input mode (SES_IO_TYPE) *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

Appendix A: 4DDN Error Codes

Table A-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 A-1 Error Codes and Recommended Actions (continued)

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 A-1 Error Codes and Recommended Actions (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 A-1 Error Codes and Recommended Actions (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 A-1 Error Codes and Recommended Actions (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 A-1 Error Codes and Recommended Actions (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 A-1 Error Codes and Recommended Actions (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 A-1 Error Codes and Recommended Actions (continued)

Name	Meaning	Recommended Actions
DUPE_NODE_NAME	Duplicate node name detected.	Check the contents of the NCP database. If 4DDN was initialized with this node name associated with a different node number, change the name with the NCP "set node" and "define node" commands.
DUPE_NODE_NUM	Duplicate node number detected.	Check the contents of the NCP database. If 4DDN was initialized with this node associated with a different node number, change the 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.

Appendix B: 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*.

B.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);
}
```

B.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 C: Glossary

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 and adjacent node
: A node that is currently communicating or ready to communicate with another node.

Application-dependent
: The application programmer has the option of filling some fields in the data structures with user-defined data.

Area number
: The area number identifies a group of nodes in the network. The area number must be an integer in the 1-63 range.

Blocking I/O
: Input mode. When blocking I/O is selected, read operations wait until data becomes available.

Client
: Local process that requests a logical link connection in task-to-task communication.

Collision
: Simultaneous transmissions by two or more nodes on an ethernet network.

Congestion
: Occurs when there are too many packets to be queued.

Counters

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

Datagram

When a unit of data is received, the routing control information is removed and the remaining information is called a datagram.

DNA

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

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.

Inactive node

A node that is not currently communicating or ready to communicate with another node on the ethernet.

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 packet in ethernet terminology.

Inactive node

A node that is not currently communicating or ready to communicate with another system on the ethernet.

Interrupt data

Special high-priority control information that is transmitted immediately.

Logical link

A virtual circuit between two application programs.

Logical link device

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

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

Input mode. If non-blocking I/O is selected, the 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.

Null-terminated string

A string that ends with zero.

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 includes the node name, object type or name, user name, and the password.

Optional data

Special data field that is generally used by the application program to explain the reason for termination of a logical link.

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.

Record

I/O data format. When this format is selected, a process passes and receives data in a structure indicating whether the message is complete or incomplete. If incomplete, a special status field indicates whether it is the beginning, the middle, or the end of the message.

Server

Remote process that accepts or rejects a logical link connection when a process is attempting to establish a logical link in task-to-task communication Stream I/O data format. In stream format, a process receives data as it appears across the logical link without distinguishing where messages begin and end.

Task-to-task communications

The exchange of data between two processes via a logical link.