Tru64 UNIX

Network Programmer’s Guide

Part Number: AA-RH9UD-TE

June 2001

Product Version: Tru64 UNIX Version 5.1A or higher

This manual describes the Compaq Tru64™ UNIX network programming environment. It describes the sockets and STREAMS frameworks, including information about system calls, header files, libraries, and software bridges that allow sockets programs to use STREAMS drivers and STREAMS programs to use BSD-based drivers. Additionally, it describes how to write programs to the X/Open Transport Interface (XTI), as well as how to port sockets-based applications to XTI. It also describes the Tru64 UNIX eSNMP API.

Compaq Computer Corporation
Houston, Texas

© 2001 Compaq Computer Corporation

COMPAQ and the Compaq logo. Registered in the U.S. Patent and Trademark Office. Alpha and Tru64 are trademarks of Compaq Information Technologies Group, L.P.

Microsoft and Windows NT are trademarks of Microsoft Corporation. UNIX and The Open Group are trademarks of The Open Group.All other product names mentioned herein may be trademarks of their respective companies.

Confidential computer software. Valid license from Compaq required for possession, use, or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

Compaq shall not be liable for technical or editorial errors or omissions contained herein. The information in this document is provided "as is" without warranty of any kind and is subject to change without notice. The warranties for Compaq products are set forth in the express limited warranty statements accompanying such products. Nothing herein should be construed as constituting an additional warranty.

Portions of this document are adapted from *A STREAMS-based Data Link Provider Interface – Version 2* © 1991 UNIX International, Inc. Permission to use, copy, modify, and distribute this documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appears in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name UNIX International not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. UNIX International makes no representations about the suitability of this documentation for any purpose. It is provided "as is" without express or implied warranty.

Contents

About This Manual

1 Introduction to the Network Programming Environment

2 Data Link Provider Interface

3 X/Open Transport Interface

4 Sockets

5 Tru64 UNIX STREAMS

6 Extensible SNMP Application Programming Interface

7 RSVP Application Programming Interface

8 Tru64 UNIX STREAMS/Sockets Coexistence

A Sample STREAMS Module

B Socket and XTI Programming Examples

C IPv4 and IPv6 Sockets Programming Examples

D TCP Specific Programming Information

E Information for Token Ring Driver Developers

F The Data Link Interface

Glossary

Index

Examples

Figures

Tables

About This Manual

This manual explains how to write programs with the X/Open Transport Interface (XTI) calls, STREAMS I/O calls, and the Berkeley Software Distribution (BSD) socket calls. For XTI and sockets, it provides conceptual and programming information. Additionally, it explains how to port applications from Transport Layer Interface (TLI) to XTI and from sockets to XTI. For STREAMS, this manual explains any differences between the Tru64 UNIX implementation and the AT&T System V Release 4 implementation. It also provides information on the Extensible System Network Management Protocol (eSNMP) application programming interface, the Resource ReSerVation Protocol (RSVP) application programming interface, and AF_INET6 sockets.

After reading this manual, you should be able to:

- Understand the programming support provided in Tru64 UNIX for networking
- Write an XTI application by using either connection-oriented or connectionless service
- Understand the Tru64 UNIX implementation of STREAMS
- Write a socket application
- Understand the differences between TLI and XTI and between sockets and XTI
- Write an eSNMP application

Audience

This manual addresses experienced UNIX programmers. We assume you are familiar with the following:

- C language
- Programming interfaces for UNIX operating systems
- Basic networking concepts, including an understanding of the Open Systems Interconnection (OSI) 7-layer model
- Efforts required to write networking applications

New and Changed Features

This revision of the *Network Programmer's Guide* contains the following changes:

- *Chapter 4* has been revised. The chapter now includes information that was previously contained in a separate chapter on AF_INET6 sockets. The chapter also includes a new section on creating Internet applications and a new section on porting applications to use AF_INET6 sockets. In addition, the chapter includes a list of IPv6 address testing macros.
- *Appendix C* is new, and contains sample client and server programs that use the IPv6 network library routines and the TCP transport. This is information was previously contained in a separate chapter on AF_INET6 sockets.
- *Appendix D* has been revised. The appendix now includes updated information about TCP buffer sizes and the maximum buffer size allowed on the system. In addition, it contains information on the following TCP performance options: selective acknowledgement, protection against wraparound sequence numbers (PAWS), and timestamp option.

Organization

This manual is organized as follows:

Chapter 1	Provides an overview of XTI, STREAMS, sockets, and the programming tasks required for network applications.
Chapter 2	Describes the `dlb` pseudodriver, which implements a subset of the the Data Link Provider Interface (DLPI).
Chapter 3	Describes the fundamental concepts associated with XTI, how to write connection-oriented and connectionless applications, compatibility issues with TLI, and how to port applications to XTI. XTI errors are also covered in this chapter.
Chapter 4	Describes the concepts associated with the socket interface, and how to write socket applications.
Chapter 5	Describes Tru64 UNIX implementation of STREAMS.
Chapter 6	Describes the Extensible System Network Management Application Programming Interface.
Chapter 7	Describes the Resource ReSerVation Protocol Application Programming Interface.
Chapter 8	Describes the `ifnet` STREAMS module and `dlb` STREAMS pseudodriver communication bridges.
Appendix A	Provides a sample STREAMS module.
Appendix B	Provides XTI and sockets programming examples.

Appendix C Provides AF_INET and AF_INET6 sockets programming examples.

Appendix D Provides Transport Protocol Control (TCP) specific programming information.

Appendix E Provides information required by token ring driver developers.

Appendix F Describes the Data Link Interface (DLI) and provides programming examples.

This guide also contains a glossary of terms and an index.

Related Documents

For general information about programming with Tru64 UNIX, refer to the *Programmer's Guide*.

For additional information about networking APIs, refer to the following manual:

- *UNIX Network Programming, Networking APIs: Sockets and XTI* by W. Richard Stevens, ISBN 0-13-490012-X, published by Prentiss-Hall

For additional information about XTI, refer to the following manuals:

- *X/Open Portability Guide Volume 7: Networking Services* (XPG3), ISBN 0-13-685892-9
- *Application Environment Specification (AES) Operating System Programming Interfaces Volume*, ISBN 0-13-043522-8, published by Prentice-Hall, includes all of the mandatory XTI calls
- *X/Open CAE Specification: Networking Services, Issue 4* (XNS4.0), ISBN 1-85912-049-0

For additional information about the STREAMS I/O framework, refer to the following manuals:

- *Programmer's Guide: STREAMS*. Englewood Cliffs:Prentice-Hall, Inc., 1990.

  This manual explains how to write applications, modules, and device drivers with STREAMS.
- *AT&T System V Release 4 Programmer's Reference Manual*. Englewood Cliffs:Prentice-Hall, Inc., 1989.

  This manual contains the reference pages for all programming interfaces, including those for STREAMS.
- *AT&T System V Release 4 System Administrator's Reference Manual*. Englewood Cliffs:Prentice-Hall, Inc., 1989.

  This manual contains the reference pages for STREAMS `ioctl` commands.

- *Transport Provider Interface (TPI) Specification*, UNIX International

For additional information about the socket interface, refer to the following books:

- *Internetworking with TCP/IP: Principles, Protocols, and Architecture*. Englewood Cliffs:Prentice-Hall, Inc., 1988.

  This book, by Douglas Comer, includes a chapter that describes the socket interface.
- *X/Open CAE Specification: Networking Services, Issue 4* (XNS4.0), ISBN 1-85912-049-0
- *Protocol Independent Interfaces P1003.1g Draft 6.6*. IEEE draft 1997.
- *Design and Implementation of the 4.3BSD UNIX Operating System*. Reading:Addison-Wesley Publishing Company, 1989.

  This book, by Leffler, McKusick, Karels, and Quarterman, includes information about the purpose and use of sockets.

For information about administering networking interfaces, refer to the *System Administration* manual and the *Network Administration: Connections* manual.

Icons on Tru64 UNIX Printed Manuals

The printed version of the Tru64 UNIX documentation uses letter icons on the spines of the manuals to help specific audiences quickly find the manuals that meet their needs. (You can order the printed documentation from Compaq.) The following list describes this convention:

G	Manuals for general users
S	Manuals for system and network administrators
P	Manuals for programmers
R	Manuals for reference page users

Some manuals in the documentation help meet the needs of several audiences. For example, the information in some system manuals is also used by programmers. Keep this in mind when searching for information on specific topics.

The *Documentation Overview* provides information on all of the manuals in the Tru64 UNIX documentation set.

Reader's Comments

Compaq welcomes any comments and suggestions you have on this and other Tru64 UNIX manuals.

You can send your comments in the following ways:

- Fax: 603-884-0120 Attn: UBPG Publications, ZKO3-3/Y32
- Internet electronic mail: `readers_comment@zk3.dec.com`

  A Reader's Comment form is located on your system in the following location:

  `/usr/doc/readers_comment.txt`

Please include the following information along with your comments:

- The full title of the manual and the order number. (The order number appears on the title page of printed and PDF versions of a manual.)
- The section numbers and page numbers of the information on which you are commenting.
- The version of Tru64 UNIX that you are using.
- If known, the type of processor that is running the Tru64 UNIX software.

The Tru64 UNIX Publications group cannot respond to system problems or technical support inquiries. Please address technical questions to your local system vendor or to the appropriate Compaq technical support office. Information provided with the software media explains how to send problem reports to Compaq.

Conventions

This document uses the following typographic conventions:

`%`
`$`

A percent sign represents the C shell system prompt. A dollar sign represents the system prompt for the Bourne, Korn, and POSIX shells.

`#`

A number sign represents the superuser prompt.

`%` **`cat`**

Boldface type in interactive examples indicates typed user input.

`file`

Italic (slanted) type indicates variable values, placeholders, and function argument names.

[|]
{ | }

In syntax definitions, brackets indicate items that are optional and braces indicate items that are required. Vertical bars separating items inside brackets or braces indicate that you choose one item from among those listed.

. . .

In syntax definitions, a horizontal ellipsis indicates that the preceding item can be repeated one or more times.

`cat`(1)

A cross-reference to a reference page includes the appropriate section number in parentheses. For example, `cat`(1) indicates that you can find information on the `cat` command in Section 1 of the reference pages.

Return

In an example, a key name enclosed in a box indicates that you press that key.

Ctrl/*x*

This symbol indicates that you hold down the first named key while pressing the key or mouse button that follows the slash. In examples, this key combination is enclosed in a box (for example, Ctrl/C).

1

Introduction to the Network Programming Environment

The network programming environment includes the programming interfaces for application, kernel, and driver developers writing network applications and implementing network protocols. Additionally, it includes the kernel-level resources that an application requires to process and transmit data, some of which include libraries, data structures, header files, and transport protocols.

This chapter introduces network programming environment by focusing on how the data link and application programming interfaces work to get data from an application in user space, through the network layers in kernel space, out onto the network, and back again.

Information about the kernel resources that support the interfaces is included in later chapters in this book. Individual chapters describe the particular system and library calls, data structures, and other programming considerations for each interface.

The primary components of the network programming environment are summarized in Table 1–1.

Table 1–1: Components of the Network Programming Environment

Component	Interface	Description
Data Link Interfaces	Data Link Interface (DLI)	Allows programs to access the data link layer to communicate with DLI programs on other systems.Tru64 UNIX provides DLI for backward compatibility with ULTRIX. See Appendix F.

Table 1–1: Components of the Network Programming Environment (cont.)

Component	Interface	Description
	`dlb` interface	Kernel-level interface targeted for STREAMS protocol modules that either use or provide data link services. The `dlb` STREAMS pseudodriver implements a subset of the Data Link Provider Interface (DLPI). See Chapter 2 and the Data Link Provider Specification (`dlpi.ps`) located in the `/usr/share/doclib/dlpi` directory. Note that the OSFPGMR*nnn* subset must be installed to access the DLPI specification online.
Application Programming Interfaces	Sockets	The de facto industry standard programming interface. Tru64 UNIX implements the 4.3BSD, 4.4BSD, XNS4.0, and POSIX 1003.1g Draft 6.6 socket interface. The Internet Protocol Suite, which consists of TCP, UDP, IP, ARP, ICMP, and SLIP is implemented over sockets. See RFC 1200: *IAB Protocol Standards* and Chapter 4.
	STREAMS	A kernel mechanism that supports the implementation of device drivers and networking protocol stacks. The STREAMS framework defines interface standards for character input and output within the kernel as well as between the kernel and user levels. The Tru64 UNIX operating system provides an AT&T, System V Release 4.0 compatible version of STREAMS. See Chapter 5.

Table 1–1: Components of the Network Programming Environment (cont.)

Component	Interface	Description
	XTI/TLI	A protocol independent, transport layer application interface that consists of a series of functions. XTI is based on the Transport Layer Interface (TLI) and the transport service definition for the Open Systems Interconnection (OSI) model. See Chapter 3.
	eSNMP	A set of routines that enables you to extend the SNMP agent process by creating Management Information Bases (MIBs). See Chapter 6.
	RSVP	An industry—standard protocol and set of routines that enable applications to request enhanced quality-of-service (QoS). See Chapter 7.
Communication Bridges Between STREAMS and Sockets	`ifnet` STREAMS module	Allows STREAMS-based network device drivers to access the sockets-based TCP/IP protocol stack provided on Tru64 UNIX. See Chapter 8.
	`dlb` pseudodriver	Allows applications that use STREAMS-based protocol stacks to access BSD-based drivers. The `dlb` pseudodriver implements a subset of the DLPI specification. See Chapter 8.

It is easiest to understand the network programming environment by examining each component. The following sections introduce the environment piece by piece, starting with the components closest to the network and working up.

1.1 Data Link Interfaces

The network programming environment supports both the Data Link Interface (DLI) and the Data Link Provider Interface (DLPI). DLI enables you to port programs that run on ULTRIX systems to Tru64 UNIX systems. See Appendix F for information about DLI.

DLPI is a kernel-level interface that maps to the data link layer of the OSI reference model. DLPI frees its users from specific knowledge of the characteristics of the data link provider, allowing them to be implemented independently of a specific communications medium. Chapter 2 describes in greater detail DLPI, the Tru64 UNIX `dlb` pseudodriver, and the supported primitives.

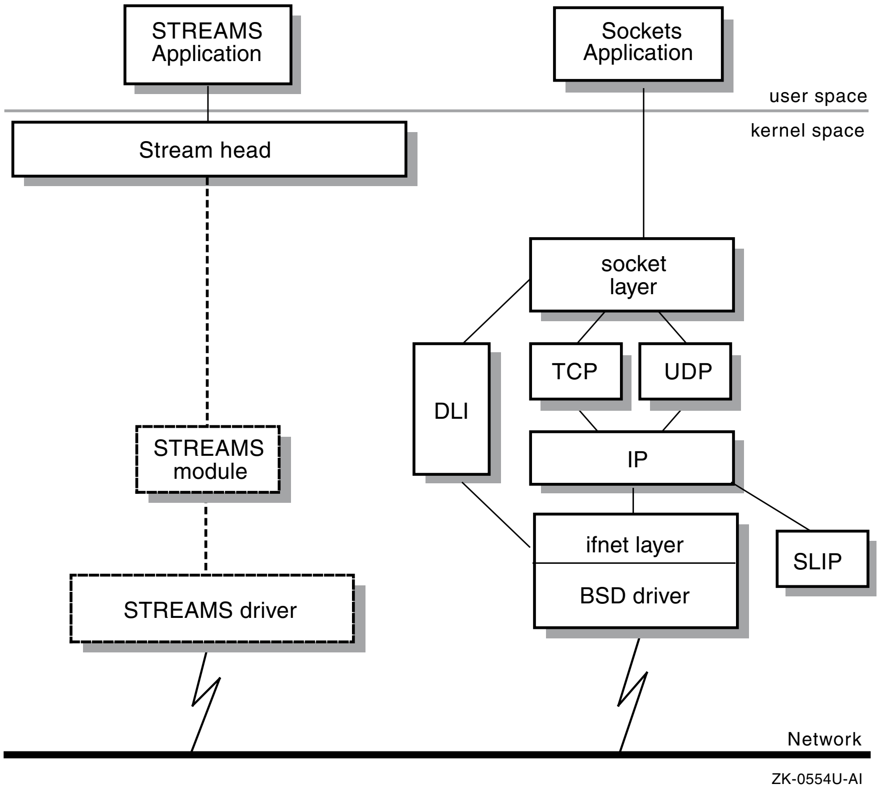
1.2 Sockets and STREAMS Frameworks

The Tru64 UNIX operating system supports AT&T's System V Release 4 STREAMS and BSD sockets frameworks for writing networking applications and for doing kernel-level network input/output (I/O). A framework comprises a particular programming interface and the kernel-level resources that the system requires to transmit and receive data.

Sockets is the de facto industry standard interface for writing networking applications. The sockets framework is BSD-based, consisting of a series of system and library calls, header files, and data structures. Applications can access kernel-resident networking protocols, such as the Internet Protocol suite, through socket system calls. Applications can also use socket library calls to manipulate network information; for example, mapping service names to service numbers or translating the byte order of incoming data to that appropriate for the local system's architecture.

The STREAMS framework provides an alternative to sockets. The STREAMS interface was developed by AT&T and consists of system calls, kernel routines, and kernel utilities that are used to implement everything from networking protocol suites to device drivers. Applications in user space access the kernel portions of the STREAMS framework using system calls such as `open`, `close`, `putmsg`, `getmsg`, and `ioctl`. Figure 1–1 illustrates the STREAMS and sockets frameworks.

Figure 1–1: Sockets and STREAMS Frameworks



Note

Tru64 UNIX supports, but does not provide any STREAMS-based transport providers (the dotted line portion of in Figure 1–1.

With sockets, the application in user space passes data to the appropriate socket system calls, which then pass it to the network layer. Finally, the network layer passes it, via the `ifnet` layer, to the BSD driver, which puts it on to the network.

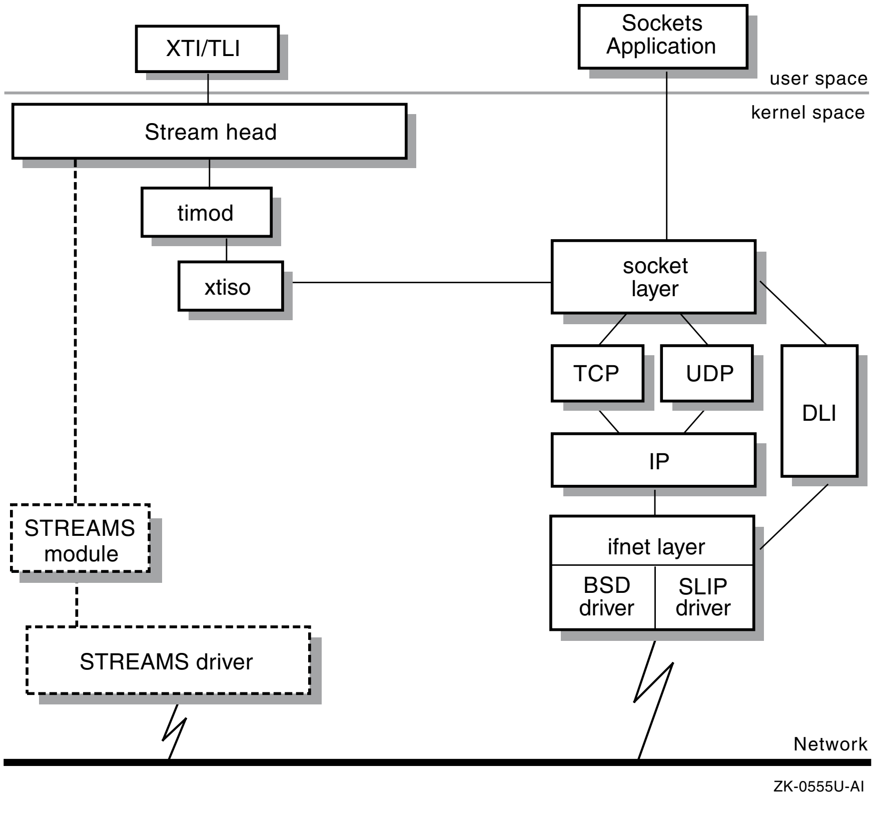
With STREAMS, the application in user space passes data to the Stream head, which passes it to any STREAMS modules that have been pushed on the Stream to process it. Each module passes the data to the next module until it finally reaches the STREAMS driver, which puts it out on to the network.

1.3 X/Open Transport Interface

The X/Open Transport Interface (XTI) defines a transport layer application interface that is independent of any transport provider. This means that programs written to XTI can be run over a variety of transport providers, such as the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP). The application specifies which transport provider to use.

Figure 1–2 illustrates the interaction between XTI and the STREAMS and sockets frameworks.

Figure 1–2: XTI, STREAMS, and Sockets Interactions



Depending on the transport provider specified by the application, data can flow along one of two paths:

1. If a STREAMS-based transport provider is specified, data follows the same route that it did for an application written to run over STREAMS. It passes first through the Stream head, then to any modules that the application pushed onto the Stream, and finally to the STREAMS driver, which puts it on to the network.

Note

Tru64 UNIX does not provide any STREAMS-based transport providers.

2. If a socket-based transport provider is specified (TCP or UDP), data is passed through `timod` and `xtiso`. The appropriate socket layer routines are called and the data is passed through the Internet protocols and `ifnet` layer to the BSD-based driver, which puts it on to the network.

1.4 Extensible SNMP

The Tru64 UNIX SNMP agent provides a framework for extensibility (called eSNMP). The SNMP daemon functions as an extensible master agent, communicating with various subagents via the eSNMP protocol. The master agent implements the SNMP on behalf of the entire system, while subagents provide the actual MIB instrumentation. The eSNMP subagent development tools and API provide the mechanism for users to develop subagents that communicate with the master agent and extend the MIB view.

1.5 RSVP Application Programming Interface

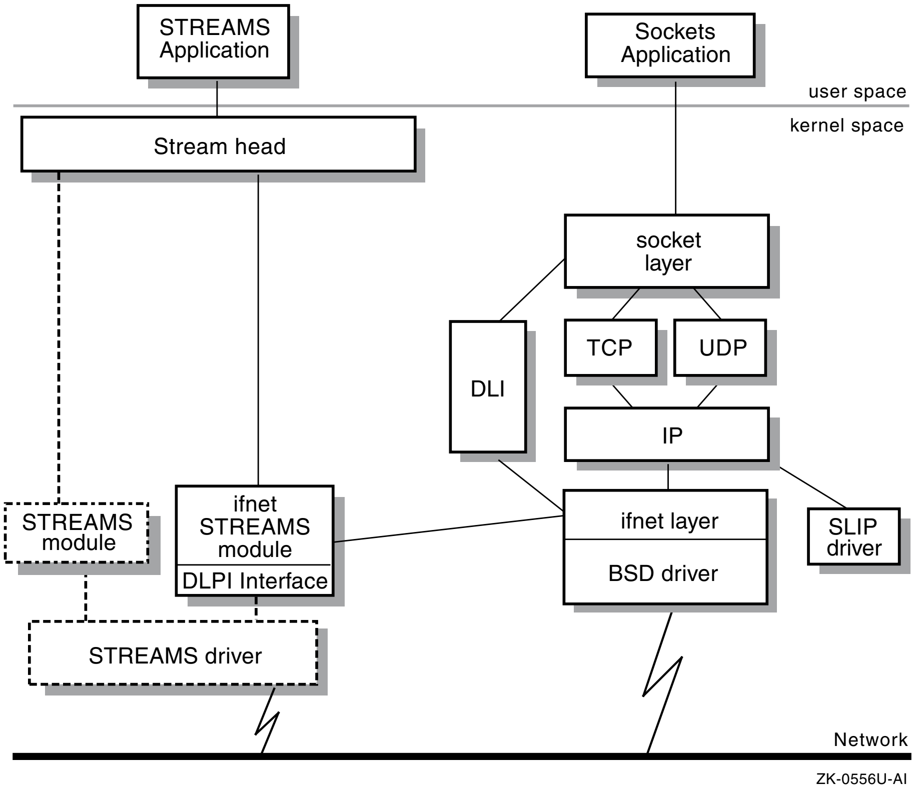
The Tru64 UNIX implementation of Resource ReSerVation Protocol (RSVP) enables applications to use the RSVP Application Programming Interface (RAPI) to request enhanced quality-of-service (QoS) for specific application data streams or flows. The application makes these request to the `rsvpd` daemon, which in turn uses RSVP to send and receive RSVP messages through the network. The `rsvpd` daemon also communicates with the Traffic Control subsystem to install and modify flows and filters on a given network interface.

1.6 Sockets and STREAMS Interaction

The `ifnet` STREAMS module allows programs using Tru64 UNIX BSD-based TCP/IP to access STREAMS-based drivers. It provides the `dlb` pseudodriver to allow programs using a STREAMS-based protocol stack to access BSD-based drivers provided on Tru64 UNIX.

Figure 1–3 illustrates an application using the BSD-based TCP/IP provided on the operating system and accessing a STREAMS-based driver.

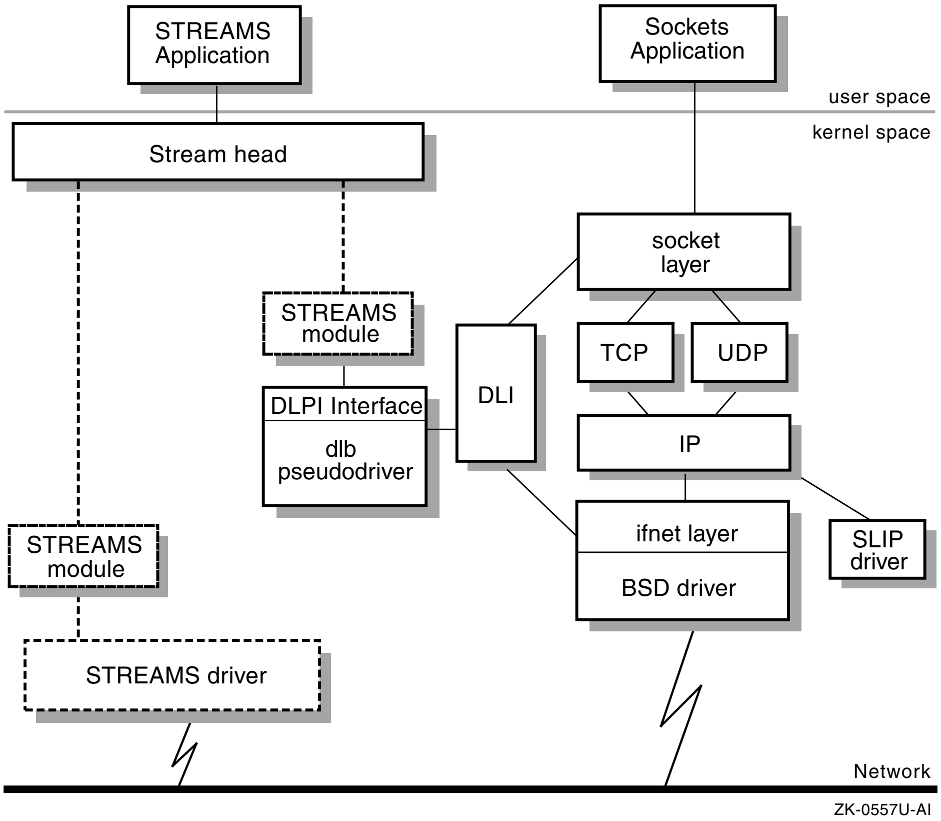
Figure 1–3: Bridging STREAMS Drivers to Sockets Protocol Stacks



In Figure 1–3, data travels from a sockets-based application through the appropriate sockets system calls and is processed by the Internet protocols. Then the BSD `ifnet` layer of the networking subsystem, whose function is to map BSD `ifnet` messages to DLPI, passes the data to the `ifnet` STREAMS module. The `ifnet` STREAMS module processes it so that the STREAMS driver can put it on to the network. When information for the sockets-based application is returned, the STREAMS driver picks it up off of the network and passes it to the DLPI interface of the `ifnet` STREAMS module. The DLPI interface of the `ifnet` STREAMS module translates DLPI messages to BSD `ifnet` and passes it back to the BSD `ifnet` layer. The data is then processed by the Internet protocols and passed back to the application.

Figure 1–4 illustrates an application using a STREAMS-based protocol stack and accessing a BSD-based driver.

Figure 1–4: Bridging BSD Drivers to STREAMS Protocol Stacks

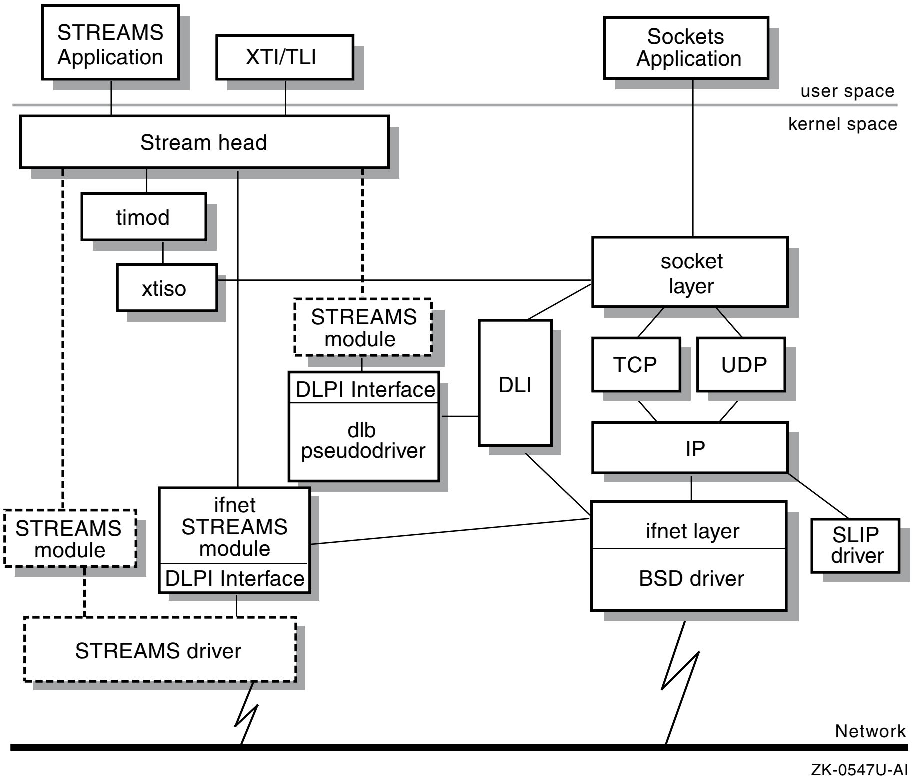


In Figure 1–4, data travels from a STREAMS-based application through the Stream head and is processed by whatever Streams modules have been pushed onto the stack. Instead of finally being passed to a STREAMS driver, the data is passed to the `dlb` STREAMS pseudodriver and is then forwarded to the `ifnet` layer of the sockets framework. From there it is further processed by a BSD driver and put on to the network.

1.7 Putting It All Together

Figure 1–5 represents the entire network programming environment. Variations of this figure appear in each chapter to give you perspective on the information being presented.

Figure 1–5: The Network Programming Environment



2

Data Link Provider Interface

Tru64 UNIX provides the dlb STREAMS pseudodriver, which is a partial implementation of the Data Link Provider Interface (DLPI).

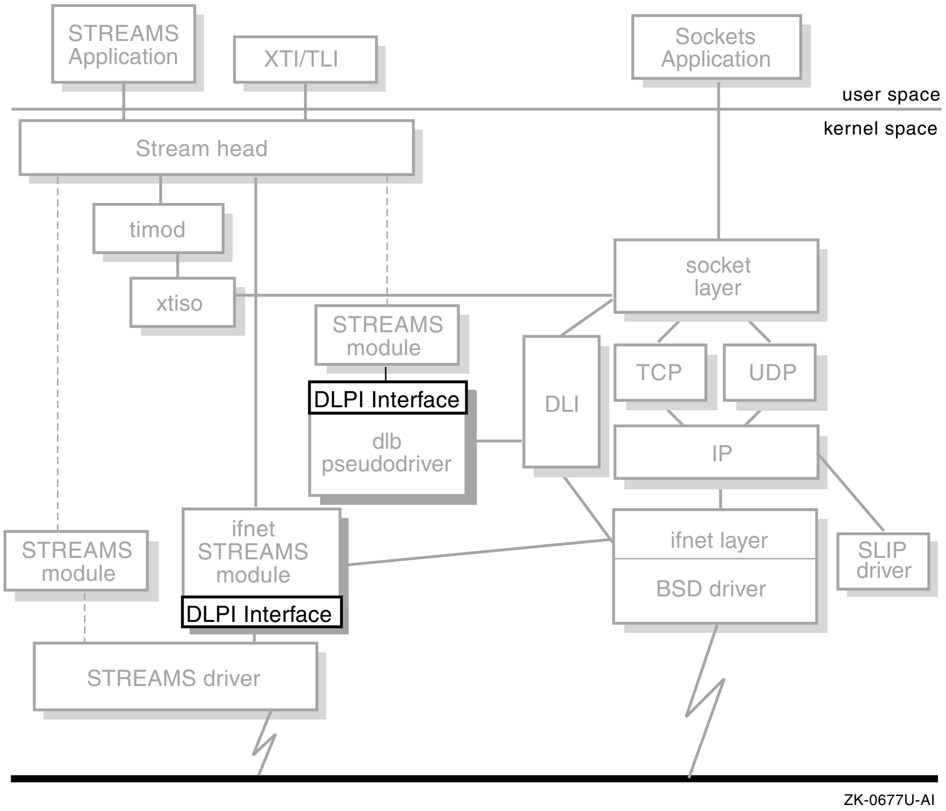
This chapter describes the dlb STREAMS pseudodriver and the basics of DLPI. A PostScript copy of the DLPI specification (dlpi.ps) is located in the /usr/share/doclib/dlpi directory.

Note

You must have the OSFPGMR*nnn* subset installed to access the DLPI specification online.

Figure 2–1 highlights the data link interfaces and shows their relationship to the rest of the network programming environment.

Figure 2–1: DLPI Interface



Note

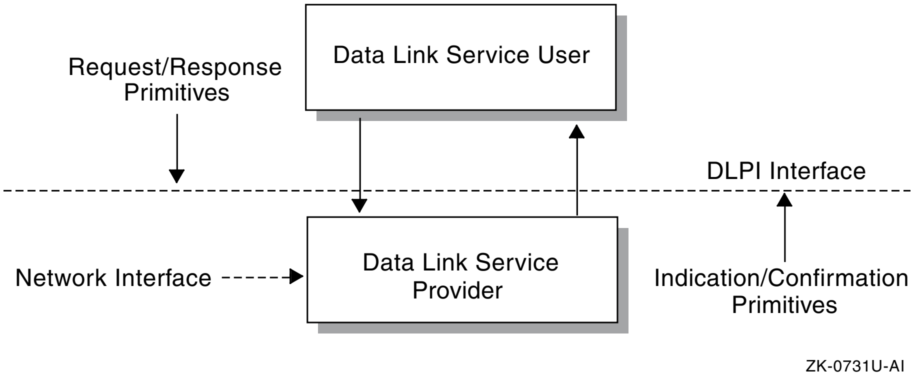
> The `dlb` STREAMS pseudodriver supports a subset of DLPI primitives. See Section 2.4 for a list of the supported primitives.

The data link interface is the boundary between the network and data link layers of the OSI reference model. A network application, or data link service user (DLS user), uses the services of the data link interface. A driver, pseudodriver, or data link service provider (DLS provider), provides the services to the data link layer.

DLPI specifies a STREAMS kernel-level service interface that maps to the OSI reference model. It defines an interface to the services of the data link layer and provides a definition of service primitives that make up the interface.

Figure 2–2 shows the components of DLPI. The DLS user communicates with the DLS provider using request/response primitives; the DLS provider communicates with the DLS user with indication/confirmation primitives.

Figure 2–2: DLPI Service Interface


Data Link Service User
Request/Response Primitives
DLPI Interface
Network Interface
Data Link Service Provider
Indication/Confirmation Primitives
ZK-0731U-AI

Section 2.4 lists supported primitives.

2.1 Modes of Communication

DLPI supports three modes of communication:

- Connection

  Enables a DLS user to establish a data link connection, transfer data over that connection, reset the link, and release the connection when the conversation has terminated.

  The connection service establishes a data link connection between a local DLS user and a remote DLS user for the purpose of sending data. Only one data link connection is allowed on each Stream.

- Connectionless

  Enables a DLS user to transfer units of data to peer DLS users without incurring the overhead of establishing and releasing a connection. The connectionless service does not, however, guarantee reliable delivery of data units between peer DLS users (for instance, lack of flow control may cause buffer resource shortages that result in data being discarded).

  Once a Stream has been initialized using the local management services, it may be used to send and receive connectionless data units.

Note

Tru64 UNIX supports only the connectionless mode of communication.

- Acknowledged connectionless

  Designed for general use for the reliable transfer of information between peer DLS users. These services are intended for applications that require acknowledgement of data unit transfer across LANs, but wish to avoid the complexity associated with the connection-mode services. Although the exchange service is connectionless, in-sequence delivery is guaranteed for data sent by the initiating station.

2.2 Types of Service

This section describes the types of service, or phases of communication, supported by DLPI. Note that the types of service available depend on the mode of communication (connection, connectionless, acknowledged connectionless) between the DLS provider and the DLS user.

DLPI supports the following types of service:

- Local management services
  - Information reporting service
  - Attach service
  - Bind service
- Connection-mode services
  - Connection establishment
  - Data transfer
  - Connection release
  - Reset service
- Connectionless-mode services
  - Connectionless data transfer
  - Quality of Service (QOS) management
  - Error reporting
- Acknowledged connectionless-mode services
  - Acknowledged connectionless-mode data transfer
  - Quality of service (QOS) management
  - Error reporting

2.2.1 Local Managment Services

The local management services apply to all three modes of communication supported by DLPI. They enable a DLS user to initialize a Stream that is connected to a DLS provider and to establish an identity with that provider. The local management services support the following:

- Information reporting service

  Provides information about the DLPI Stream to the DLS user.
- Attach service

  Assigns a physical point of attachment (PPA) to a Stream. See Section 2.3 for more information.
- Bind service

  Associates a data link service access point (DLSAP) with a Stream.

2.2.2 Connection-Mode Services

The connection-mode services allow two DLS users to establish a data link connection between them to exchange data, and to reset the link and release the connection when the conversation is through. The connection-mode services support the following:

- Connection establishment service

  Establishes a data link connection between a local DLS user and a remote DLS user for the purposes of sending data.
- Data transfer service

  Provides for the exchange of user data in either direction or both directions simultaneously. Data is sent in logical groups called data link service data units (DLSDUs) and is guaranteed to be delivered in the order in which it was sent.
- Connection release service

  Enables either the DLS user or DLS provider to break an established connection.
- Reset service

  Allows a DLS user to resynchronize the use of a data link connection, or a DLS provider to report detected loss of data unrecoverable within the data link service.

2.2.3 Connectionless-Mode Services

The connectionless-mode services allow DLS users to exchange data without incurring the overhead of establishing and releasing a connection. The connectionless-mode services support the following:

- Connectionless data transfer service

  Provides for the exchange of user data (DLSDU) in either direction or in both directions simultaneously.

- Quality of service (QOS) management service

  Enables a DLS user to specify the quality of service it can expect for each invocation of the connectionless data transfer service.

- Error reporting service

  Provides a means to notify a DLS user that a previously sent data unit either produced an error or could not be delivered. However, the error reporting service does not guarantee that an error indication will be issued for every undeliverable data unit.

2.2.4 Acknowledged Connectionless-Mode Data Transfer

The acknowledged connectionless-mode data transfer services are designed for general use for the reliable transfer of data between peer DLS users. These services are intended for applications that require acknowledgment of data transfer between local area networks, but wish to avoid using the connection mode services. In-sequence delivery is guaranteed for data sent by the initiating station. The following services are supported:

- Acknowledged connectionless-mode data transfer service

  Provides for the exchange of DLSDUs which are acknowledged at the LLC sublayer.

- Quality of service (QOS) management service

  Enables a DLS user to specify the quality of service it can expect for each invocation of the connectionless data transfer service.

- Error reporting service

  Provides a means to notify a DLS user that a previously sent data unit either produced an error or could not be delivered. However, the error reporting service does not guarantee that an error indication will be issued for every undeliverable data unit.

2.3 DLPI Addressing

Each DLPI user must establish an identity to communicate with other data link users. This identity consists of the following pieces of information:

- Physical attachment identification

  This identifies the physical medium over which the DLS user communicates. The importance of identifying the physical medium is particularly evident on systems that are attached to multiple physical media. See Section 2.5 for information about identifying the available physical points of attachment (PPAs) on your system.

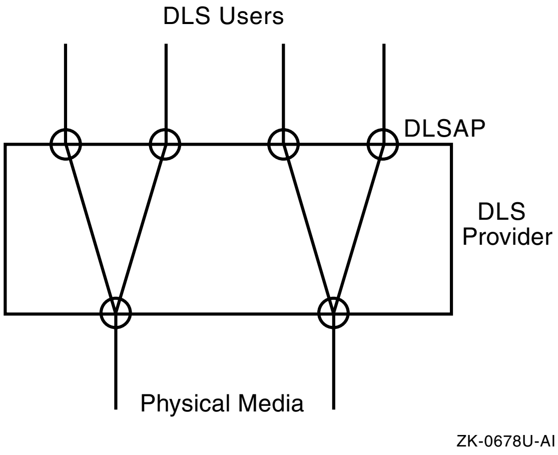
- Data link user identification

The DLS user must register with the DLS provider so that the provider can deliver protocol data units destined for that user.

The format of the DLSAP address is an unsigned character array containing the Medium Access Control (MAC) addresses followed by the bound Service Access Point (SAP). The SAP is usually two bytes in the case of Ethernet, or one byte in the case of ISO 8802-2 (IEEE 802.2). The one exception is when a HIERACHICAL `DL_SUBS_BIND_REQ` is processed. In that case, the DLSAP address consists of the MAC address, the SNAP SAP (0xAA), and a five-byte SNAP.

Figure 2–3 illustrates the components of this identification approach.

Figure 2–3: Identifying Components of a DLPI Address



The PPA is the point at which a system attaches itself to a physical communications medium. All communication on that physical medium funnels through the PPA. On systems where a DLS provider supports more than one physical medium, the DLS user must identify the medium through which it will communicate. A PPA is identified by a unique PPA identifier.

DLPI defines the following two styles of DLS provider, which are distinguished by the way they enable a DLS user to choose a particular PPA:

- The style 1 provider assigns a PPA based on the major/minor device the DLS user opened. A style 1 driver can be implemented so that it reserves a major device for each PPA the data link driver would support.

  This implementation of a style 1 driver allows the STREAMS clone open feature to be used for each PPA configured. Style 1 providers are appropriate when few PPAs are supported.

- The style 2 provider requires a DLS user to identify a PPA explicitly, using a special `attach` service primitive. For a style 2 driver, the `open` system call creates a Stream between the DLS user and DLS provider. Then, the attach primitive associates a particular PPA with that Stream. The format of the PPA identifier is specific to the DLS provider.

Tru64 UNIX supports only the style 2 provider because it is more suitable for supporting large numbers of PPAs.

2.4 DLPI Primitives

Table 2–1 lists and describes the DLPI primitives that are supported in the dlb STREAMS pseudodriver. For a complete list of DLPI primitives see the DLPI specification in the /usr/share/doclib/dlpi/dlpi.ps file.

Table 2–1: Supported DLPI Primitives

Primitive	Description
DL_ATTACH_REQ	Requests that the DLS provider associate a PPA with a Stream. Used on style 2 providers only.
DL_BIND_REQ	Requests that the DLS provider bind a DLSAP to the Stream. The DLS user must identify the address of the DLSAP to be bound to the Stream.
DL_BIND_ACK	Reports the successful bind of a DLSAP to a Stream, and returns the bound DLSAP address to the DLS user. Generated in response to a DL_BIND_REQ.
DL_DETTACH_REQ	Requests the DLS provider disassociate a PPA with a stream.
DL_DISABMULTI_REQ	Request the DLS provider disable the multicast address.
DL_ENABMULTI_REQ	Request the DLS provider enable a specific multicast address. (The current implementation of the DLB driver requires the state to be DL_IDLE.)
DL_ERROR_ACK	Informs DLS user of a previously issued request which was invalid.
DL_INFO_ACK	Response to DL_INFO_REQ primitive; conveys information about the DLPI stream.
DL_INFO_REQ	Requests the DLS provider return information about the DLPI stream.
DL_OK_ACK	Acknowledges to the DLS user that a previously issued request primitive was successfully received.
DL_PHYS_ADDR_REQ	Requests that the DLS provider return either the default (factory) or current value of the physical address associated with the Stream, depending upon the value of the address type selected in the request.

Table 2–1: Supported DLPI Primitives (cont.)

Primitive	Description
DL_PHYS_ADDR_ACK	Returns the value for the physical address to the link user in response to a DL_PHYS_ADDR_REQ.
DL_SUBS_BIND_ACK	Is the positive response to a DL_SUBS_BIND_REQ from the DLS provider.
DL_SUBS_BIND_REQ	Requests the DLS provider bind a subsequent DLSAP to stream. There are two classes of subsequent bind requests: HIERACHICAL and PEER. HIERACHICAL requests are only valid for SNAPs (see the IEEE 802.1 specification) and you must have bound to the SNAP SAP (0xAA) with a DL_BINDS_REQ before issuing the DL_SUBS_BIND_REQ for the SNAP. The PEER request binds to additional SAPs but does not change the DLSAP address of the stream.
DL_SUBS_UNBIND_REQ	Requests the DLS provider to unbind a SAP which was previously bound by a DL_SUBS_BIND_REQ.
DL_TEST_CON	Conveys that a DLSDU TEST response was received in response to a DL_TEST_REQ.
DL_TEST_IND	Conveys to the DLS user that a TEST command DLSDU was received.
DL_TEST_REQ	Requests the DLS provider to transmit a TEST command DLSDU on behalf of the DLS user.
DL_TEST_RES	Requests the DLS provider to send a TEST response command on behalf of the DLS user.
DL_UDERROR_IND	Informs a DLS user that a previously sent DL_UNITDATA_REQ failed.
DL_UNBIND_REQ	Requests that the DLS provider unbind the DLSAP that was bound by a previous DL_BIND_REQ from this Stream.
DL_UNITDATA_REQ	Conveys one DLSDU from the DLS user to the DLS provider for transmission to a peer DLS user.
DL_UNITDATA_IND	Conveys one DLSDU from the DLS provider to the DLS user.
DL_XID_CON	Conveys that a XID DLSDU was received in response to a DL_XID_REQ.
DL_XID_IND	Conveys to the DLS user that a XID DLSDU was received.

Table 2–1: Supported DLPI Primitives (cont.)

Primitive	Description
DL_XID_REQ	Requests the DLS provider to transmit a XID DLSDU on behalf of the DLS user.
DL_XID_RES	Requests the DLS provider to send a XID DLSDU on behalf of the DLS user. This is in repsonse to a DL_XID_RES.

2.5 Identifying Available PPAs

When compiled and run as root, the following program opens the STREAMS device /dev/streams/dlb and prints to the screen the PPAs available on the system. The PPA number should be passed in using the dl_ppa field of the DL_ATTACH_REQ DLPI primitive.

```
#include <sys/ioctl.h>
#include <stropts.h>
#include <errno.h>
#include <fcntl.h>

#define ND_GET ('N' << 8 + 0)
#define BUFSIZE 256

main()
{
        int i;
        int fd;
        char buf [BUFSIZE];
        struct strioctl stri;

        fd = open("/dev/streams/dlb", O_RDWR, 0);
        if (fd < 0) {
                perror("open");
                exit(1);
        }

        sprintf(buf, "dl_ifnames");
        stri.ic_cmd = ND_GET;
        stri.ic_timout = -1;
        stri.ic_len = BUFSIZE;
        stri.ic_dp = buf;

        if (ioctl(fd, I_STR, &stri) < 0) {
                perror("ioctl");
                exit(1);
        }

        printf("Valid PPA names on this system are:\n");
        for (i=0; i<stri.ic_len; i++)  {
                if (buf[i] == 0)
                        printf(" ");
                else
                        printf("%c",buf[i]);
        }
        printf("\n");
}
```

```
# a.out
Valid PPA names on this system are:
sscc0 (PPA 1) ln0 (PPA 2) dsy0 (PPA 3) dsy1 (PPA 4) \
sl0 (PPA 5) sl1 (PPA 6) lo0
#
```

3

X/Open Transport Interface

The X/Open Transport Interface (XTI) is a transport layer application interface that consists of a series of functions designed to be independent of the specific transport provider used. In this operating system, XTI is implemented according to the XPG3, XNS4.0, and XNS5.0 specifications. XNS4.0 is the default. (XPG3, to be retired in a future release, is provided for backward compatibility and is available by using a compiler switch. XNS5.0 is also available by using a compiler switch.) For more information about XPG3, XNS4.0 and XNS5.0, see the *X/Open Portability Guide Volume 7: Networking Services*, *X/Open CAE Specification: Networking Services, Issue 4*, and *X/Open CAE Specification: Networking Services (XNS), Issue 5*, respectively. This operating system's implementation of XTI is also thread safe.

Although similar in concept to the Berkeley socket interface, XTI is based on the AT&T Transport Layer Interface (TLI). TLI, in turn, is based on the **transport service** definition for the Open Systems Interconnection (OSI) model.

> **Note**
>
> This operating system includes the Transport Control Protocol (TCP) and User Datagram Protocol (UDP) transport providers. Although the information provided in this chapter applies to all transport providers that this operating system's XTI supports, such as DECnet/OSI, the examples are specific to TCP or UDP. For more specific information using XTI over TCP and UDP, see `xti_internet`(7). For examples and information specific to other transport providers, see the documentation that accompanies their software.

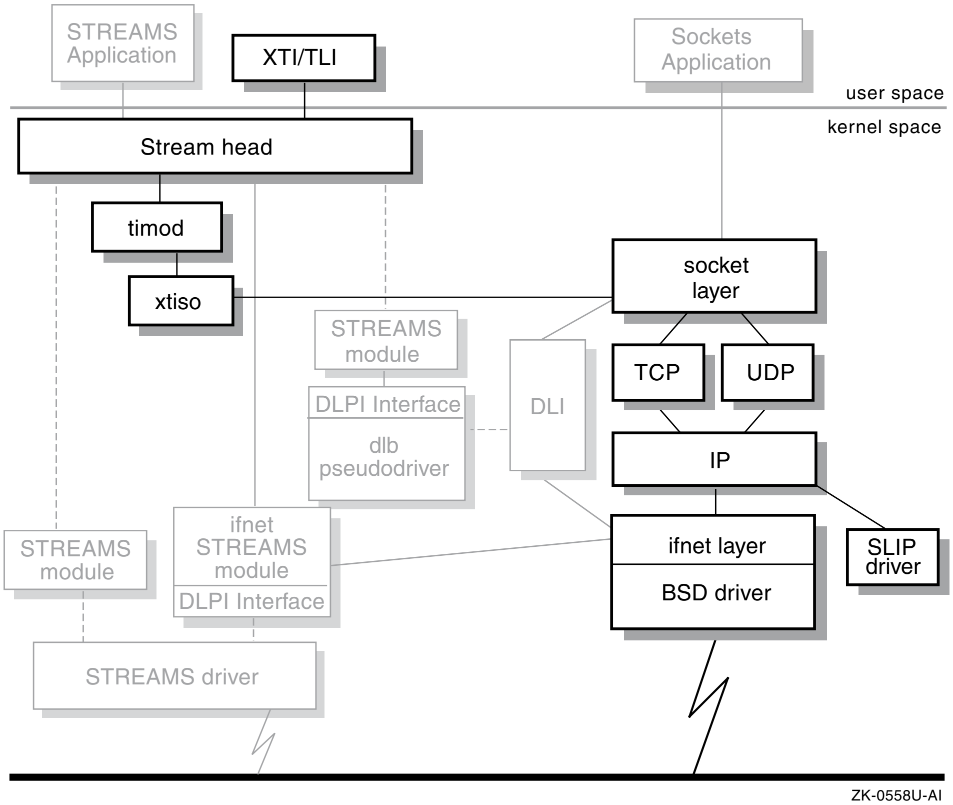
This chapter contains the following information:

- Overview of XTI
- Description of XTI features
- Instructions on how to use XTI
- Instructions on how to port applications to XTI
- Information on the differences among XPG3, XNS4.0, and XNS5.0.

- Explanation of XTI errors and error messages
- Information on configuring transport providers.

Figure 3–1 highlights XTI and its relationship to the operating system's implementation of the Internet Protocol suite. It also shows how XTI and the Internet Protocol suite fit into the rest of the network programming environment.

Figure 3–1: X/Open Transport Interface



3.1 Overview of XTI

XTI involves the interaction of the following entities:

- Transport providers

  A **transport provider** is a transport protocol, such as TCP or UDP, that offers transport layer services.

- Transport users

A **transport user** is an application program that requires the services of a transport provider to send data to or receive data from another program. A transport user communicates with a transport provider over a communications path identified by a transport endpoint.

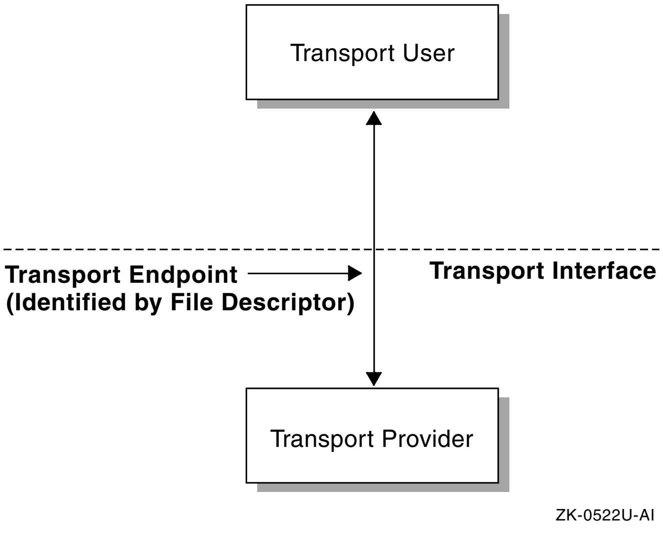
- Transport endpoints

  A **transport endpoint** is created when an application issues a `t_open` library call. All of the transport user's requests to the transport provider pass through the endpoint associated with that provider.

The transport user activates a transport endpoint by binding a transport address to it. Once an endpoint is active, a transport user can send data over it. The transport provider routes the data to the appropriate peer user or other destination.

When using a connection-oriented transport service, such as TCP, the transport user must establish a connection between itself and a peer transport user with a `t_connect` function, specifying an active endpoint, before sending data. In a transport connection, the transport user initiating the connection is the **active user**, or client, and the peer transport user responding to the connection request is the **passive user**, or server. Figure 3–2 summarizes the relationship between transport providers, transport users, and transport endpoints.

Figure 3–2: A Transport Endpoint



3.2 XTI Features

XTI consists of library calls, header files, and the rules and restrictions elaborating how XTI processes work and interact. This section describes the library calls and header files, as well as the regulations that govern the interaction between communicating processes.

3.2.1 Modes of Service and Execution

Transport users use different service modes and execution modes to determine how data is exchanged with transport providers. The following sections introduce the service modes and execution modes available in XTI.

3.2.1.1 Connection-Oriented and Connectionless Service

In XTI, an endpoint can support one of the following modes of service:

- Connection-oriented transport service

  A circuit-oriented service that transfers data over an established connection in a reliable, sequenced manner.

  Connection-oriented transport is useful for applications that require long, order dependent and reliable, stream-oriented interactions. With connection-oriented transport, transport users and providers can negotiate the parameters and options that govern data transfer. In addition, because a connection provides identification of both parties, the transport user avoids the overhead of transmitting and resolving addresses during data transfer. A connection also provides a context that logically relates successive units of data.

- Connectionless transport service

  A message-oriented service that transfers data in self-contained units or datagrams, which have no logical sequence with respect to one another.

  Connectionless transport is best suited for applications that have the following qualities:

  - Short-term request and response interactions
  - Dynamic reconfiguration of connections to multiple endpoints
  - No need for the guaranteed, sequential delivery of data

  Each data unit is self-contained and has no relationship to previous or successive data units, so the transport provider can route it independently.

3.2.1.2 Asynchronous and Synchronous Execution

Execution modes provide a means for transport users to handle completion of functions and receipt of events. An event is an occurrence or happening that is significant to a transport user. XTI supports two execution modes:

- Synchronous mode

  Waits for transport primitives to complete before returning control to the transport user. Also known as **blocking mode**.

  Synchronous mode is suited for applications that want to wait for functions to complete or maintain only a single transport connection. In synchronous mode, the transport user cannot perform other tasks while waiting for a function to complete. For example, if the transport user issues a `t_rcv` function in synchronous mode, `t_rcv` waits until data is received before returning control to the transport user.

  Even while using synchronous mode, it is possible to get some event notification, which the transport user does not ordinarily expect. Such asynchronous events are returned to the user through a special error, TLOOK.

  If an asynchronous event occurs while a function is executing, the function returns the TLOOK error; the transport user can then issue the `t_look` function to retrieve the event.

- Asynchronous mode

  Returns control to the transport user before transport primitives complete. Also known as **nonblocking mode**.

  Asynchronous mode is useful for applications that have long delays between completion of functions and other tasks to perform in the meantime. This mode is also useful for applications that handle multiple connections simultaneously. Many applications handle networking functions in asynchronous mode because they can perform useful work while waiting for particular networking functions to complete. For example, if a transport user issues a `t_rcv` function call in asynchronous mode, the function returns control to the user immediately if no data is available. The user periodically polls for data until the data arrives.

By default, all functions that process incoming events operate in synchronous mode, blocking until the task completes. To select asynchronous mode, the transport user specifies the O_NONBLOCK flag with the `t_open` function when the endpoint is created or before executing a function or group of functions with the `fcntl` operating system call.

For a full discussion of the specific events supported by XTI, see Section 3.2.3.

3.2.2 The XTI Library, TLI Library, and Header Files

XTI functions are implemented as part of the XTI library, `libxti.a`. TLI functions are implemented in a separate TLI library, `libtli.a`. There are also shared versions of these libraries, `libxti.so` and `libtli.so`.

Shared library support is provided by default when you link an XTI or TLI application with the XTI or TLI library.

The first of the following examples illustrates how to relink an XTI application's object files with the XTI shared library; the second illustrates how to relink a TLI application's object files with the TLI shared library:

```
% cc -o XTIapp XTIappmain.o XTIapputil.o -lxti

% cc -o TLIapp TLIappmain.o TLIapputil.o -ltli
```

To link programs statically with the XTI or TLI libraries, use the `non_shared` option to the `cc` command. The following example illustrates how to link an XTI application's object files to the XTI library statically:

```
% cc -non_shared -o XTIapp XTIappmain.o XTIapputil.o -lxti
```

See `cc`(1) for more information.

To make a program thread safe, build the program with POSIX threads `pthreads` routines. For more information, see the *Guide to the POSIX Threads Library*.

The few differences between XTI and TLI are described in Section 3.5.2, which also describes how to link your programs with the correct library at compile time.

3.2.2.1 XTI and TLI Header Files

XTI and TLI header files contain data definitions, structures, constants, macros, and options used by the XTI and TLI library calls. An application program must include the appropriate header file to make use of structures or other information a particular XTI or TLI library call requires. Table 3–1 lists the XTI and TLI header files.

Table 3–1: Header Files for XTI and TLI

File Name	Description
`<tiuser.h>`	Contains data definitions and structures for TLI applications. You must include this file for all TLI applications.
`<xti.h>`	Contains data definitions and structures for XTI applications. You must include this file for all XTI applications.
`<xti_inet.h>`	Contains Internet data definitions and structures for XTI applications. You must include this file for all XTI applications.

Table 3–1: Header Files for XTI and TLI (cont.)

File Name	Description
<xti_osi.h>	Contains ISO OSI data definitions and structures for XTI applications. You must include this file for all XTI applications.
<fcntl.h>	Defines flags for modes of execution for the t_open function. You must include this file for all XTI and TLI applications.

Note

Typically, header file names are enclosed in angle brackets (< >). To obtain the absolute path to the header file, prepend /usr/include/ to the information enclosed in the angle brackets. For example, the absolute path for the tiuser.h file is /usr/include/tiuser.h.

3.2.2.2 XTI Library Calls

Some of the calls apply to connection-oriented transport (COTS), some to connectionless transport (CLTS), some to connection-oriented transport when used with the orderly release feature (COTS_ORD), and some to all service modes. A small group of the calls are utility functions and do not apply to a particular service mode. Table 3–2 lists the name, purpose, and service mode of each XTI library call. Each call has an associated reference page by the same name.

The operating system provides XTI reference pages only; it does not provide TLI reference pages. For information about TLI and for the TLI reference pages see the *UNIX System V Programmer's Guide: Networking Interfaces*, which is issued by UNIX System Laboratories, Inc. The operating system provides reference pages for each of the functions. For more information, see the *X/Open CAE Specification: Networking Services.*

Table 3–2: XTI Library Calls

Name of Call	Purpose	Service Mode
t_accept	Accepts a connection request	COTS, COTS_ORD
t_alloc	Allocates memory for a library structure	All
t_bind	Binds an address to a transport endpoint	All
t_close	Closes a transport endpoint	All
t_connect	Establishes a connection with another transport user	COTS, COTS_ORD

Table 3–2: XTI Library Calls (cont.)

Name of Call	Purpose	Service Mode
`t_error`	Produces an error message	All
`t_free`	Frees memory previously allocated for a library structure	All
`t_getinfo`	Returns protocol-specific information	All
`t_getprotaddr` [a]	Returns the protocol address	All
`t_getstate`	Returns the current state for the transport endpoint	All
`t_listen`	Listens for a connection request	COTS, COTS_ORD
`t_look`	Returns the current event on the transport endpoint	All
`t_open`	Establishes a transport endpoint	All
`t_optmgmt`	Retrieves, verifies, or negotiates protocol options	All
`t_rcv`	Receives data or expedited data over a connection	COTS, COTS_ORD
`t_rcvconnect`	Receives the confirmation from a connection request	COTS, COTS_ORD
`t_rcvdis`	Identifies the cause of a disconnect, and retrieves information sent with a disconnect	COTS, COTS_ORD
`t_rcvrel` [b]	Acknowledges receipt of an orderly release indication	COTS_ORD
`t_rcvreldata` [c]	Receives an orderly release indication or confirmation containing user data	COTS_ORD
`t_rcvv` [c]	Receives data or expedited data over a connection and puts it into buffers	COTS, COTS_ORD
`t_rcvvudata` [c]	Receives a data unit into buffers	CLTS
`t_rcvudata`	Receives a data unit	CLTS
`t_rcvuderr`	Receives information about an error associated with a data unit	CLTS
`t_snd`	Sends data or expedited data over a connection	COTS, COTS_ORD

Table 3–2: XTI Library Calls (cont.)

Name of Call	Purpose	Service Mode
t_snddis	Initiates a release on an established connection, or rejects a connection request	COTS, COTS_ORD
t_sndrel [b]	Initiates an orderly release	COTS_ORD
t_sndreldata [c]	Initiates or responds to an orderly release with user data	COTS_ORD
t_sndudata	Sends a data unit	CLTS
t_sndv [c]	Sends data or expedited data on a connection	COTS, COTS_ORD
t_sndvudata [c]	Sends a data unit	CLTS
t_strerror [a]	Produces an error message string	All
t_sync	Synchronizes the data structures in the transport library	All
t_sysconf [c]	Retrieves configurable XTI variables	All
t_unbind	Disables a transport endpoint	All

[a] This function is supported in XNS4.0 only.
[b] Tru64 UNIX as supplied by Compaq Computer Corporation does not provide a transport provider that supports the use of COTS_ORD; therefore, this function returns an error.
[c] This function is supported in XNS5.0 only.

XTI supports an orderly release mechanism, `t_sndrel` and `t_rcvrel` functions. (See Table 3–2 for more information.) However, if your applications need to be portable to the ISO transport layer, we recommend that you do not use this mechanism.

Finally, the XTI header file defines the following constants to identify service modes:

- T_COTS – Connection-oriented transport service (for example, OSI transport)
- T_CLTS – Connectionless transport service (for example, UDP)
- T_COTS_ORD – Connection-oriented transport service with the orderly release mechanism implemented (for example, TCP)

These service modes are returned by the transport provider in the *servtype* field of the `info` structure when you create an endpoint with the `t_open` function.

3.2.3 Events and States

Each transport provider has a particular state associated with it, as viewed by the transport user. The state of a transport provider and its transition to the next allowable state is governed by outgoing and incoming events, which correspond to the successful return of specified user-level transport functions. Outgoing events correspond to functions that send a request or response to the transport provider, whereas incoming events correspond to functions that retrieve data or event information from the transport provider. This section describes the possible states of the transport provider, the outgoing and incoming events that can occur, and the allowable sequence of function calls.

3.2.3.1 XTI Events

XTI applications must manage asynchronous events. An asynchronous event is identified by a mnemonic which is defined as a constant in the XTI header file. Table 3–3 lists the name, purpose, and service mode for each type of asynchronous event in XTI.

Table 3–3: Asynchronous XTI Events

Event Name	Purpose	Service Mode
T_CONNECT	The transport provider received a connection response. This event usually occurs after the transport user issues the `t_connect` function.	COTS, COTS_ORD
T_DATA	The transport provider received **normal data**, which is all or part of a **Transport Service Data Unit** (TSDU).	COTS, CLTS, COTS_ORD
T_DISCONNECT	The transport provider received a disconnect request. This event usually occurs after the transport user issues data transfer functions, the `t_accept` function, or the `t_snddis` function.	COTS, COTS_ORD
T_EXDATA	The transport provider received expedited data.	COTS, COTS_ORD
T_GODATA	The flow control restrictions on the flow of normal data are lifted. The transport user can send normal data again.	COTS, CLTS, COTS_ORD

Table 3–3: Asynchronous XTI Events (cont.)

Event Name	Purpose	Service Mode
T_GOEXDATA	The flow control restrictions on the flow of expedited data are lifted. The transport user can send expedited data again.	COTS, COTS_ORD
T_LISTEN	The transport provider received a connection request from a remote user. This event occurs only when the file descriptor is bound to a valid address and no transport connection is established.	COTS, COTS_ORD
T_ORDREL	The transport provider received a request for an orderly release.	COTS_ORD
T_UDERR	An error was found on a datagram that was previously sent. This event usually occurs after the transport user issues the `t_rcvudata` or `t_unbind` functions.	CLTS

XTI stores all events that occur at a transport endpoint.

If using a synchronous mode of execution, the transport user returns from the function it was executing with a value of -1 and then checks for a value of TLOOK in `t_errno` and retrieves the event with the `t_look` function. In asynchronous mode, the transport user continues doing productive work and periodically checks for new events.

Every event at a transport endpoint is consumed by a specific XTI function, or it remains outstanding. Exceptions are the T_GODATA and T_GOEXDATA events, which are cleared by retrieving them with `t_look`. Thus, once the transport user receives a TLOOK error from a function, subsequent calls to that function or a different function continue to return the TLOOK error until the transport user consumes the event. Table 3–4 summarizes the consuming functions for each asynchronous event.

Table 3–4: Asynchronous Events and Consuming Functions

Event	Cleared by t_look	Consuming Function(s)
T_CONNECT	No	`t_connect`, `t_rcv-connect`
T_DATA	No	`t_rcv, t_rcvudata`
T_DISCONNECT	No	`t_rcvdis`
T_EXDATA	No	`t_rcv`

Table 3–4: Asynchronous Events and Consuming Functions (cont.)

Event	Cleared by t_look	Consuming Function(s)
T_GODATA	Yes	`t_snd, t_sndudata`
T_GOEXDATA	Yes	`t_snd`
T_LISTEN	No	`t_listen`
T_ORDREL	No	`t_rcvrel`
T_ORDRELDATA[a]	No	`t_rcvreldata`
T_UDERR	No	`t_rcvuderr`

[a] The Tru64 UNIX TCP transport provider does not support orderly release data.

Table 3–5 lists the events that cause a specific XTI function to return the TLOOK error. This information may be useful when you structure the event checking mechanisms in your XTI applications.

Table 3–5: XTI Functions that Return TLOOK

Function	Events Causing TLOOK
`t_accept`	T_DISCONNECT, T_LISTEN
`t_connect`	T_DISCONNECT, T_LISTEN [a]
`t_listen`	T_DISCONNECT [b]
`t_rcv`	T_DISCONNECT, T_ORDREL [c]
`t_rcvconnect`	T_DISCONNECT
`t_rcvrel`	T_DISCONNECT
`t_rcvreldata`	T_DISCONNECT
`t_rcvudata`	T_UDERR
`t_rcvv`	T_DISCONNECT, T_ORDREL
`t_rcvvudata`	T_UDERR
`t_snd`	T_DISCONNECT, T_ORDREL
`t_sndv`	T_DISCONNECT, T_ORDREL
`t_snddis`	T_DISCONNECT
`t_sndrel`	T_DISCONNECT
`t_sndreldata`	T_DISCONNECT
`t_sndudata`	T_UDERR
`t_sndvudata`	T_UDERR
`t_unbind`	T_LISTEN, T_DATA [d]

[a] This event occurs only when `t_connect` is issued for an endpoint that was bound with a *qlen* > 0, and has a pending connection indication.

Table 3–5: XTI Functions that Return TLOOK (cont.)

[b] This event indicates a disconnect on an outstanding connection indication.
[c] This occurs only when all pending data has been read.
[d] T_DATA may only occur for the connectionless mode.

Each XTI function manages one transport endpoint at a time. It is not possible to wait for several events from different sources, particularly from several transport connections at a time. This implementation of XTI allows the transport user to monitor input and output on a set of file descriptors with the `poll` function. See `poll`(2) for more information.

3.2.3.2 XTI States

XTI controls the legality of the calls issued by a program at a given point in time. XTI uses eight states to manage communication over a transport endpoint. Both the active and passive user have a unique state that reflects the function in process.

Table 3–6 describes the purpose of each XTI state. A service mode of COTS indicates the state occurs regardless of whether or not orderly service is implemented. A service mode of COTS_ORD indicates the state occurs only when orderly service is implemented.

Table 3–6: XTI States

State	Description	Service Mode
T_UNINIT	Uninitialized. Initial and final state of the interface. To establish a transport endpoint, the user must issue a `t_open`.	COTS, CLTS, COTS_ORD
T_UNBIND	Unbound. The user can bind an address to a transport endpoint or close a transport endpoint.	COTS, CLTS, COTS_ORD
T_IDLE	Idle. The active user can establish a connection with a passive user (COTS), disable a transport endpoint (COTS, CLTS), or send and receive data units (CLTS). The passive user can listen for a connection request (COTS).	COTS, CLTS, COTS_ORD
T_OUTCON	Outgoing connection pending. The active user can receive confirmations for connection requests.	COTS, COTS_ORD
T_INCON	Incoming connection pending. The passive user can accept connection requests.	COTS, COTS_ORD

Table 3–6: XTI States (cont.)

State	Description	Service Mode
T_DATAXFER	Data transfer. The active user can send data to and receive data from the passive user. The passive user can send data to and receive data from the active user.	COTS, COTS_ORD
T_OUTREL	Outgoing orderly release. The user can respond to an orderly release indication.	COTS_ORD
T_INREL	Incoming orderly release. The user can send an orderly release indication.	COTS_ORD

If you are writing a connection-oriented application, note that your program can release a connection at any time during the connection-establishment state or data-transfer state.

3.2.4 Tracking XTI Events

The XTI library keeps track of outgoing and incoming events to manage the legal states of transport endpoints. The following sections describe these outgoing and incoming events.

3.2.4.1 Outgoing Events

Outgoing events are caused by XTI functions that send a request or response to the transport provider. An outgoing event occurs when a function returns successfully. Some functions produce different events, depending on the following values:

ocnt A count of outstanding connection indications (those passed to the transport user but not yet accepted or rejected). This count is only meaningful for the current transport endpoint (*fd*). A count of outstanding connection indications (those passed to the transport user but not yet accepted or rejected). This count is only meaningful for the current transport endpoint (*fd*).

fd The file descriptor of the current transport endpoint.

resfd The file descriptor of the endpoint where a connection will be accepted.

Table 3–7 describes the outgoing events available in XTI. A service mode of COTS indicates the event occurs for a connection-oriented service regardless of whether or not orderly service is implemented. A service mode of COTS_ORD indicates the event occurs only when orderly service is implemented.

Table 3–7: Outgoing XTI Events

Event	Description	Service Mode
opened	Successful return of `t_open` function.	COTS, CLTS, COTS_ORD
bind	Successful return of `t_bind` function.	COTS, CLTS, COTS_ORD
optmgmt	Successful return of `t_optmgmt` function.	COTS, CLTS, COTS_ORD
unbind	Successful return of `t_unbind` function.	COTS, CLTS, COTS_ORD
closed	Successful return of `t_close` function.	COTS, CLTS, COTS_ORD
connect1	Successful return of `t_connect` function in synchronous execution mode.	COTS, COTS_ORD
connect2	The `t_connect` function returned the TNODATA error in asynchronous mode, or returned the TLOOK error because a disconnect indication arrived on the transport endpoint.	COTS, COTS_ORD
accept1	Successful return of `t_accept` function, where *ocnt* == 1 and *fd* == *resfd*.	COTS, COTS_ORD
accept2	Successful return of `t_accept` function, where *ocnt* == 1 and *fd* != *resfd*.	COTS, COTS_ORD
accept3	Successful return of `t_accept` function, where *ocnt* > 1.	COTS
snd	Successful return of `t_snd` function.	COTS
snddis1	Successful return of `t_snddis` function, where *ocnt* <= 1.	COTS, COTS_ORD
snddis2	Successful return of `t_snddis` function, where *ocnt* > 1.	COTS, COTS_ORD
sndrel	Successful return of `t_sndrel` function.	COTS_ORD
sndudata	Successful return of `t_sndudata` function.	CLTS

3.2.4.2 Incoming Events

Incoming events are caused by XTI functions that retrieve data or events from the transport provider. An incoming event occurs when a function returns successfully. Some functions produce different events, depending on the value of the *ocnt* variable. This variable is a count of outstanding connection indications (those passed to the transport user but not yet accepted or rejected). This count is only meaningful for the current transport endpoint (*fd*).

The pass_conn incoming event is not associated directly with the successful return of a function on a given endpoint. The pass_conn event occurs on the endpoint that is being passed a connection from the current endpoint. No function occurs on the endpoint where the pass_conn event occurs.

Table 3–8 describes the incoming events available in XTI. A service mode of COTS indicates the event occurs regardless of whether or not orderly service is implemented. A service mode of COTS_ORD indicates the event occurs only when orderly service is implemented.

Table 3–8: Incoming XTI Events

Event	Description	Service Mode
listen	Successful return of the `t_listen` function	COTS, COTS_ORD
rcvconnect	Successful return of the `t_rcvconnect` function	COTS, COTS_ORD
rcv	Successful return of the `t_rcv` function	COTS, COTS_ORD
rcvdis1	Successful return of the `t_rcvdis` function, where *ocnt* == 0	COTS, COTS_ORD
rcvdis2	Successful return of the `t_rcvdis` function, where *ocnt* == 1	COTS, COTS_ORD
rcvdis3	Successful return of the `t_rcvdis` function, where *ocnt* > 1	COTS, COTS_ORD
rcvrel	Successful return of the `t_rcvrel` function	COTS_ORD
rcvudata	Successful return of the `t_rcvudata` function	CLTS
rcvuderr	Successful return of the `t_rcvuderr` function	CLTS
pass_conn	Successfully received a connection that was passed from another transport endpoint	COTS, COTS_ORD

3.2.5 Map of XTI Functions, Events, and States

This section describes the relationship among XTI functions, outgoing and incoming events, and states. Since XTI has well-defined rules about state transitions, it is possible to know the next allowable state given the current state and most recently received event. This section provides detailed tables that map the current event and state to the next allowable state.

This section excludes the `t_getstate`, `t_getinfo`, `t_alloc`, `t_free`, `t_look`, `t_sync`, and `t_error` functions from discussions of state transitions. These utility functions do not affect the state of the transport interface, so they can be issued from any state except the uninitialized (T_UNINIT) state.

To use Table 3–9, Table 3–10, and Table 3–11, find the row that matches the current incoming or outgoing event and the column that matches the current state. Go to the intersection of the row and column to find the next allowable state. A dash (—) at the intersection indicates an invalid combination of event and state. Some state transitions are marked by a letter that indicates an action that the transport user must take. The letters and their meanings are listed at the end of the appropriate table.

Table 3–9 shows the state transitions for initialization and deinitialization functions, functions that are common to both the connection-oriented and connectionless modes of service. For example, if the current event and state are bind and T_UNBND, the next allowable state is T_IDLE. In addition, the transport user must set the count of outstanding connection indications to zero, as indicated by the letter a.

Table 3–9: State Transitions for Initialization of Connection-Oriented or Connectionless Transport Services

Event	T_UNINIT State	T_UNBND State	T_IDLE State
opened	T_UNBND	—	—
bind	—	T_IDLE [a]	—
unbind	—	—	T_UNBND
closed	—	T_UNINIT	T_UNINIT

[a] Set the count of outstanding connection indications, *ocnt*, to 0.

Table 3–10 shows the state transitions for data transfer functions in connectionless transport services.

Table 3–10: State Transitions for Connectionless Transport Services

Event	State T_IDLE
sndudata	T_IDLE
rcvudata	T_IDLE
rcvuderr	T_IDLE

Table 3–11 and Table 3–12 show the transitions for connection, release, and data transfer functions in connection-oriented transport services for incoming and outgoing events. For example, if the current event and state are `accept2` and T_INCON, the next allowable state is T_IDLE, providing the transport user decrements the count of outstanding connection indications and passes a connection to another transport endpoint.

Table 3–11: State Transitions for Connection-Oriented Transport Services: Part 1

Event	T_IDLE State	T_OUTCON State	T_INCON State	T_DATAXFER State
connect1	T_DATAXFER	—	—	—
connect2	T_OUTCON	—	—	—
rcvconnect	—	T_DATAXFER	—	—
listen	T_INCON [a]	—	T_INCON [a]	—
accept1	—	—	T_DATAXFER [b]	—
accept2	—	—	T_IDLE [b c]	—
accept3	—	—	T_INCON [b c]	—
snd	—	—	—	T_DATAXFER
rcv	—	—	—	T_DATAXFER
snddis1	—	T_IDLE	T_IDLE [b]	T_IDLE
snddis2	—	—	T_INCON [b]	—
rcvdis1	—	T_IDLE	—	T_IDLE
rcvdis2	—	—	T_IDLE [b]	—
rcvdis3	—	—	T_INCON [b]	—
sndrel	—	—	—	T_OUTREL
rcvrel	—	—	—	T_INREL
pass_conn	T_DATAXFER	—	—	—
optmgmt	T_IDLE	T_OUTCON	T_INCON	T_DATAXFER
closed	T_UNINIT	T_UNINIT	T_UNINIT	T_UNINIT

[a] Increment the count of outstanding connection indications.
[b] Decrement the count of outstanding connection indications.
[c] Pass a connection to another transport endpoint, as indicated in the `t_accept` function.

Table 3–12: State Transitions for Connection-Oriented Transport Services: Part 2

Event	T_OUTREL State	T_INREL State	T_UNBND State
connect1	—	—	—
connect2	—	—	—
rcvconnect	—	—	—
listen	—	—	—
accept1	—	—	—

Table 3–12: State Transitions for Connection-Oriented Transport Services: Part 2 (cont.)

Event	T_OUTREL State	T_INREL State	T_UNBND State
accept2	—	—	—
accept3	—	—	—
snd	—	T_INREL	—
rcv	T_OUTREL	—	—
snddis1	T_IDLE	T_IDLE	—
snddis2	—	—	—
rcvdis1	T_IDLE	T_IDLE	—
rcvdis2	—	—	—
rcvdis3	—	—	—
sndrel	—	T_IDLE	—
rcvrel	T_IDLE	—	—
pass_conn	—	—	T_DATAXFER
optmgmt	T_OUTREL	T_INREL	T_UNBND
closed	T_UNINIT	T_UNINIT	—

3.2.6 Synchronization of Multiple Processes and Endpoints

In general, if you use multiple processes, you need to synchronize them carefully to avoid violating the state of the interface.

Although transport providers treat all transport users of a transport endpoint as a single user, the following situations are possible:

- One process can create several transport endpoints simultaneously.
- Multiple processes can share a single endpoint simultaneously.

For a single process to manage several endpoints in synchronous execution mode, the process must manage the actions on each endpoint serially instead of in parallel. Optionally, you can write a server to manage several endpoints at once. For example, the process can listen for an incoming connection indication on one endpoint and accept the connection on a different endpoint, so as not to block incoming connections. Then, the application can fork a child process to service the requests from the new connection.

Multiple processes that share a single endpoint must coordinate actions to avoid violating the state of the interface. To do this, each process calls the `t_sync` function, which retrieves the current state of the transport provider, before issuing other functions. If all processes do not cooperate in this manner, another process or an incoming event can change the state of the interface.

Similarly, while several endpoints can share the same protocol address, only one can listen for incoming connections. Other endpoints sharing the protocol address can be in data transfer state or in the process of establishing a connection without causing a conflict. This means that an address can have only one server, but multiple endpoints can call the address at the same time.

3.3 Using XTI

This section presents guidelines to help you sequence functions, manage states, and use XTI options. It then describes the steps required to write both connection-oriented and connectionless programs to XTI.

3.3.1 Guidelines for Sequencing Functions

Figure 3–3 shows the typical sequence of functions and state transitions for an active user and passive user communicating with a connection-oriented transport service in nonblocking mode. The solid lines in the figure show the state transitions for the active user, while the dashed lines show the transitions for the passive user. Each line represents the call of a function, while each ellipse represents the resulting state. This example does not include the orderly release feature.

Figure 3–3: State Transitions for Connection-Oriented Transport Services

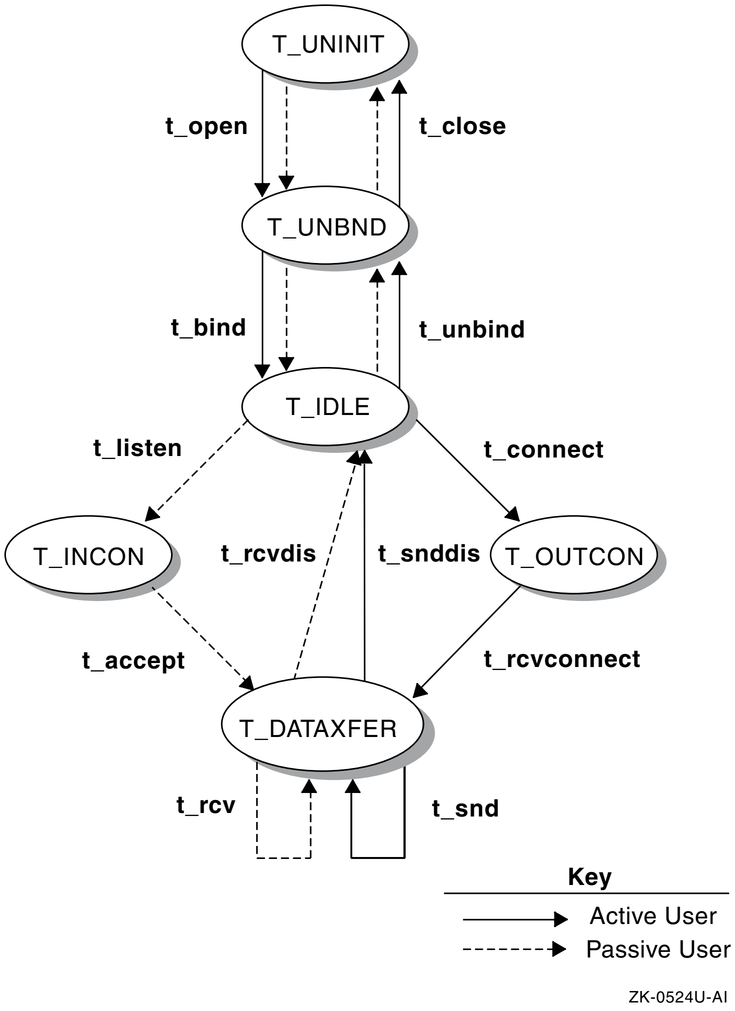
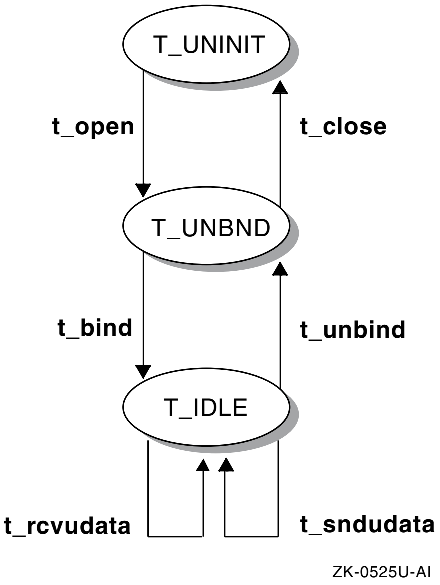


Figure 3–4 shows the typical sequence of functions and transitions in state for two users communicating with the connectionless transport service. Each line in the figure represents the call of a function, while each ellipse represents the resulting state. Both users are represented by solid lines.

Figure 3–4: State Transitions for the Connectionless Transport Service



3.3.2 State Management by the Transport Provider

All transport providers take the following actions with respect to states:

- Keep a record of the state of the interface as seen by the transport user.
- Reject any requests or responses that would place the interface out of state and return an error. In this case, the state does not change. For example, if the user passes data with a function and the interface is not in T_DATAXFER state, the transport provider does not accept or forward the data.

The uninitialized state (T_UNINIT) serves two purposes:

- The initial state of a transport endpoint. The transport user must initialize and bind the transport endpoint before the transport provider views it as active.
- The final state of a transport endpoint. The transport provider must view the endpoint as unused. When the transport user issues the `t_close` function, the transport provider is closed, and the resources associated with the transport library are freed for use by another endpoint.

3.3.3 Writing a Connection-Oriented Application

Follow these steps to write a connection-mode application:

1. Initialize an endpoint

2. Establish a connection
3. Transfer data
4. Release a connection
5. Deinitialize an endpoint

3.3.3.1 Initializing an Endpoint

To initialize an endpoint, complete the following steps:

1. Open the endpoint
2. Bind an address to the endpoint
3. Negotiate protocol options

Note that the steps described here for initializing an endpoint for connection-oriented service are identical for connectionless service.

Opening a Transport Endpoint

Both connection-oriented and connectionless applications must open a transport endpoint using the `t_open` function.

See `t_open`(3) for function syntax, parameters, and errors. See Section 3.7 for a general description of XTI errors.

This XTI implementation uses pathnames to device special files to identify transport providers, which is the same method as in the AT&T TLI. The device special files corresponding to TCP or UDP transport providers reside in the `/dev/streams/xtiso` directory. If you use a different transport provider, see its documentation for the correct device name.

Note

> Using the special device with any mechanism other than XTI/TLI, for example, direct `open`, `read`, or `write` calls, is illegal and will generate undefined results.

The XTI specification forbids the use of O_RDONLY or O_WRONLY to make the endpoint either read-only or write-only as expected.

If you are designing a protocol-independent program, you can determine data buffer sizes by accessing the information that the `t_open` function returns about the *`t_info`* structure. If the transport user exceeds the allowed data size, you receive an error. Alternatively, you can use the `t_alloc` function to allocate data buffers.

The following is an example of the t_open function for the TCP transport provider (XNS4.0):

```
if ( (newfd = t_open( "/dev/streams/xtiso/tcp+" , O_RDWR , NULL) ) == -1 )
     {
       (void) t_error("could not open tcp transport");
       exit (1);
     }
```

Binding an Address to the Endpoint

Once you open an endpoint, you need to bind a protocol address to the endpoint by using the t_bind function. By binding the address, you activate the endpoint. In connection mode, you also direct the transport provider to begin accepting connection indications or servicing connection requests on the transport endpoint. To determine if the transport provider has accepted a connection indication, you can issue the t_listen function. In connectionless mode, once you bind the address, you can send or receive data units through the transport endpoint.

See t_bind(3) for function syntax, parameters, and errors. See Section 3.7 for a general description of XTI errors.

To determine if the transport provider generates addresses, do not specify one in the t_bind function (set *req* to a null pointer). If the transport provider supplies addresses, the function returns an assigned address in the ret field. If the transport provider does not supply addresses, the function returns an error of TNOADDR.

If you accept a connection on an endpoint that is used for listening for connection indications, the bound address is busy for the duration of the connection. You cannot bind any other endpoint for listening on that same address while the initial listening endpoint is actively transferring data or in T_IDLE state.

You can use the gethostbyname routine, described in Section 4.2.3.2, to obtain host information when either TCP or UDP is the underlying transport provider.

If you use a method to retrieve host information other than the gethostbyname routine, consider the following:

- Your applications must pass XTI functions a socket address in the format that the transport provider expects. For XTI over TCP/IP, the expected address format is a sockaddr_in structure.
- Your applications also need to pass a transport provider identifier to XTI functions. The operating system expects this identifier to be in the format of a pathname to the device special file for the transport provider.

3.3.3.2 Using XTI Options

XPG3 , XNS4.0, and XNS5.0 implement option management differently.

In XPG3, option management is handled exclusively by the t_optmgmt function. In XNS4.0 and XNS5.0, several functions contain an *opt* argument which is used to convey options between a transport user and the transport provider.

For more information, see Section 3.6.7.

3.3.3.3 Establishing a Connection

The connection establishment phase typically consists of the following actions:

1. A passive user, or server, listens for a connection request.
2. An active user, or client, initiates a connection.
3. A passive user, or server, accepts a connection request and a connection indication is received.

These steps are described in the following sections.

Listening for Connection Indications

The passive user issues the t_listen function to look for enqueued connection indications. If the t_listen function finds a connection indication at the head of the queue, it returns detailed information about the connection indication and a local sequence number that identifies the indication. The number of outstanding connection indications that can be queued is limited by the value of the *qlen* parameter that was accepted by the transport provider when the t_bind function was issued.

By default, the t_listen function executes synchronously by waiting for a connection indication to arrive before returning control to the user. If you set the O_NONBLOCK flag of the t_open function or the fcntl function for asynchronous execution, the t_listen function checks for an existing connection indication and returns an error of TNODATA if none is available.

See t_listen(3) for function syntax, parameters, and errors. See Section 3.7 for a general description of XTI errors.

Initiating Connections

A connection is initiated in either synchronous or asynchronous mode. In synchronous mode, the active user issues the t_connect function, which waits for the passive user's response before returning control to the active user. In asynchronous mode, t_connect initiates a connection but returns control to the active user before a response to the connection arrives. Then, the active user can determine the status of the connection request by issuing the t_rcvconnect function. If the passive user accepted the request, the t_rcvconnect function returns successfully and the connection establishment phase is complete. If a response has not been received yet, the t_rcvconnect function returns an error of TNODATA. The active user should issue the t_rcvconnect function again later.

See t_connect(3) for function syntax, parameters, and errors. See Section 3.7 for a general description of XTI errors.

Accepting Connections

When the passive user accepts a connection indication, it can issue the t_accept function on the same endpoint (the endpoint where it has been listening with t_listen) or a different endpoint.

If the passive user accepts on the same endpoint, the endpoint can no longer receive and enqueue incoming connection indications. The protocol address that is bound to the endpoint remains busy for the duration it is active. No other transport endpoints can be bound to the same protocol address as the listening endpoint. That is, no other endpoints can be bound until the passive user issues the t_unbind function. Further, before the connection can be accepted on the same endpoint, the passive user must respond (with either the t_accept or t_snddis functions) to all previous connection indications that it has received. Otherwise, t_accept returns an error of TBADF.

If the passive user accepts the connection on a different endpoint, the listening endpoint can still receive and enqueue incoming connection requests. The different endpoint must already be bound to a protocol address and be in the T_IDLE state. If the protocol address is the same as for the endpoint where the indication was received, the *qlen* parameter must be set to zero (0).

For both types of endpoints, t_accept will fail and return an error of TLOOK if there are connect or disconnect indications waiting to be received.

See t_accept(3) for function syntax, parameters, and errors. See Section 3.7 for a general description of XTI errors.

3.3.3.4 Transferring Data

Once a connection is established between two endpoints, the active and passive users can transfer data in full-duplex fashion over the connection. This phase of connection-oriented service is known as the data transfer phase. The following sections describe how to send and receive data during the data transfer phase.

Sending Data

Transport users can send either normal or expedited data over a connection with the t_snd function. Normally, t_snd sends successfully and returns the number of bytes accepted if the transport provider can immediately accept all the data. If the data cannot be accepted immediately, the result of t_snd depends on whether it is executing synchronously or asynchronously.

By default, the t_snd function executes synchronously and waits if flow control conditions prevent the transport provider from accepting the data. The function blocks until one of the following conditions becomes true:

- The flow control conditions clear, and the transport provider can accept a new data unit. The t_snd function returns successfully.
- A disconnect indication is received. The t_snd function returns with an error of TLOOK. If you call the t_look function, it returns the T_DISCONNECT event. Any data in transit is lost.
- An internal problem occurs. The t_snd function returns with an error of TSYSERR. Any data in transit is lost.

If the O_NONBLOCK flag was set when the endpoint was created, t_snd executes asynchronously and fails immediately if flow control restrictions exist. In some cases, only part of the data was accepted by the transport provider, so t_snd returns a value that is less than the number of bytes that you requested to be sent. At this point, you can do one of the following:

- Issue t_snd again with the remaining data.
- Check with the t_look function to see if the flow control restrictions are lifted, then resend the data. The t_look function is described at the end of this chapter.

See t_snd(3) for function syntax, parameters, and errors. See Section 3.7 for a general description of XTI errors.

Receiving Data

Transport users can receive either normal or expedited data over a connection with the t_rcv function. Typically, if data is available, t_rcv returns the data. If the connection has been disconnected, t_rcv returns immediately with an error. If data is not available, but the connection still exists, t_rcv behaves differently depending on the mode of execution:

- By default, t_rcv executes synchronously and waits for one of the following to arrive:
  - Data
  - A disconnect indication
  - A signal

Instead of issuing t_rcv and waiting, you can issue the t_look function and check for the T_DATA or T_EXDATA events.

– If you set the O_NONBLOCK flag, t_rcv executes asynchronously and fails with an error of TNODATA if no data is available. You should continue to poll for data by issuing the t_rcv or t_look functions.

See t_rcv(3) for function syntax, parameters, and errors. See Section 3.7 for a general description of XTI errors.

3.3.3.5 Releasing Connections

XTI supports two ways to release connections: abortive release and orderly release. All transport providers support abortive release. Orderly release is not provided by all transport providers. For example, the OSI transport supports only abortive release, while TCP supports abortive release and optionally, orderly release.

Abortive Release

An abortive release, which can be requested by the transport user or the transport provider, aborts a connection immediately. Abortive releases cannot be negotiated, and once the abortive release is requested, there is no guarantee that user data will be delivered.

Transport users can request an abortive release in either the connection establishment or data transfer phases. During connection establishment, a transport user can use the abortive release to reject a connection request. In data transfer phase, either user can release the connection at any time. If a transport provider requests an abortive release, both users are informed that the connection no longer exists.

See t_snddis(3) for function syntax, parameters, and errors. See Section 3.7 for a general description of XTI errors.

Transport users are notified about abortive releases through the T_DISCONNECT event. If your program receives a T_DISCONNECT event, it must issue the t_rcvdis function to retrieve information about the disconnect and to consume the T_DISCONNECT event.

See t_rcvdis(3) for function syntax, parameters, and errors. See Section 3.7 for a general description of XTI errors.

Orderly Release

An orderly release allows for release of a connection without loss of data. Orderly release is not provided by all transport providers. If the transport provider returned a service type of T_COTS_ORD with the t_open or t_getinfo functions, orderly release is supported. Transport users can request an orderly release during the data transfer phase. The typical sequence of orderly release is as follows:

1. The active user issues the t_sndrel function to request an orderly release of the connection.
2. The passive user receives the T_ORDREL event indicating the active user's request for the orderly release and issues the t_rcvrel function to indicate the request was received and consume the T_ORDREL event.
3. When ready to disconnect, the passive user issues the t_sndrel function.
4. The active user responds by issuing the t_rcvrel function.

The transport user cannot send more data over the connection after it issues the t_sndrel function. The transport user can, however, continue to receive data until it receives an orderly release indication (the T_ORDREL event).

See t_sndrel(3) for function syntax, parameters, and errors. See Section 3.7 for a general description of XTI errors.

To acknowledge the receipt of an orderly release indication, issue the t_rcvrel function with the following syntax:

After a transport user receives an orderly release indication (T_ORDREL), it cannot receive more data. (If the user attempts to do so, the function blocks indefinitely.) The transport user can, however, continue to send data until it issues the t_sndrel function.

See t_rcvrel(3) for function syntax, parameters, and errors. See Section 3.7 for a general description of XTI errors.

3.3.3.6 Deinitializing Endpoints

When you are finished using an endpoint, you deinitialize it by unbinding and closing the endpoint with the t_unbind and t_close functions. Note that the steps described here for deinitializing an endpoint with connection-oriented service are identical to those for connectionless service.

When you unbind the endpoint, you disable the endpoint so that the transport provider no longer accepts requests for it.

See t_unbind(3) for function syntax, parameters, and errors. See Section 3.7 for a general description of XTI errors.

By closing the endpoint, you inform the transport provider that you are finished with it and you free any library resources associated with the endpoint.

You should call t_close when the endpoint is in the T_UNBND state. However, this function does not check state information, so it may be called to close a transport endpoint from any state.

If you close an endpoint that is not in the T_UNBND state, the library resources associated with the endpoint are freed automatically, and the file associated with the endpoint is closed. If there are no other descriptors in this or any other process that references the endpoint, the transport connection is broken.

See t_close(3) for function syntax, parameters, and errors. See Section 3.7 for a general description of XTI errors.

3.3.4 Writing a Connectionless Application

This section describes the steps required to write a connectionless mode application:

1. Initializing an endpoint
2. Transferring data
3. Deinitializing an endpoint

3.3.4.1 Initializing an Endpoint

Initializing an endpoint for connection-oriented and connectionless applications is the same. See Section 3.3.3.1 for information on how to initialize an endpoint for a CLTS application.

3.3.4.2 Transferring Data

The data transfer phase of connectionless service consists of the following:

- Sending data to other users
- Receiving data from other users
- Retrieving error information about previously sent data

Note that connectionless service:

- Does not support expedited data
- Reports only the T_UDERR, T_DATA, and T_GODATA events

Sending Data

The t_sndudata function can execute synchronously or asynchronously. When executing synchronously, t_sndudata returns control to the user when the transport provider can accept another datagram. In some cases, the function blocks for some time until this occurs. In asynchronous mode, the transport provider refuses to send a new datagram if flow control restrictions exist. The `t_sndudata` function returns an error of TFLOW, and you must either try again later or issue the `t_look` function to see when the flow control restriction is lifted, which is indicated by the T_GODATA or T_GOEXDATA events.

If you attempt to send a data unit before you activate the endpoint with the `t_bind` function, the transport provider discards the data.

See `t_sndudata`(3) for function syntax, parameters, and errors. See Section 3.7 for a general description of XTI errors.

Receiving Data

When you call the `t_rcvudata` function and data is available, `t_rcvudata` returns immediately indicating the number of octets received. If data is not available, `t_rcvudata` behaves differently depending on the mode of execution, as follows:

- Synchronous mode

  The `t_rcvudata` function blocks until either a datagram, error, or signal is received. As an alternative to waiting for `t_rcvudata` to return, you can issue the `t_look` function periodically for the T_GODATA event, and then issue `t_rcvudata` to receive the data.

- Asynchronous mode

  The `t_rcvudata` function returns immediately with an error. You then must either retry the function periodically or poll for incoming data with the `t_look` function.

See `t_rcvudata`(3) for function syntax, parameters, and errors. See Section 3.7 for a general description of XTI errors.

Retrieving Error Information

If you issue the `t_look` function and receive the T_UDERR event, previously sent data has generated an error. To clear the error and consume the T_UDERR event, you should issue the `t_rcvuderr` function. This function also returns information about the data that caused the error and the nature of the error, if you want.

To receive an error indication with information about data, issue the `t_rcvuderr` function with the following syntax:

`t_rcvuderr`*`(fd,uderr)`***`;`**

See `t_rcvuderr`(3) for function syntax, parameters, and errors. See Section 3.7 for a general description of XTI errors.

3.3.4.3 Deinitializing Endpoints

Deinitializing an endpoint for connection-oriented and connectionless applications is the same. See Section 3.3.3.6 for information on how to deinitialize an endpoint for a connectionless application.

3.4 Phase-Independent Functions

XTI provides a number of functions that can be issued during any phase of connection-oriented or connectionless service (except the uninitialized state) and do not affect the state of the interface. Table 3–13 lists and briefly describes these functions.

Table 3–13: Phase-Independent Functions

Function	Description
t_getinfo	Returns information about the characteristics of the transport provider associated with the endpoint.
t_getprotaddr [a]	Returns the protocol address.
t_getstate	Returns the current state of the endpoint.
t_strerror [a]	Produces an error message string.
t_sync	Synchronizes the data structures managed by the transport library with information from the transport provider.
t_sysconf [b]	Returns the current value of configurable XTI limits or options.
t_alloc	Allocates storage for a specified data structure.
t_free	Frees storage for a data structure that was previously allocated by t_alloc.
t_error	Prints a message describing the last error returned by an XTI function. (Optional)
t_look	Returns the current event associated with the endpoint.

[a] This function is supported in XNS4.0 only.
[b] This function is supported in XNS5.0 only.

The t_getinfo and t_getstate functions can be useful for retrieving important information. The t_getinfo function returns the same information about the transport provider as t_open. It offers the advantage that you can call it during any phase of communication, whereas you can call t_open only during the initialization phase. If a function returns the TOUTSTATE error to indicate that the endpoint is not in the proper state, you can issue t_getstate to retrieve the current state and take action appropriate for the state.

The t_sync function can do the following:

- Synchronize data structures managed by the transport library with information from the underlying transport provider.
- Permit two cooperating processes to synchronize their interaction with a transport provider.

The `t_alloc` and `t_free` functions are convenient for allocating and freeing memory because you specify the names of the XTI structures rather than information about their size. If you use `t_alloc` and `t_free` to manage the memory for XTI structures, and the structures change in future releases, you will not need to change your program.

With `t_error` you can print a user-supplied message (explanation) plus the contents of *`t_errno`* to standard output.

Finally, `t_look` is an important function for retrieving the current outstanding event associated with the endpoint. Typically, if an XTI function returns TLOOK as an error to indicate a significant asynchronous event has occurred, the transport user follows by issuing the `t_look` function to retrieve the event. For more information about events, see Section 3.2.3.

3.5 Porting to XTI

This section provides the following:

- Guidelines for writing programs to XTI
- Information about XTI and TLI compatibility
- Information about rewriting sockets applications to use XTI

3.5.1 Protocol Independence and Portability

XTI was designed to provide an interface that is independent of the specific transport protocol used. You can write applications that can modify their behavior according to any subset of the XTI functions and facilities supported by each of the underlying transport providers.

Providers do not have to provide all the features of all the XTI functions. Therefore, application programmers should follow these guidelines when writing XTI applications:

- Use only the functions that are commonly supported features of XTI.

  If your application uses features that are not provided by all transport providers, it may not be able to use them with some transport providers or some XTI implementations.

  For example, the orderly release facility (the `t_sndrel` and `t_rcvrel` functions) is not supported by all connection-based transport protocols; in particular it is not supported by ISO protocols. If your application runs in an environment with multiple protocols, make sure it does not use the orderly release facility.

As an alternative to using only the commonly supported features, write your application so that it modifies its behavior according to the subset of XTI functions supported by each transport provider.

- Do not assume that logical data boundaries are preserved across a connection.

  Some transport providers, such as TCP, do not support the concept of a TSDU, so they ignore the T_MORE flag when used with the `t_snd`, `t_sndudata`, `t_rcv`, and `t_rcvudata` functions.

- Do not exceed the protocol-specific service limits returned on the `t_open` and `t_getinfo` functions.

  Make sure your application retrieves these limits before transferring data and adheres to the limits throughout the communication process.

- Do not rely on options that are protocol-specific.

  Although the `t_optmgmt` function allows an application to access the default protocol options from the transport provider and pass them as an argument to the connection-establishment function, make sure your application avoids examining the options or relying on the existence of certain ones.

- Do not interpret the reason codes associated with the `t_rcvdis` function or the error code associated with the `t_rcvuderr` function.

  These codes depend on the underlying protocol so, to achieve protocol independence, make sure your application does not attempt to interpret the codes.

- Perform only XTI operations on the file descriptor returned by the `t_open` function.

  If you perform other operations, the results can vary from system to system.

The following sections explain how to port applications from different transport-level programming interfaces to XTI. Specifically, they discuss how to port from the two most common transport-level programming interfaces: Transport Layer Interface (TLI), which many UNIX System V applications use, and the 4.3BSD socket interface, which many Berkeley UNIX applications use.

The information presented in the following sections presumes that you are experienced at programming with TLI or sockets and that you understand fundamental XTI concepts and syntax.

3.5.2 XTI and TLI Compatibility

This section discusses issues to consider before you recompile your TLI programs and explains how to recompile them. As a long-term solution, you should use the XTI interface instead of the TLI interface. As more applications and transport providers use XTI, you might find it advantageous to do so as well.

XTI and TLI support the same functions, states, and modes of service. Shared library support is the default when you link an XTI or TLI application with the XTI or TLI library. For more information on shared library support, see Section 3.2.2.

Before you recompile your TLI program, you should consider your program's current implementation of the following event management: The System V UNIX operating system provides the `poll` function as a tool for managing events. The Tru64 UNIX implementation of XTI supports the `poll` function, so if your application uses it, you can recompile. If your program uses a unique mechanism for managing events, you should port that mechanism to Tru64 UNIX or change to the polling mechanism provided with Tru64 UNIX.

Because the Tru64 UNIX implementation of TLI is compatible at the source level with AT&T TLI, you can recompile your TLI program with the Tru64 UNIX TLI library using the following steps:

1. Make sure the TLI header file is included in your source code:

   ```
   #include <tli/tiuser.h>
   ```

2. Recompile your application using the following command syntax:

   ```
   cc -o name name.c –ltli
   ```

If you decide to change your TLI application to an XTI application, be aware of the following minor differences between TLI and XTI.

- In XTI, `t_error` is a function of type `int` that returns an integer value (0 for success and -1 for failure), while in TLI, it is a procedure of type `void`.
- In XTI, `t_look` does not support the T_ERROR event (as in TLI); it returns -1 and the `t_errno` instead.
- For the *oflag* parameter of the `t_open` function, the O_NDELAY value in TLI is known as the O_NONBLOCK value in XTI.
- XTI opens an endpoint with read-write access because most of its functions require read-write access to transport providers. TLI opens with read-only, write-only, or read-write access. Specifically, in the `t_open` function, XTI uses the bitwise inclusive OR of O_RDWR and O_NONBLOCK as the value of the *oflag* parameter; TLI uses the bitwise inclusive OR of O_NDELAY and either O_RDONLY, O_WRONLY,

or O_RDWR. The O_RDONLY and O_WRONLY values are not available in XTI; O_RDWR is the only valid value for access to an endpoint.

- TLI assumes the transport provider has an automatic address generator; XTI does not. If the transport provider does not have an automatic address generator, XTI can return the proper error message if conflicting requests are issued.
- XTI defines protocol-specific information for the TCP/IP and OSI protocols. The Tru64 UNIX XTI implementation adds support for protocol-specific options for STREAMS-based protocols; TLI does not provide such information.
- XTI provides additional events to manage flow control, such as T_GODATA and T_GOEXDATA; in TLI, you keep sending until successful.
- XTI provides additional error messages to convey more precise error information to applications. All functions that change the state of an endpoint use the TOUTSTATE error to indicate the function was called when the endpoint was in the wrong state. Some XTI functions return the TLOOK error to indicate that an urgent asynchronous event occurred. With TLI, you must call the `t_look` function explicitly before the function or set a signal for the TLOOK event, which are less convenient. The TBADQLEN error, returned when there are no queued connection requests, prevents an application from waiting forever after issuing the `t_listen` function. See the XTI reference pages for more information on error messages.

To make a TLI application a true XTI application, do the following:

1. Include the XTI header file instead of the TLI header file in your source code:

   ```
   #include <xti.h>
   ```

2. Make any changes or extensions to your program resulting from the differences between TLI and XTI.
3. Recompile your application using the following command syntax:

   ```
   cc -o name name.c –lxti
   ```

3.5.3 Rewriting a Socket Application to Use XTI

This section explains the differences between the socket interface and XTI. It assumes that your applications use the standard 4.3BSD socket interface and does not account for any extensions or changes you have made to the socket interface. See Appendix B for examples of both sockets and XTI servers and clients.

XTI shares many common functions with the socket interface. However, you should be aware of any differences between it and your current socket interface when rewriting an application for use with XTI.

XTI provides 30 functions. Of the 14 socket functions that map onto corresponding XTI functions, 6 have subtle differences. Table 3–14 lists each XTI function, its corresponding socket function (if one exists), and whether the two functions share common semantics. Generally, socket calls pass parameters by value, while most XTI functions pass pointers to structures containing a combination of input and output parameters.

Table 3–14: Comparison of XTI and Socket Functions

XTI Function	Socket Function	Shared Semantics
t_accept	accept	No
t_alloc	—	—
t_bind	bind	No
t_close	close	Yes
t_connect	connect	Yes
t_error	—	—
t_free	—	—
t_getinfo	—	—
t_getstate	—	—
t_listen	listen, accept	Yes [a]
t_look	select	No
t_open	socket	Yes
t_optmgmt	setsockopt, getsockopt	No
t_rcv	recv	Yes
t_rcvconnect	—	—
t_rcvdis	—	—
t_rcvrel	—	—
t_rcvreldata	—	—
t_rcvudata	recvfrom	Yes
t_rcvuderr	—	—
t_rcvv	recv	Yes
t_rcvvudata	recv	Yes

Table 3–14: Comparison of XTI and Socket Functions (cont.)

XTI Function	Socket Function	Shared Semantics
t_snd	send	Yes
t_snddis	shutdown	No
t_sndrel	—	—
t_sndreldata	—	—
t_sndudata	sendto	Yes
t_sndv	send	Yes
t_sndvudata	send	Yes
t_sync	—	—
t_sysconf	—	—
t_unbind	—	—

[a] In XTI, the t_listen function specifies the queue length parameter as well as waiting for the incoming connection. In sockets, the listen function only specifies the queue length parameter.

The XTI functions that do not share all semantics with their socket counterparts have the following differences:

t_accept

The t_accept function takes the user-specified *resfd* argument and establishes a connection with the remote endpoint. In contrast, the accept call from sockets asks the system to select the file descriptor to which the connection will be established. Additionally, the t_accept function is issued after a connection indication is received; therefore, it does not block. Conversely, the accept call is issued in anticipation of a connect request and therefore may block until the connect request occurs.

t_bind

XTI can bind one protocol address to many endpoints, while the socket interface permits one address to be bound with only one socket.

t_look

The t_look function returns the current event, which can be one of nine possible events: T_LISTEN, T_CONNECT, T_DATA, T_EXDATA, T_DISCONNECT, T_UDERR, T_ORDREL, T_GODATA, T_GOEXDATA. The poll function can be used to monitor incoming events on a transport endpoint. The select call can be used to see if a single descriptor is ready for read or write, or if an exceptional condition is pending.

t_snddis

The t_snddis function initiates an abortive release on an established connection or rejects a connection request. After an XTI program issues the t_snddis functions it can continue to listen for requests with the t_listen function or reestablish a connection with the t_connect function. In sockets, once you shut down a connection with the shutdown and close calls, the system automatically frees all local resources that are allocated for this connection. Therefore, in order to continue to listen for connections or establish a connection, the program needs to reissue the socket and bind calls.

XTI and sockets both use a series of states to control the appropriate sequence of calls, but each uses a different set of states. XTI states and socket states do not share similar semantics. For example, XTI states are mutually exclusive; socket states are not.

Few error messages are common among sockets and XTI. Table 3–15 lists the socket error messages that have comparable XTI error messages.

Table 3–15: Comparison of Socket and XTI Messages

Socket Error	XTI Error	Description
EBADF	TBADF	You specified an invalid file descriptor.
EOPNOTSUPP	TNOTSUPPORT	You issued a function the underlying transport provider does not support.
EADDRINUSE	TADDRBUSY	You specified an address that is already in use.
EACCES	TACCES	You do not have permission to use the specified address.

Note

XTI and TLI are implemented using STREAMS. You should use the poll function instead of the select call on any STREAMS file descriptors.

3.6 Differences Among XPG3, XNS4.0, and XNS5.0

This section provides information on the differences among the XPG3, XNS4.0, and XNS5.0 implementation of XTI.

In earlier versions of Tru64 UNIX, the XTI implementation conformed to X/Open's XPG3 specification. The current implementation conforms to X/Open's XNS4.0 and XNS5.0 specifications for XTI; XNS4.0 is the default.

There are some changes in the specification of which you, as a programmer, should be aware. This section outlines these differences and the related programming issues.

Note that the implementation of Tru64 UNIX converges XPG3, XNS4.0, and XNS5.0 versions of XTI in a single subset. This section also provides details about the usage of the appropriate level of functionality.

In this manual, the terms XNS4.0 or XNS4.0 XTI are used to refer to the default implementation of XTI available in this version of Tru64 UNIX. The terms XNS5.0 or XNS5.0 XTI are used to refer to the implementation of XTI that confirms to X/Open's XNS5.0 specification. The terms XPG3 XTI refer to the implementation of XTI that conforms to X/Open's XPG3 specification. Note that the latter can be available in the current versions of Tru64 UNIX due to binary compatibility or source migration features.

3.6.1 Major Differences Between XPG3 and XNS4.0

Most of the changes between the two specifications are upwardly compatible, with the exception of the `t_optmgmt` function.

The following is a quick summary of the basic changes in the XTI from XPG3 to XNS4.0:

- Optional functions were made mandatory. This does not affect the Tru64 UNIX implementation of XTI, because Tru64 UNIX implemented all the optional functions in its XPG3 version of XTI.
- Many aspects of the XPG3 specification were clarified, which makes XTI applications more portable.
- Some new error codes were added, ensuring better programmatic behavior.
- Options and management structures were revised to provide more control over various aspects of communications.

The changes to the `t_optmgmt` function are extensive and incompatible with the XPG3 specification. In general, an application that uses the XPG3 implementation of the `t_optmgmt` function cannot use the `t_optmgmt` function on a system running the XNS4.0 or XNS5.0 specification, without making some modifications to the source.

3.6.2 Major Differences Between XNS4.0 and XNS5.0

Most of the changes between the two specifications are upwardly compatible. The following is a quick summary of the basic changes in the XTI from XNS4.0 to XNS5.0:

- Seven new functions were added: `t_rcvreldata`, `t_rcvv`, `t_rcvvudata`, `t_sndreldata`, `t_sndv`, `t_sndvudata`, and `t_sysconf`.

3.6.3 Source Code Migration

If you have an application that was developed for XPG3 XTI, you have the following choices to support it under the current version of the operating system:

- Use the older binaries of the application; see Section 3.6.4.
- Recompile the unaltered sources.
- Make changes to the sources to comply with XNS4.0 XTI, if wanted.
- Make changes to the sources to comply with XNS5.0 XTI, if wanted.

Which option you choose will depend on your situation. The following sections describe these conditions in detail.

3.6.3.1 Use the Older Binaries of your Application

This choice is appropriate if the sources and features of your application are not going to change. It is useful to provide continued coverage by ensuring that older releases of your products are still functional.

3.6.3.2 Unaltered Sources

This situation arises from minor changes due to correcting minor problems. Therefore, there are no changes to the structure or features or the application. In this case, you might want to compile the sources in the same manner as XPG3 development environment. In that case, compile your source code with the `–DXPG3` compiler switch. This ensures that the headers automatically define the older features for you.

3.6.3.3 XNS4.0 Compliant Application

If you need to use the new features supported by XNS4.0 XTI, you will have to make changes in your source code. You cannot combine the features from the XPG3 and XNS4.0 XTI. Therefore, if you have large applications consisting of multiple files, you will need to recompile all files with the new features, rather than just the few you might have changed.

You need to compile your source code with the –DXOPEN_SOURCE compiler switch. Additionally, you must ensure that the names of the transport protocols (as provided through the streams device special files as in /dev/streams/xtiso/tcp) are updated to reflect the naming convention used in XNS4.0 XTI. For example, the names for TCP and UDP are /dev/streams/xtiso/tcp+ and /dev/streams/xtiso/udp+. Check the reference manual for the names for the other protocols.

3.6.3.4 XNS5.0 Compliant Application

If you need to use the new features supported by XNS5.0 XTI, you will have to make changes in your source code. You cannot combine the features from the XPG3 and XNS4.0 XTI. Therefore, if you have large applications consisting of multiple files, you will need to recompile all files with the new features, rather than just the few you might have changed.

You need to compile your source code with the –DXOPEN_SOURCE=500 compiler switch. Additionally, you must ensure that the names of the transport protocols (as provided through the streams device special files as in /dev/streams/xtiso/tcp) are updated to reflect the naming convention used in XNS5.0 XTI. For example, the names for TCP and UDP are /dev/streams/xtiso/tcp5 and /dev/streams/xtiso/udp5. Check the reference manual for the names for the other protocols.

3.6.4 Binary Compatibility

There are certain conditions of which you should be aware when running application binaries developed with XPG3 XTI.

Under unusual circumstances, the errors in XPG3 programs may have been masked due to the way in which the programs or libraries were compiled and linked. It is feasible that the new implementation is able to flag such conditions as errors. Since the error manifested is a programming error in the application, you will have to correct it. The common programming errors that may cause these errors are pointer overruns and uninitialized variables.

Another issue to consider is the availability of XNS4.0 features through STREAMS special files. This is significant if your application accepts command line input for the specifying transport protocol or imports the protocol names from some configuration files. Since the system configured with XTI will have the file names for XNS4.0-compliant protocols as well, it is important to warn users and administrators that those special names should not be used with applications running with binary-compatibility mode. The results of such an action are undefined.

If you are planning to run an old applications without recompiling them, check them for binary compatibility to avoid these problems.

3.6.5 Packaging

Systems running the current version of the operating system and configured to run XTI support XPG3, XNS4.0, and XNS5.0-compliant functionality. You cannot run the XPG3, XNS4.0, and XNS5.0 functionality separately. Therefore, you only need to ensure that XTI subsystem is configured.

3.6.6 Interoperability

You can use the XPG3, XNS4.0, and XNS5.0 versions of XTI on the same network. If you are using compatible versions of your application, then the operation should be transparent to users.

It is possible to convert your application in simple steps, so that you have some pieces that are XPG3 XTI compatible, some pieces that are XNS4.0 compatible, and some pieces that are XNS5.0 compatible. The only thing you have to ensure is that application-level protocol remains the same. Apart from that there will be no issue for interoperability of these components. Therefore, if you have client and server components of an application, you can choose to upgrade the server component for XNS4.0 or XNS5.0 compliance, while the client component is still operational in binary compatibility mode. Later, once the server functionality is updated satisfactorily, you can choose to update the client software.

3.6.7 Using XTI Options

This section provides information on using XTI options in XNS4.0, XNS5.0, and XPG3.

3.6.7.1 Using XTI Options in XNS4.0 and XNS5.0

This section provides the following information on using XTI options:

- General information on using options
- Format of options
- Elements of negotiation
- Option management of transport endpoint

General Information

The following functions contain an *opt* argument of the type `struct netbuf` as an input or output parameter. This argument is used to convey options between the transport user and the transport provider:

- `t_accept`
- `t_connect`

- `t_listen`
- `t_optmgmt`
- `t_rcvconnect`
- `t_rcvudata`
- `t_rcvuderr`
- `t_sndudata`

There is no general definition about the possible contents of options. There are general XTI options and those that are specific for each transport provider. Some options allow you to tailor your communication needs; for instance, by asking for high throughput or low delay. Others allow the fine-tuning of the protocol behavior so that communication with unusual characteristics can be handled more effectively. Other options are for debugging purposes.

All options have default values. Their values have meaning to and are defined by the protocol level in which they apply. However, their values can be negotiated by a transport user. This includes the simple case where the transport user can enforce its use. Often, the transport provider or even the remote transport user can have the right to negotiate a value of lesser quality than the proposed one, that is, a delay can become longer, or a throughput can become lower.

It is useful to differentiate between options that are association-related and those that are not. (Association-related means a pair of communication transport users.) Association-related options are intimately related to the particular transport connection or datagram transmission. If the calling user specifies such an option, some ancillary information is transferred across the network in most cases. The interpretation and further processing of this information is protocol-dependent. For instance, in an ISO connection-oriented communication, the calling user can specify quality-of-service parameters on connection establishment. These are first processed and possibly lowered by the local transport provider, then sent to the remote transport provider that may degrade them again, and finally conveyed to the called user who makes the final selection and transmits the selected values back to the caller.

Options that are not association-related do not contain information destined for the remote transport user. Some have purely local relevance; for example, an option that enables debugging. Others influence the transmission; for instance, the option that sets the IP *`time-to-live`* field or TCP_NODELAY. (See `xti_internet`(7).) Local options are negotiated solely between the transport user and the local transport provider. The distinction between these two categories of options is visible in XTI through the following relationship: on output, the `t_listen` and `t_rcvudata` functions return association-related options only. The t_rcvconnect and t_rcvuderr functions may return options of both categories. On input, options of both categories may be specified with the t_accept and t_sndudata functions. The t_connect and t_optmgmt functions can process and return both categories of options.

The transport provider has a default value for each option it supports. These defaults are sufficient for the majority of communication relations. Therefore, a transport user should only request options actually needed to perform the task and leave all others at their default value.

This section describes the general framework for the use of options. This framework is obligatory for transport providers. The t_optmgmt reference page provides information on general XTI options. The xti_internet reference page provides information on the specific options that are legal with the TCP and UDP transport providers.

Format of Options

Options are conveyed through an *opt* argument of struct netbuf. Each option in the buffer specified is of the form struct t_opthdr possibly followed by an option value.

A transport provider embodies a stack of protocols. The *level* field of struct t_opthdr identifies the XTI level or a protocol of the transport provider as TCP or ISO 8073:1986. The *name* field identifies the option within the level and the *len* field contains the total length; that is, the length of the option header t_ophdr plus the length of the option value. The *status* field is used by the XTI level or the transport provider to indicate success or failure of a negotiation.

Several options can be concatenated; however, The transport user has to ensure that each option starts at a long-word boundary. The macro OPT_NEXTHDR*(pbuf,buflen,poptons)* for XNS4.0 and T_OPT_NEXTHDR*(pbuf,buflen,poptons)* for XNS5.0 can be used for that purpose. The parameter *pbuf* denotes a pointer to an option buffer *opt.buf* and *buflen* is its length. The parameter *poption* points to the current options in the option buffer. OPT_NEXTHDR and T_OPT_NEXTHDR return a pointer to the position of the next option or return a null pointer if the option buffer is exhausted. These macros are helpful for writing and reading the option list.

Elements of Negotiation

This section describes the general rules governing the passing and retrieving of options and the error conditions that can occur. Unless explicitly restricted, these rules apply to all functions that allow the exchange of options.

Multiple Options and Options Levels

When multiple options are specified in an option buffer on input, different rules apply to the levels that may be specified, depending on the function call. Multiple options specified on input to `t_optmgmt` must address the same option level. Options specified on input to `t_connect`, `t_accept`, and `t_sndudata` can address different levels.

Illegal Options

Only legal options can be negotiated; illegal options can cause failure. An option is illegal if one of the following applies:

- The length specified in the *`t_opthdr.len`* parameter exceeds the remaining size of the option buffer (counted from the beginning of the option).
- The option value is illegal. The legal values are defined for each option. See `t_optmgmt`(3) and `xti_internet`(7).

If an illegal option is passed to XTI, the following will happen:

- A call to the `t_optmgmt` function fails with a TBADOPT error.
- The `t_accept` or `t_connect` functions fail with a TBADOPT error or the connection establishment aborts, depending on the implementation and the time the illegal option is detected. If the connection aborts, a T_DISCONNECT event occurs and a synchronous call to `t_connect` fails with a TLOOK error. It depends on timing and implementation conditions whether a `t_accept` function will succeed or fail with a TLOOK error in that case.
- A call to the `t_sndudata` function either fails with a TBADOPT error or it successfully returns; but a T_UDERR event occurs to indicate that the datagram was not sent.

If the transport user passes multiple options in one call and one of them is illegal, the call fails as described previously. It is, however, possible that some or even all of the submitted legal options were successfully negotiated. The transport user can check the current status by a call to the `t_optmgmt` function with the T_CURRENT flag set. See `t_optmgmt`(3) and `xti_internet`(7).

Specifying an option level unknown to or not supported by the protocol selected by the option level does not cause failure. The option is discarded in calls to the `t_connect`, `t_accept`, or `t_sndudata` functions. The `t_opmgmt` function returns T_NOTSULPORT in the *`level`* field of the option.

Initiating an Option Negotiation

A transport user initiates an option negotiation when calling the `t_connect`, `t_sndudata`, or `t_optmgmt` functions with the T_NEGOTIATE flag set.

The negotiation rules for these functions depend on whether an option request is an absolute requirement. This is explicitly defined for each option. See `t_optmgmt`(3) and `xti_internet`(7). In the case of an ISO transport provider, for example, the option that requests use of expedited data is not an absolute requirement. On the other hand, the option that requests protection could be an absolute requirement.

Note

The term **absolute requirement** originates from the quality-of-service parameters in the ISO 8072:1986 specification. Its use is extended here to all options.

If the proposed option value is an absolute requirement, there are three possible outcomes:

- The negotiated value is the same as the proposed one. When the result of the negotiation is retrieved, the *status* field in `t_opthdr` is set to T_SUCCESS.
- The negotiation is rejected if the option is supported but the proposed value cannot be negotiated. This leads to the following:
  - The `t_optmgmt` function successfully returns; but the returned option has its *status* field set to T_FAILURE.
  - Any attempt to establish a connection aborts; a T_DISCONNECT event occurs and a synchronous call to the `t_connect` function fails with a TLOOK error.
  - The `t_sndudata` function fails with a TLOOK error or successfully returns; but a T_UDERR event occurs to indicate that the datagram was not sent.

  If multiple options are submitted in one call and one of them is rejected, XTI behaves as just described. Although the connection establishment or the datagram transmission fails, options successfully negotiated before some option was rejected retain their negotiated values. There is no roll-back mechanism. See the Option Management of a Transport Endpoint section for more information.

  The `t_optmgmt` function attempts to negotiate each option. The *status* fields of the returned options indicate success (T_SUCCESS) or failure (T_FAILURE).

- If the local transport provider does not support the option at all, the `t_optmgmt` function reports T_NOTSULPORT in the *status* field. The `t_connect` and `t_sndudata` functions ignore this option.

If the proposed option value is not an absolute requirement, the following outcomes are possible:

- The negotiated value is of equal or lesser quality than the proposed one; for example, a delay may become longer.

  When the result of the negotiation is retrieved, the *status* field in `t_opthdr` is set to T_SUCCESS if the negotiated value equals the proposed one; otherwise, it is set to T_PARTSUCCESS.

- If the local transport provider does not support the option at all, `t_optmgmt` reports T_NOTSULPORT in the *status* field. The `t_connect` and `t_sndudata` functions ignore this option.

Unsupported options do not cause functions to fail or a connection to abort, since different vendors possibly implement different subsets of options. Furthermore, future enhancements of XTI might encompass additional options that are unknown to earlier implementations of transport providers. The decision whether or not the missing support of an option is acceptable for the communication is left to the transport user.

The transport provider does not check for multiple occurrences of the same options, possibly with different option values. It simply processes the options in the option buffer sequentially. However, the user should not make any assumption about the order of processing.

Not all options are independent of one another. A requested option value might conflict with the value of another option that was specified in the same call or is currently effective. See the Option Management of a Transport Endpoint section for more information. These conflicts may not be detected at once, but they might later lead to unpredictable results. If detected at negotiation time, these conflicts are resolved within the rules stated above. The outcomes may thus be quite different and depend on whether absolute or nonabsolute requests are involved in the conflict.

Conflicts are usually detected at the time a connection is established or a datagram is sent. If options are negotiated with the `t_optmgmt` function, conflicts are usually not detected at this time, since independent processing of the requested options must allow for temporal inconsistencies.

When called, the `t_connect`, and `t_sndudata` functions initiate a negotiation of all association-related options according to the rules of this section. Options not explicitly specified in the function calls themselves are taken from an internal option buffer that contains the values of a previous negotiation. See the Option Management of a Transport Endpoint section for more information.

Responding to a Negotiation Proposal

In connection-oriented communication, some protocols give the peer transport users the opportunity to negotiate characteristics of the transport connection to be established. These characteristics are association-related options. With the connect indication, the called user receives (through the `t_listen` function) a proposal about the option values that should be effective for this connection. The called user can accept this proposal or weaken it by choosing values of lower quality; for example, longer delays than proposed. The called user can, of course, refuse the connection establishment altogether.

The called user responds to a negotiation proposal using the `t_accept` function. If the called transport user tries to negotiate an option of higher quality than proposed, the outcome depends on the protocol to which that option applies. Some protocols may reject the option, some protocols take other appropriate action described in protocol-specific reference pages. If an option is rejected, the connection fails; a T_DISCONNECT event occurs. In that case, whether a `t_accept` function can still succeed or fail with a TLOOK error depends on timing and implementation conditions.

If multiple options are submitted with the `t_accept` function and one of them is rejected, the connection fails as described previously. Options that could be successfully negotiated before the erroneous option was processed retain their negotiated value. There is no rollback mechanism. See the Option Management of a Transport Endpoint section for more information.

The response options can either be specified with the `t_accept` call or can be preset for the responding endpoint (not the listening endpoint) *`resfd`* in a `t_optmgmt` call (action T_NEGOTIATE) prior to the `t_accept` call. (See the Option Management of a Transport Endpoint section for more information.) Note that the response to a negotiation proposal is activated when the `t_accept` function is called. A `t_optmgmt` function call with erroneous option values as described previously will succeed; the connection aborts at the time the `t_accept` function is called.

The connection also fails if the selected option values lead to contradictions.

The `t_accept` function does not check for multiple specification of an option. (See the Initiating an Option Negotiation section.) Unsupported options are ignored.

Retrieving Information About Options

This section describes how a transport user can retrieve information about options.

A transport user must be able to:

- Know the result of a negotiation; for example, at the end of a connection establishment.
- Know the proposed option values under negotiation during connection establishment.
- Retrieve option values sent by the remote transport user for notification only; for example, IP options.
- Check option values currently in effect for the transport endpoint.

To this end, the following functions take an output argument *opt* of the `struct netbuf`:

- `t_connect`
- `t_listen`
- `t_optmgmt`
- `t_rcvconnect`
- `t_rcvudata`
- `t_rcvuderr`

The transport user has to supply a buffer to which the options will be written; the *opt.buf* parameter must point to this buffer and the *opt.maxlen* parameter must contain the buffer's size. The transport user can set the *opt.maxlen* parameter to zero to indicate that no options are to be retrieved.

Which options are returned depend on the function call involved:

- `t_connect` in synchronous mode and `t_rcvconnect`

  The functions return the values of all association-related options that were received with the connection response and the negotiated values of those nonassociation-related options that had been specified on input. However, options specified on input in the `t_connect` call that are not supported or refer to an unknown option level are discarded and not returned on output.

  The *status* field of each option returned with the `t_connect` or `t_rcvconnect` function indicates if the proposed value (T_SUCCESS) or a degraded value (T_PARTSUCCESS) has been negotiated. The *status* field of received ancillary information (for example, IP options) that is not subject to negotiation is always set to T_SUCCESS.

- `t_listen`

  The received association-related options are related to the incoming connection (identified by the sequence number), not to the listening endpoint. (However, the option values currently in effect for the listening endpoint can affect the values retrieved by the `t_listen` function, since the transport provider might also be involved in the negotiation process.)

Therefore, if the same options are specified in a call to the t_optmgmt function with action T_CURRENT, they will usually not return the same values.

The number of received options may vary for subsequent connect indications, since many association-related options are only transmitted on explicit demand by the calling user; for example, IP options or ISO 8072:1986 throughput. It is even possible that no options at all are returned.

The *status* field is irrelevant.

- t_rcvudata

  The received association-related options are related to the incoming datagram, not to the transport endpoint *fd*. Therefore, if the same options are specified in a call to the t_optmgmt function with action T_CURRENT, the t_optmgmt function will usually not return the same values.

  The number of options received may vary from call to call.

  The *status* field is irrelevant.

- t_rcvuderr

  The returned options are related to the options input of the previous t_sndudata call that produced the error. Which options are returned and which values they have depend on the specific error condition. The *status* field is irrelevant.

- t_optmgmt

  This call can process and return both categories of options. It acts on options related to the specified transport endpoint, not on options related to a connect indication or an incoming datagram. For more information, see t_optmgmt(3).

Privileged and Read-Only Options

Only privileged users can request privileged options, or option values. The meaning of privilege is hereby implementation-defined.

Read-only options serve for information purposes only. The transport user may be allowed to read the option value but not to change it. For instance, to select the value of a protocol timer or the maximum length of a protocol data unit may be too subtle to leave to the transport user, though the knowledge about this value might be of some interest. An option might be read-only for all users or solely for nonprivileged users. A privileged option might be inaccessible or read-only for nonprivileged users.

An option might be negotiable in some XTI states and read-only in other XTI states. For instance, the ISO quality-of-service options are negotiable in the T_IDLE and T_INCON states, and read-only in all other states (except T_UNINIT).

If a transport user requests negotiation of a read-only option, or a nonprivileged user requests illegal access to a privileged option, the following outcomes are possible:

- The `t_optmgmt` function successfully returns, but the returned option has its *`status`* field set to T_NOTSULPORT if a privileged option was requested illegally, and to T_READONLY if modification of a read-only option was requested.
- If negotiation of a read-only option is requested, the `t_accept` or `t_connect` functions fail with TACCES or the connection establishment aborts and a T_DISCONNECT event occurs. If the connection aborts, a synchronous call to `t_connect` fails with TLOOK. If a privileged option is illegally requested, the option is quietly ignored. A nonprivileged user is not able to select an option that is privileged or unsupported. Timing and implementation conditions determine whether a `t_accept` call succeeds or fails with TLOOK.
- If negotiation of a read-only option is requested, the `t_sndudata` function may return TLOOK or successfully return, but a T_UDERR event occurs to indicate that the datagram was not sent. If a privileged option is illegally requested, the option is quietly ignored. A nonprivileged user cannot select an option that is privileged or unsupported.

If multiple options are submitted to the `t_connect`, `t_accept`, or `t_sndudata` functions and a read-only option is rejected, the connection or the datagram transmission fails as described. Options that could be successfully negotiated before the erroneous option was processed retain their negotiated values. There is no rollback mechanism. See the Option Management of a Transport Endpoint section for more information.

Option Management of a Transport Endpoint

This section describes how option management works during the lifetime of a transport endpoint.

Each transport endpoint is (logically) associated with an internal option buffer. When a transport endpoint is created, this buffer is filled with a system default value for each supported option. Depending on the option, the default may be OPTION ENABLED, OPTION DISABLED, or denote a time span, and so on. These default settings are appropriate for most uses. Whenever an option value is modified in the course of an option negotiation, the modified value is written to this buffer and overwrites the previous one. At any time, the buffer contains all option values that are currently effective for this transport endpoint.

The current value of an option can be retrieved at any time by calling the t_optmgmt function with the T_CURRENT flag set. Calling the t_optmgmt function with the T_DEFAULT flag set yields the system default for the specified option.

A transport user can negotiate new option values by calling the t_optmgmt function with the T_NEGOTIATE flag set. The negotiation follows the rules described in the Elements of Negotiation section.

Some options may be modified only in specific XTI states and are read-only in other XTI states. Many association-related options, for instance, may not be changed in the T_DATAXFER state, and an attempt to do so fails; see the Privileged and Read-Only Options section. The legal states for each option are specified with its definition.

As usual, association-related options take effect at the time a connection is established or a datagram is transmitted. This is the case if they contain information that is transmitted across the network or determine specific transmission characteristics. If such an option is modified by a call to the t_optmgmt function, the transport provider checks whether the option is supported and negotiates a value according to its current knowledge. This value is written to the internal option buffer.

The final negotiation takes place if the connection is established or the datagram is transmitted. This can result in a degradation of the option value or even in a negotiation failure. The negotiated values are written to the internal option buffer.

Some options can be changed in the T_DATAXFER state; for example, those specifying buffer sizes. Such changes might affect the transmission characteristics and lead to unexpected side effects; for example, data loss if a buffer size was shortened.

The transport user can explicitly specify both categories of options on input when calling the t_connect, t_accept, or t_sndudata functions. The options are at first locally negotiated option by option and the resulting values written to the internal option buffer. The modified option buffer is then used if a further negotiation step across the network is required; for example, in connection-oriented ISO communication. The newly negotiated values are then written to the internal option buffer.

At any stage, a negotiation failure can cause the transmission to abort. If a transmission aborts, the option buffer preserves the content it had at the time the failure occurred. Options that could be negotiated before the error occurred are written back to the option buffer, whether the XTI call fails or succeeds.

It is up to the transport user to decide which option it explicitly specifies on input when calling the t_connect, t_accept, or t_sndudata functions.

The transport user need not pass options at all by setting the *len* field of the function's input *opt* argument to zero (0). The current content of the internal option buffer is then used for negotiation without prior modification.

The negotiation procedure for options at the time of a `t_connect`, `t_accept`, or `t_sndudata` call always obeys the rules in the Initiating an Option Negotiation section whether the options were explicitly specified during the call or implicitly taken from the internal option buffer.

The transport user should not make assumptions about the order in which options are processed during negotiation.

A value in the option buffer is only modified as a result of a successful negotiation of this option. It is, in particular, not changed by a connection release. There is no history mechanism that would restore the buffer state existing prior to the connection establishment of the datagram transmission. The transport user must be aware that a connection establishment or a datagram transmission may change the internal option buffer, even if each option was originally initialized to its default value.

The Option Value T_UNSPEC

Some options may not always have a fully specified value. An ISO transport provider, for instance, that supports several protocol classes might not have a preselected preferred class before a connection establishment is initiated. At the time of the connection request, the transport provider may conclude from the destination address, quality-of-service parameters, and other locally available information which preferred class it should use. A transport user asking for the default value of the preferred class option in the T_IDLE state would get the value T_UNSPEC. This value indicates that the transport provider did not yet select a value. The transport user could negotiate another value as the preferred class; for example, T_CLASS2. The transport provider would then be forced to initiate a connect request with class 2 as the preferred class.

An XTI implementation may also return the T_UNSPEC value if it currently cannot access the option value. This can happen in the T_UNBND state in systems where the protocol stacks reside on separate controller cards and not in the host. The implementation may never return T_UNSPEC if the option is not supported at all.

If T_UNSPEC is a legal value for a specific option, it can be used on input, as well. It is used to indicate that it is left to the provider to choose an appropriate value. This is especially useful in complex options as ISO throughput, where the option value has an internal structure. The transport user can leave some fields unspecified by selecting this value. If the user proposes T_UNSPEC, the transport provider is free to select an appropriate value. This might be the default value, some other explicit value, or T_UNSPEC.

For each option, it is specified whether T_UNSPEC is a legal value for negotiation purposes.

The info Argument

The `t_open` and `t_getinfo` functions return values representing characteristics of the transport provider in the *info* argument. The value of *info->options* is used by the `t_alloc` function to allocate storage for an option buffer to be used in an XTI call. The value is sufficient for all uses.

In general, *info->options* also includes the size of privileged options; even if these are not read-only for nonprivileged users. Alternatively, an implementation can choose to return different values in *info->options* for privileged and nonprivileged users.

The values in *info->etsdu*, *info->connect*, and *info->discon* possibly diminish as soon as the T_DATAXFER state is entered. Calling the `t_optmgmt` function does not influence these values. For more information, see `t_optmgmt`(3).

Portability Issues

An application programmer who writes XTI programs has the following portability issues across the following:

- Protocol profiles
- Different system platforms

Options are intrinsically coupled with a definite protocol or protocol profile. Therefore, explicit use of options degrades portability across protocol profiles.

Different vendors might offer transport providers different option support. This is due to different implementation and product policies. The list of options on the `t_optmgmt`(3) reference page and in the protocol-specific reference pages are maximal sets, but do not necessarily reflect common implementation practice. Vendors implement subsets that suit their needs. Therefore, making careless use of options endangers portability across different system platforms.

Every implementation of a protocol profile accessible by XTI can be used with the default values of options. Applications can thus be written that do not care about options at all.

An application program that processes options retrieved from an XTI function should discard options it does not know to lessen its dependence from different system platforms and future XTI releases with possibly increased option support.

3.6.7.2 Negotiating Protocol Options in XPG3

The Tru64 UNIX XPG3 implementation of XTI provides an optional function, t_optmgmt, for retrieving, verifying, and negotiating protocol options with transport providers. After you create an endpoint with t_open and bind an address to it, you can verify or negotiate options with the transport provider.

Note

Although other transport providers may support the t_optmgmt function, the TCP transport provider provided with this operating system does not. See the transport provider documentation for information about option management.

See t_optmgmt(3) for function syntax, parameters, and errors. See Section 3.7 for a general description of XTI errors.

3.7 XTI Errors

XTI returns library errors and system errors. When an XTI function encounters an error, it returns a value of -1, and can do one of the following:

- Check the external variable *t_errno* to get the specific error. (For multithreaded applications, *t_errno* is thread specific.)
- Call the t_error function to print the text of the message associated with the error stored in *t_errno.*
- Check the state of the transport endpoint with the t_getstate function. Some errors change the state of the endpoint.

Note

Since a successful call to an XTI function does not clear the contents of *t_errno,* check *t_errno* only after an error occurs.

The <xti.h> header file defines the *t_errno* variable as a macro as follows:

```
#define t_errno(*_t_errno())
```

For more information on errors, see the individual XTI reference pages.

3.8 Configuring XTI Transport Providers

Use the `xtiso` kernel configuration option to configure XTI transport providers. You can configure the `xtiso` option into your system at installation time or you can add it to your system using the `doconfig` command. See the *Installation Guide*.

You can use the `doconfig` command in one of the following ways:

- Use the `doconfig` command without options if you have not customized your kernel. Without options the `doconfig` command creates a new kernel configuration file for your system.
- Use the `doconfig −c` command if you have customized your kernel and you do not want to recustomize it. The `doconfig −c` command allows you to add information to the existing kernel configuration file.

To use the `doconfig` command without any options, do the following:

1. Enter the `/usr/sbin/doconfig` command at the superuser prompt (#).
2. Enter a name for the kernel configuration file. It should be the name of your system in uppercase letters, and will probably be the default provided in square brackets ([]). For example:

   ```
   Enter a name for the kernel configuration file. [HOST1]: RETURN
   ```

3. Enter `y` when the system asks whether you want to replace the system configuration file. For example:

   ```
   A configuration file with the name 'HOST1' already exists.
   Do you want to replace it? (y/n) [n]: y

   Saving /sys/conf/HOST1 as /sys/conf/HOST1.bck

   *** KERNEL CONFIGURATION AND BUILD PROCEDURE ***
   ```

4. Select the `X/Open Transport Interface (XTISO, TIMOD, TIRDWR)` option from the Kernel Option Selection menu. Confirm your choice at the prompt. For example:

   ```
   *** KERNEL OPTION SELECTION ***

       Selection   Kernel Option
   --------------------------------------------------------------
          1       System V Devices
          2       NTP V3 Kernel Phase Lock Loop (NTP_TIME)
          3       Kernel Breakpoint Debugger (KDEBUG)
          4       Packetfilter driver (PACKETFILTER)
          5       Point-to-Point Protocol (PPP)
          6       STREAMS pckt module (PCKT)
          7       X/Open Transport Interface (XTISO, TIMOD, TIRDWR)
          8       File on File File System (FFM)
          9       ISO 9660 Compact Disc File System (CDFS)
          10      Audit Subsystem
          11      ACL Subsystem
          12      Logical Storage Manager (LSM)
   ```

```
        13      Advanced File System (ADVFS)
        14      All of the above
        15      None of the above
        16      Help
--------------------------------------------------------------

Enter the selection number for each kernel option you want.
For example, 1 3 [15]: 7

Enter the selection number for each kernel option you want.
For example, 1 3 : 7

You selected the following kernel options:

        X/Open Transport Interface (XTISO, TIMOD, TIRDWR)
Is that correct? (y/n) [y]: y

Configuration file complete.
```

5. Enter `n` when the `doconfig` command asks whether you want to edit the configuration file.

   The `doconfig` command then creates device special files, indicates where a log of the files it created is located, and builds the new kernel. After the new kernel is built, you must move it from the directory where `doconfig` places it to the root directory (/) and reboot your system.

   When you reboot, the `strsetup –i` command runs automatically, creating the device special files for any new STREAMS modules.

6. Enter the `strsetup –c` command to verify that the device is configured properly.

   The following example shows the output from the `strsetup –c` command:

```
# /usr/sbin/strsetup –c

STREAMS Configuration Information...Fri Nov  3 14:23:36 1995

           Name       Type   Major  Module ID
           ----       ----   -----  ---------
          clone                 32          0
            dlb     device      52       5010
          kinfo     device      53       5020
            log     device      54         44
           nuls     device      55       5001
           echo     device      56       5000
            sad     device      57         45
           pipe     device      58       5304
       xtisoUDP     device      59       5010
       xtisoTCP     device      60       5010
      xtisoUDP+     device      61       5010
      xtisoTCP+     device      62       5010
            ptm     device      63       7609
            pts     device       6       7608
            bba     device      64      24880
            lat     device       5          5
          pppif     module               6002
       pppasync     module               6000
        pppcomp     module               6001
       bufcall     module                0
           null     module              5002
           pass     module              5003
           errm     module              5003
           ptem     module              5003
          spass     module              5007
         rspass     module              5008
        pipemod     module              5303
          timod     module              5006
         tirdwr     module                 0
          ldtty     module              7701

        Configured devices = 15, modules = 14
```

To use the `doconfig –c` command to add the XTISO option to the kernel configuration file, do the following:

1. Enter the `doconfig –c` *HOSTNAME* command from the superuser prompt (#). *HOSTNAME* is the name of your system in uppercase letters. For example, for a system called `host1` you would enter:

   ```
   # doconfig –c HOST1
   ```

2. Add XTISO to the options section of the kernel configuration file.

   Enter `y` at the prompt to edit the kernel configuration file. The `doconfig` command allows you to edit the configuration file with the `ed` editor. For information about using the `ed` editor, see `ed`(1).

   The following `ed` editing session shows how to add the XTISO option to the kernel configuration file for `host1`. The number of the line after which you append the new line can differ between kernel configuration files:

   ```
   *** KERNEL CONFIGURATION AND BUILD PROCEDURE ***

   Saving /sys/conf/HOST1 as /sys/conf/HOST1.bck

   Do you want to edit the configuration file? (y/n) [n]: y

   Using ed to edit the configuration file.  Press return when ready,
   or type 'quit' to skip the editing session:
   2153

   48a
   options         XTISO
   .
   1,$w
   2185
   q



   *** PERFORMING KERNEL BUILD ***
   ```

3. After the new kernel is built you must move it from the directory where `doconfig` places it to the root directory (/) and reboot your system.

   When you reboot, the `strsetup –i` command is run automatically, creating the device special files for any new STREAMS modules.

4. Run the `strsetup –c` command to verify that the device is configured properly.

   The following example shows the output from the `strsetup –c` command:

```
# /usr/sbin/strsetup –c

STREAMS Configuration Information...Fri Nov  3 14:23:36 1995

           Name       Type   Major  Module ID
           ----       ----   -----  ---------
          clone                 32          0
            dlb     device      52       5010
          kinfo     device      53       5020
            log     device      54         44
           nuls     device      55       5001
           echo     device      56       5000
            sad     device      57         45
           pipe     device      58       5304
       xtisoUDP     device      59       5010
       xtisoTCP     device      60       5010
      xtisoUDP+     device      61       5010
      xtisoTCP+     device      62       5010
            ptm     device      63       7609
            pts     device       6       7608
            bba     device      64      24880
            lat     device       5          5
          pppif     module               6002
       pppasync     module               6000
        pppcomp     module               6001
       bufcall     module                  0
           null     module               5002
           pass     module               5003
           errm     module               5003
           ptem     module               5003
          spass     module               5007
         rspass     module               5008
        pipemod     module               5303
          timod     module               5006
         tirdwr     module                  0
          ldtty     module               7701

        Configured devices = 15, modules = 14
```

For detailed information on reconfiguring your kernel or the `doconfig` command see the *System Administration* manual.

4

Sockets

The operating system's sockets programming interface supports the XNS4.0 standard, POSIX 1003.1g Draft 6.6, and the Berkeley Software Distribution (BSD) socket programming interface. In addition, the operating system supports the basic sockets interface extensions for Internet Protocol Version 6 (IPv6) as defined in RFC 2553. The basic syntax of socket functions remains the same. Existing IPv4 applications will continue to operate as before, and IPv6 applications can interoperate with IPv4 applications.

In this operating system, sockets provide an interface to the Internet Protocol suite (TCP/IP) and to the UNIX domain for interprocess communication on the same system. However, you can use sockets to build network-based applications that are independent of the underlying networking protocols and hardware.

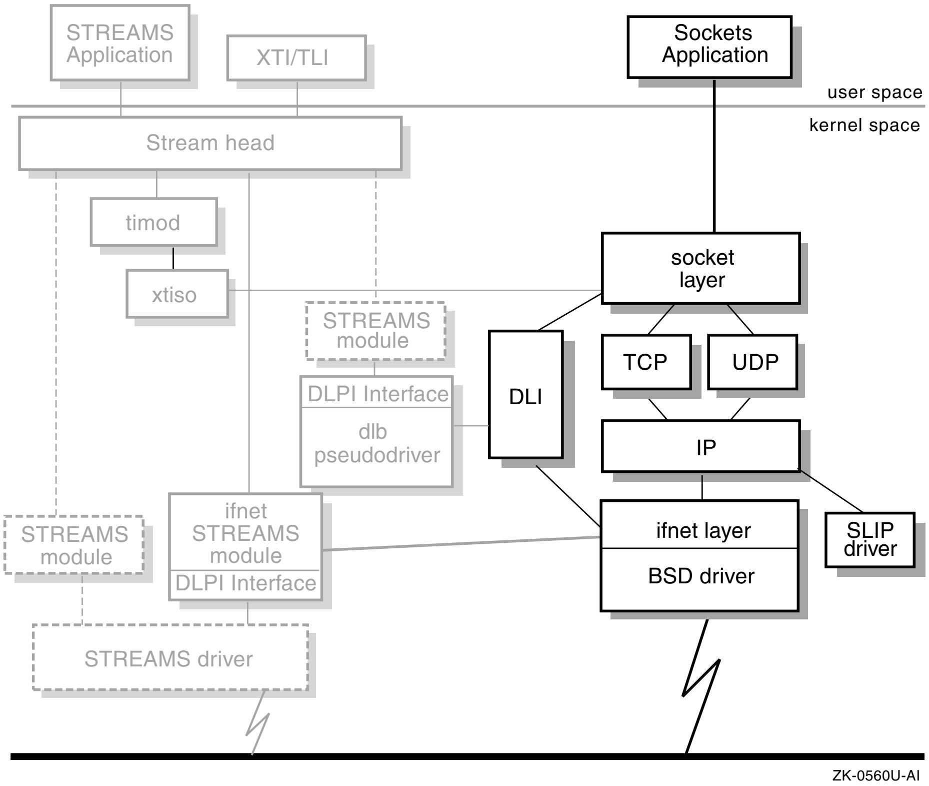
To use the XNS4.0 standard implementation in your program, you must compile your program using the `c89` compiler command. See `standards(5)` for additional information. The examples in this chapter are based on the XNS4.0 standard. See Section 4.5 for information on the differences between the XNS4.0, POSIX 1003.1g Draft 6.6, and the BSD interfaces.

This chapter contains the following information:

- Overview of the sockets framework
- Description of the application interface to sockets
- Information on how to use sockets
- Information on writing network applications
- Information on the BSD socket interfaces
- Explanation of common socket error messages
- Information about advanced topics

Figure 4–1 highlights the sockets framework and shows its relationship to the rest of the network programming environment:

Figure 4–1: The Sockets Framework



4.1 Overview of the Sockets Framework

The sockets framework consists of:

- A set of abstractions, such as communication domains and socket types, that defines socket communication properties
- A programming interface, or set of system and library calls, used by application programs to access the socket framework
- Kernel resources, including networking protocols, that application programs access using system and library calls

  The operating system implements the Internet Protocol suite and UNIX domain using sockets to achieve interprocess communication. It also implements BSD-based device drivers that are accessed using sockets system calls.

4.1.1 Communication Properties of Sockets

This section describes the abstractions and definitions that underlie sockets communication properties.

4.1.1.1 Socket Abstraction

Sockets function as endpoints of communication. A single socket is one endpoint; a pair of sockets constitutes a two-way communication channel that enables unrelated processes to exchange data locally and over networks.

Application programs request the operating system to create a socket when one is needed. The operating system returns a socket descriptor that the program uses to reference the newly created socket for further operations.

Sockets have the following characteristics:

- Exist only as long as some process holds a descriptor referencing them.
- Are referenced by descriptors and have qualities similar to those of a character special device. Read, write, and select operations are performed on sockets by using the appropriate system calls.
- Can be created in pairs or given names and used to rendezvous with other sockets in a communications domain, accepting connections from these sockets or exchanging messages with them.

Sockets are typed according to their communication properties. See Section 4.1.1.3 for a description of the available socket types.

4.1.1.2 Communication Domains

Communication domains define the semantics of communication between systems whose hardware and software differ. Communication domains specify the following:

- A set of protocols called the protocol family
- A set of rules for manipulating and interpreting names
- A collection of related socket address formats (an address family)

  The socket address for the Internet communication domain contains an Internet address and a port number. The socket address for the UNIX communication domain contains a local pathname.

  See Section 4.2.3.4 for more information on socket-related data structures.

The operating system provides default support for the following socket domains[1]:

[1] The operating system can also be configured to support the AF_DLI domain. For information about the Data Link Interface and using the AF_DLI domain, see Appendix F.

- UNIX domain

  The operating system provides socket communication between processes running on the same system when a domain of AF_UNIX is specified. In the UNIX communication domain, sockets are named with UNIX pathnames, such as `/dev/printer`.

- Internet domain

  The operating system provides socket communication between a process running locally and one running on a remote host when a domain of AF_INET or AF_INET6 is specified. This domain requires that TCP/IP be configured and running on your system.

Table 4–1 summarizes the characteristics of the UNIX and Internet domains.

Table 4–1: Characteristics of the UNIX and Internet Communication Domains

	UNIX	Internet
Socket Types	SOCK_STREAM, SOCK_DGRAM	SOCK_STREAM, SOCK_DGRAM, SOCK_RAW.
Naming	String of ASCII characters, for example, `/dev/printer`.	32-bit IP Version 4 address plus 16-bit port number (AF_INET), 128-bit IP Version 6 address plus 16-bit port number (AF_INET6).
Security	Process connecting to a pathname must have write access to it.	Not applicable.
Raw Access	Not applicable.	Privileged process can access the raw facilities of IP. Raw socket is associated with one IP protocol number, and receives all traffic received for that protocol.

4.1.1.3 Socket Types

Each socket has an associated abstract type which describes the semantics of communications using that socket type. Properties such as reliability, ordering, and prevention of duplication of messages are determined by the socket type. The basic set of socket types is defined in the `<sys/socket.h>` header file.

Note

Typically, header file names are enclosed in angle brackets (< >). To obtain the absolute path to the header file, prepend

`/usr/include/` to the information enclosed in the angle brackets. In the case of `<sys/socket.h>`, `socket.h` is located in the `/usr/include/sys` directory.

Within the UNIX and Internet domains you can use the following socket types:

SOCK_DGRAM

Provides datagrams that are connectionless messages of a fixed maximum length where each message can be addressed individually. This type of socket is generally used for short messages because the order and reliability of message delivery is not guaranteed. An important characteristic of a datagram socket is that record boundaries in data are preserved, so individual datagrams are kept separate when they are read.

Often datagrams are used for requests that require a response or responses from the recipient, such as with the `finger` program. If the recipient does not respond in a specified period of time, the sending application can repeat the request. The time period varies with the communication domain.

In the UNIX domain, SOCK_DGRAM is similar to a message queue. In the Internet domain, SOCK_DGRAM is implemented using the User Datagram Protocol (UDP).

SOCK_STREAM

Provides sequenced, two-way byte streams across a connection with a transmission mechanism for out-of-band data. The data is transmitted on a reliable basis, in order.

In the UNIX domain, SOCK_STREAM is like a full-duplex pipe. In the Internet domain, SOCK_STREAM is implemented using the Transmission Control Protocol (TCP).

SOCK_RAW

Provides access to network protocols and interfaces. Raw sockets are only available to privileged processes.

A raw socket allows an application to have direct access to lower-level communications protocols. Raw sockets are intended for advanced users who want to employ protocol features not directly accessible through a normal interface, or who want to build new protocols using existing lower-level protocols. You can also use SOCK_RAW to communicate with hardware interfaces.

Raw sockets are normally datagram-oriented, though their exact characteristics depend on the interface provided by the protocol. They are available only within the Internet domain.

4.1.1.4 Socket Names

Sockets can be named, which allows unrelated processes on a system or network to locate a specific socket and to exchange data with it. The bound name is a variable-length byte string that is interpreted by the supporting protocol or protocols. Its interpretation varies from communication domain to communication domain. In the Internet domain, names contain an Internet address and port number, and the family is either AF_INET or AF_INET6; AF_INET sockets support IPv4 communication, whereas AF_INET6 sockets support both IPv4 and IPv6 communication. In the UNIX domain, names contain a pathname and the family is AF_UNIX.

Communicating processes are bound by an association. In the Internet domain, an association comprises a protocol, local and foreign addresses, and local and foreign ports. When a name is bound to a socket in the Internet domain, the local address and port are specified.

In the UNIX domain, an association comprises local pathnames. Binding a name to a socket in the UNIX domain means specifying a pathname.

In most domains, associations must be unique.

4.2 Application Interface to Sockets

The kernel implementation of sockets separates the networking subsystem into the following three interacting layers:

- The socket layer which supplies the interface between the application program and the lower layers, such as the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP) and IP.
- The protocol layer which consists of transport layer protocols (TCP and UDP) and network layer protocols (IP).
- The device layer which consists of the `ifnet` layer and the device driver.

In addition to the abstractions described in Section 4.1.1, the socket interface comprises system and library calls, library functions, and data structures that enable you to manipulate sockets and send and receive data.

Additionally, the kernel provides ancillary services to the sockets framework, such as buffer management, message routing, standardized interfaces to the protocols, and interfaces to the network interface drivers for use by the various network protocols.

4.2.1 Modes of Communication

The sockets framework supports connection-oriented and connectionless modes of communication. Connection-oriented communication means that the application specifies a socket type in a communication domain that supports a connection-oriented protocol. For example, an application could open a SOCK_STREAM socket in the AF_INET domain. SOCK_STREAM sockets in the AF_INET and AF_INET6 domains are supported by the TCP protocol, which is a connection-oriented protocol.

Connectionless communication means that the application specifies a socket type in a communication domain that supports a connectionless protocol. For example, a SOCK_DGRAM socket in the AF_INET communication domain is supported by the UDP protocol, which is a connectionless protocol.

4.2.1.1 Connection-Oriented Communication

TCP is the connection-oriented protocol implemented on this operating system. TCP is a reliable end-to-end transport protocol that provides for recovery of lost data, transmission errors, and failures of intervening gateways. TCP ensures accurate delivery of data by requiring that two processes be connected before communicating. TCP/IP connections are often compared to telephone connections. Data passed through a SOCK_STREAM socket in the AF_INET or AF_INET6 domain is divided into segments and identified by sequence numbers. The remote process acknowledges receipt of data by including sequence numbers in the acknowledgement. If data is lost enroute, it is resent; thus ensuring that data arrives in the correct sequence to the application.

For applications where large amounts of data are exchanged and the sequence in which the data arrives is important, connection-oriented communication is preferable. File transfer programs are a good example of applications that benefit from the connection-oriented mode of communication offered by TCP.

4.2.1.2 Connectionless Communication

UDP is the connectionless protocol implemented on the operating system. UDP functions as follows:

- Delivers messages based on the messages’ address information
- Requires no connection between communicating processes

- Does not use acknowledgements to ensure that data arrives
- Does not order incoming messages
- Provides no feedback to control the rate at which data is exchanged between hosts

UDP messages can be lost, duplicated, or arrive out of order.

Where small amounts of data are exchanged and sequencing is not vital, connectionless communication works well. A good example of a program that uses connectionless communication is the `rwhod` daemon, which periodically broadcasts UDP packets containing system information to the network. It matters little whether or in what sequence those packets are delivered.

UDP is also appropriate for applications that use IP multicast for delivery of datagrams to a subset of hosts on a local area network.

4.2.2 Client/Server Paradigm

The most commonly used paradigm in constructing distributed applications is the client/server model. A server process offers services to a network; a client process uses those services. The client and server require a well-known set of conventions before services are rendered and accepted. This set of conventions a protocol comprises that must be implemented at both ends of a connection. Depending on the situation, the protocol can be connection-oriented (asymmetric) or connectionless (symmetric).

In a connection-oriented protocol, such as TCP, one side is always recognized as the server and the other as the client. The server binds a socket to a well-known address associated with the service and then passively listens on its socket. The client requests services from the server by initiating a connection to the server's socket. The server accepts the connection and then server and client can exchange data. An example of a connection-oriented protocol application is Telnet.

In a connectionless protocol, such as UDP, either side can play the server or client role. The client does not establish a connection with the server; instead, it sends a datagram to the server's address. Similarly, the server does not accept a connection from a client. Rather, it issues a `recvfrom` system call that waits until data arrives from a client. (See Section 4.3.6.)

4.2.3 System Calls, Library Calls, Header Files, and Data Structures

This section lists the system and library calls that the socket layer comprises. It also lists the header files that define socket-related constants and structures, and describes some of the most important data structures contained in those header files.

4.2.3.1 Socket System Calls

Table 4–2 lists the socket system calls and briefly describes their function. Note that each call has an associated reference page by the same name.

Table 4–2: Socket System Calls

System Call	Description
accept	Accepts a connection on a socket to create a new socket.
bind	Binds a name to a socket.
connect	Initiates a connection on a socket.
getpeername	Gets the name of the connected peer.
getsockname	Gets the socket name.
getsockopt	Gets options on sockets.
listen	Listens for socket connections and specifies the maximum number of queued requests.
recv	Receives messages, peeks at incoming data, and receives out-of-band data.
recvfrom	Receives messages. Has all of the functions of the recv call, plus supplies the address of the peer process.
recvmsg	Receives messages. Has all of the functions of the recv and recvfrom calls, plus receives specially interpreted data (access rights), and performs scatter I/O operations on message buffers.
send	Sends messages. Also sends out-of-band data and normal data without network routing.
sendmsg	Sends messages. Has all of the functions of the send and sendto calls, plus transmits specially interpreted data (access rights), and performs gather I/O operations on message buffers.
sendto	Sends messages. Has all of the functions of the send call, plus supplies the address of the peer process.
setsockopt	Sets socket options.
shutdown	Shuts down all socket send and receive operations.
socket	Creates an endpoint for communication and returns a descriptor.
socketpair	Creates a pair of connected sockets.

4.2.3.2 Socket Library Calls

Application programs use socket library calls to construct network addresses for use by the interprocess communications facilities in a distributed environment.

Network library subroutines map the following items:

- Host names to network addresses
- Network names to network numbers
- Protocol names to protocol numbers
- Service names to port numbers

Additional socket library calls exist to simplify manipulation of names and addresses.

An application program must include the <netdb.h> header file when using any of the socket library calls.

Host Names

Application programs use the following network library routines to map Internet host names to addresses:

- gethostbyname (AF_INET only)
- gethostbyaddr (AF_INET only)
- getaddrinfo (AF_INET and AF_INET6)
- getnameinfo (AF_INET and AF_INET6)

The gethostbyname routine takes an Internet host name and returns a hostent structure, while the gethostbyaddr routine maps Internet host addresses into a hostent structure. The hostent structure consists of the following components:

```
struct hostent {
   char *h_name;        /* official name of host */
   char **h_aliases;    /* alias list */
   int  h_addrtype;     /* host address type (AF_INET or AF_INET6) */
   int  h_length;       /* length of address */
   char **h_addr_list;  /* list of addresses, null terminated
                           first address, network byte order */
#define h_addr h_addr_list[0]
};
```

The gethostbyaddr and gethostbyname subroutines return the official name of the host and its public aliases, along with the address family and a null terminated list of variable-length addresses. This list of addresses is required because it is possible for a host to have many addresses with the same name.

The database for these calls is the /etc/hosts file. If the named name server is running, the hosts database is maintained on a designated server on the network. Because of the differences in the databases and their access protocols, the information returned can differ. When using the /etc/hosts version of gethostbyname, only one address is returned, but all listed aliases are included. The named version can return alternate addresses, but does not provide any aliases other than one given as a parameter value.

The getaddrinfo routine takes an Internet node name or service name and returns one or more addrinfo structures, while the getnameinfo routine takes a sockaddr and returns a host name (if requested) and service name for the port (if requested).

The database for these calls is both the /etc/ipnodes file and /etc/hosts file. If the named name server is running, the hosts database is maintained on a designated server on the network. Because of the differences in the databases and their access protocols, the information returned can differ. When using the /etc/ipnodes and /etc/hosts version of getaddrinfo, two addresses may be returned (one from each file), depending on the request; no aliases are returned. The named version can return alternate addresses, but does not provide any aliases other than one given as a parameter value.

Network Names

Application programs use the following network library routines to map network names to numbers and network numbers to names:

- getnetbyaddr
- getnetbyname
- getnetent

The getnetbyaddr, getnetbyname, and getnetent routines extract their information from the /etc/networks file and return a netent structure, as follows:

```
struct netent {
   char          *n_name;     /* official name of net */
   char          **n_aliases; /* alias list */
   int           n_addrtype;  /* net address type */
   in_addr_t     n_net;       /* network number, host byte order */
};
```

Protocol Names

Application programs use the following network library routines to map protocol names to protocol numbers:

- getprotobynumber
- getprotobyname

- getprotoent

The getprotobynumber, getprotobyname, and getprotoent subroutines extract their information from the /etc/protocols file and return the protoent entry, as follows:

```
struct protoent {
   char *p_name;     /* official protocol name */
   char **p_aliases; /* alias list */
   int  p_proto;     /* protocol number */
};
```

Service Names

Application programs use the following network library routines to map service names to port numbers:

- getservbyname
- getservbyport
- getservent

A service is expected to reside at a specific port and employ a particular communication protocol. This view is consistent with the Internet domain, but inconsistent with other network architectures. Further, a service can reside on multiple ports. If this occurs, the higher-level library routines must be bypassed or extended. Services available are contained in the /etc/services file. A service mapping is described by the servent structure, as follows:

```
struct servent {
   char *s_name;     /* official service name */
   char **s_aliases; /* alias list */
   int  s_port;      /* port number, network byte order */
   char *s_proto;    /* protocol to use */
};
```

The getservbyname routine maps service names to a servent structure by specifying a service name and, optionally, a qualifying protocol. Thus, the following call returns the service specification for a Telnet server by using any protocol:

```
sp = getservbyname("telnet", (char *) NULL);
```

In contrast, the following call returns only the Telnet server that uses the TCP protocol:

```
sp = getservbyname("telnet", "tcp");
```

The getservbyport and getservent routines are also provided. The getservbyport routine has an interface similar to that provided by getservbyname; an optional protocol name can be specified to qualify lookups.

Network Byte Order Translation

When you have to create or interpret Internet Protocol (IP) suite data in your program, standard methods exist for conversion. The IP suite ensures consistency by requiring particular data formats. The operating system provides functions that let a program convert data to and from those formats. Additionally, the Internet Protocol suite assumes that the most significant byte is in the lowest address, a format known as big-endian. Functions are available to convert from network-byte order to host-byte order and vice versa.

Four functions ensure that data passed by your program is interpreted correctly by the network and vice versa:

- `htonl`
- `htons`
- `ntohl`
- `ntohs`

Application programs use the following related network library routines to manipulate Internet address strings and binary address quantities:

- `inet_addr` (AF_INET only)
- `inet_lnaof` (AF_INET only)
- `inet_makeaddr` (AF_INET only)
- `inet_netof` (AF_INET only)
- `inet_network` (AF_INET only)
- `inet_ntoa` (AF_INET only)
- `getnameinfo` (AF_INET and AF_INET6)
- `getaddrinfo` (AF_INET and AF_INET6)

Table 4–3 lists and briefly describes the socket library calls. Note that each call has an associated reference page by the same name. The socket library calls are part of `libc`, so there is no need to link in a special library.

Table 4–3: Socket Library Calls

Name	Description
`endhostent`	Ends a series of host entry lookups.
`endnetent`	Ends a series of network entry lookups.
`endprotoent`	Ends a series of protocol entry lookups.
`endservent`	Ends a series of service entry lookups.

Table 4–3: Socket Library Calls (cont.)

Name	Description
freeaddrinfo	Frees addrinfo structures and storage that were returned by getaddrinfo.
getaddrinfo	Given the name of a host and an optional address family, retrieves the host entry from either the name server (named), the /etc/ipnodes file, or the /etc/hosts file. Translates a node's address in standard numeric string format to an Internet address as part of a sockaddr structure.
gethostbyaddr	Given the address of a host, retrieves the host entry from either the name server (named) or the /etc/hosts file.
gethostbyname	Given the name of a host, retrieves the host entry from either the name server (named) or the /etc/hosts file.
gethostent	Retrieves the next host entry from either the name server (named) or the /etc/hosts file, opening this file if necessary.
getnameinfo	Given the sockaddr structure containing the address of a node, retrieves the host entry from either the name server (named), the /etc/ipnodes file, or the /etc/hosts file. Translates an Internet address from the sockaddr structure into a standard numeric string format.
getnetbyaddr	Given the address of a network, retrieves the network entry from the /etc/networks file.
getnetbyname	Given the name of a network, retrieves the network entry from the /etc/networks file.
getnetent	Retrieves the next network entry from the /etc/networks file, opening this file if necessary.
getprotobyname	Given the protocol name, retrieves the protocol entry from the /etc/protocols file.
getprotobynumber	Given the protocol number, retrieves the protocol entry from the /etc/protocols file.
getprotoent	Retrieves the next protocol entry from the /etc/protocols file, opening this file if necessary.
getservbyname	Given the name of a service, retrieves the service entry from the /etc/services file.
getservbyport	Given the port number of a service, retrieves the service entry from the /etc/services file.

Table 4–3: Socket Library Calls (cont.)

Name	Description
getservent	Retrieves the next service entry from the /etc/services file, opening this file if necessary.
htonl	Converts a 32-bit integer from host-byte order to Internet network-byte order.
htons	Converts an unsigned short integer from host-byte order to Internet network-byte order.
inet_addr	Breaks apart a character string representing numbers expressed in the Internet standard dot (.) notation, and returns an Internet address.
inet_lnaof	Breaks apart an Internet host address and returns the local network address.
inet_makeaddr	Constructs an Internet address from an Internet network number and a local network address.
inet_ntoa	Translates an Internet address value into a character string.
inet_netof	Breaks apart an Internet host address and returns the network number.
inet_network	Breaks apart a character string representing numbers expressed in the Internet standard dot (.) notation, and returns an Internet network number.
ntohl	Converts a 32-bit integer from Internet network standard-byte order to host-byte order.
ntohs	Converts an unsigned short integer from Internet network-byte order to host-byte order.
sethostent	Begins a series of host entry lookups.
setnetent	Begins a series of network entry lookups.
setprotoent	Begins a series of protocol entry lookups.
setservent	Begins a series of service entry lookups.

4.2.3.3 Header Files

Socket header files contain data definitions, structures, constants, macros, and options used by the socket system calls and subroutines. An application program must include the appropriate header file to make use of structures or other information a particular socket system call or subroutine requires. Table 4–4 lists commonly used socket header files.

Table 4–4: Header Files for the Socket Interface

File Name	Description
<sys/socket.h>	Contains data definitions and socket structures. You need to include this file in all socket applications.
<sys/types.h>	Contains data type definitions. You need to include this file in all socket applications. This header file is included in <sys/socket.h>.
<sys/un.h>	Defines structures for the UNIX domain. You need to include this file in your application if you plan to use UNIX domain sockets.
<netinet/in.h>	Defines constants and structures for the Internet domain. You need to include this file in your application if you plan to use TCP/IP in the Internet domain.
<netdb.h>	Contains data definitions for socket subroutines. You need to include this file in your application if you plan to use TCP/IP and need to look up host entries, network entries, protocol entries, or service entries.

4.2.3.4 Socket Related Data Structures

This section describes the following data structures:

- sockaddr
- sockaddr_in
- sockaddr_in6
- sockaddr_storage
- sockaddr_un
- msghdr

The sockaddr structures contain information about a socket's address format. Because the communication domain in which an application creates a socket determines its address format, it also determines its data structure.

Socket address data structures are defined in the header files described in Section 4.2.3.3. Which header file is appropriate depends on the type of socket you are creating. The possible types of socket address data structures are as follows:

struct sockaddr

Defines the generic version of the socket address structure. These sockets are limited to 14 bytes of direct addressing. The <sys/socket.h> file contains the sockaddr structure, which contains the following elements:

```
unsigned char   sa_len;         /* total length */
sa_family_t     sa_family;      /* address family */
char            sa_data[14];    /* actually longer;
                                   address value
```

The sa_len parameter defines the total length. The sa_family parameter defines the socket address family or domain, which is AF_UNIX for the UNIX domain, or AF_INET or AF_INET6 for the Internet domain. The contents of sa_data depend on the protocol in use, but generally a socket name consists of a machine-name part and a port-name or service-name part.

struct sockaddr_storage

Defines Internet domain sockets (AF_INET and AF_INET6 address families) used for machine-to-machine communication across a network and local interprocess communication. This allows applications to handle multiple address families with a single variable. The <sys/socket.h> file contains the sockaddr_storage structure. The sockaddr_storage structure contains the following elements:

```
unsigned char  ss_len;     /* address length */
sa_family_t    ss_family;  /* address family */
char           __ss_pad1[_SS_PAD1SIZE];  /* pad to alignment field */
ulong_t        __ss_align;               /* force structure alignment */
char           __ss_pad2[_SS_PAD2SIZE];  /* pad to desired size */
```

The _SS_PAD1SIZE and _SS_PAD2SIZE variables are also defined in <sys/socket.h>.

struct sockaddr_un

Defines UNIX domain sockets used for communications between processes on the same machine. These sockets require the specification of a full pathname. The <sys/un.h> header file contains the sockaddr_un structure. The sockaddr_un structure contains the following elements:

```
unsigned char   sun_len;        /* sockaddr len including null*/
sa_family_t     sun_family;     /* AF_UNIX, address family*/
char            sun_path[];     /* path name */
```

UNIX domain protocols (AF_UNIX) have socket addresses up to PATH_MAX plus 2 bytes long. The PATH_MAX parameter defines the maximum number of bytes of the pathname.

struct sockaddr_in

Defines Internet domain sockets (AF_INET address family) used for machine-to-machine communication across a network and local interprocess communication. The <netinet/in.h> file contains the sockaddr_in structure. The sockaddr_in structure contains the following elements:

```
     unsigned char   sin_len;
     sa_family_t     sin_family;
     in_port_t       sin_port;
     struct  in_addr sin_addr;
```

```
struct sockaddr_in6
```

Defines Internet domain sockets (AF_INET6 address family) used for machine-to-machine communication across a network and local interprocess communication. The <netinet/in.h> file contains the sockaddr_in6 structure. The sockaddr_in6 structure contains the following elements:

```
     uint8_t          sin6_len;
     sa_family_t      sin6_family;
     in_port_t        sin6_port;
     uint32_t         sin6_flowinfo
     struct in6_addr  sin6_addr;
     uint32_t         sin6_scope_id
```

The in6_addr structure stores the address in network byte order as an array of sixteen 8-bit elements.

The msghdr data structure, which is defined in the <sys/socket.h> header file, allows applications to pass access rights to system-maintained objects (such as files, devices, or sockets) using the sendmsg and recvmsg system calls. (See Section 4.3.6 for information on the sendmsg and recvmsg system calls.) The processes transmitting data must be connected with a UNIX domain socket.

The data structure also allows AF_INET sockets to receive certain data. See ip(7) for the descriptions of the IP_RECVDSTADDR and IP_RECVOPTS options.

The msghdr data structure consists of the following components:

```
struct msghdr {
        void           *msg_name;       /* optional address */
        size_t          msg_namelen;    /* size of address */
        struct iovec   *msg_iov;        /* scatter/gather array */
        int             msg_iovlen;     /* # elements in msg_iov */
        void           *msg_control;    /* ancillary data, see below */
        size_t          msg_controllen; /* ancillary data buffer len */
        int             msg_flags;      /* flags on received message */
};
```

In addition to the XNS4.0 msghdr data structure, the operating system also supports the 4.3BSD, 4.4BSD, and the POSIX 1003.1g Draft 6.6 versions of this data structure. The BSD versions of the msghdr data structure are described in greater detail in Section 4.5.

4.3 Using Sockets

This section outlines the steps required to create and use sockets. Connection-oriented and connectionless modes of communication are described in the following sections:

- Creating sockets

  Describes how to create a socket with the `socket` and `socketpair` system calls.

- Binding names and addresses

  Describes how to bind a name and address to a socket with the `bind` system call.

- Establishing connections (clients)

  Describes how to use the `connect` system call on a client to connect to a server.

- Accepting connections (servers)

  Describes how to use the `listen` and `accept` system calls to connect a server to a client.

- Setting and getting socket options

  Describes how to use the `setsockopt` and `getsockopt` system calls to set and retrieve the values of socket characteristics.

- Transferring data

  Describes how to use the `read` and `write` system calls, as well as the `send` and `recv` related system calls to transmit data.

- Shutting down sockets

  Describes how to use the `shutdown` system call to shut down a socket.

- Closing sockets

  Describes how to use the `close` system call to close a socket.

4.3.1 Creating Sockets

The first step in using sockets is creating a socket. Sockets are opened, or created, with the `socket` or `socketpair` system calls.

The `socket` call returns a socket descriptor, which is an a nonnegative integer that the application program uses to reference the newly created socket in subsequent system calls. The socket descriptor returned is the lowest unused number available in the calling process for such descriptors and is an index into the kernel descriptor table.

See `socket`(2) for function syntax, parameters, and errors.

For example, to create a stream socket in the Internet domain for use with the AF_INET address family, you can use the following call:

```
if ((s = socket(AF_INET, SOCK_STREAM,0)) == -1 ) {
         fprintf(file1,"socket() failed\n");
         local_flag = FAILED;
     }
```

This call results in the creation of a stream socket with the TCP protocol providing the underlying communication support. To create a datagram socket in the UNIX domain, you can use the following call:

```
if ((s = socket(AF_UNIX, SOCK_DGRAM,0)) == -1 ) {
         fprintf(file1, "socket() failed\n");
         local_flag = FAILED;
     }
```

This call results in the creation of a datagram socket with a UNIX domain protocol providing the underlying communication support.

The `socketpair` system call can also be used to create sockets. The `socketpair` system call creates an unnamed pair of sockets that are already connected.

The `socketpair` system call returns a pair of socket descriptors, which are a nonnegative integers, that the application uses to reference the newly created socket pair in subsequent system calls.

See `socketpair`(2) for function syntax, parameters, and errors.

The following example shows how to create a socket pair:

```
{
⋮

 int sv[2];
⋮


    if ((s = socketpair (AF_UNIX, SOCK_STREAM, 0, sv)) < 0) {
             local_flag=FAILED;
             fprintf(file1, "socketpair() failed\n");
        }
⋮

}
```

4.3.1.1 Setting Modes of Execution

Sockets can be set to blocking or nonblocking I/O mode. The O_NONBLOCK `fcntl` operation is used to determine this mode. When O_NONBLOCK is clear (not set), which is the default, the socket is in blocking mode. In blocking mode, when the socket tries to do a `read` and the data is not available, it waits for the data to become available.

When O_NONBLOCK is set, the socket is in nonblocking mode. In nonblocking mode, when the calling process tries to do a `read` and the data is not available, the socket returns immediately with the EWOULDBLOCK error code. It does not wait for the data to become available. Similarly, during writing, when a socket has O_NONBLOCK set and the output queue is full, an attempt by the socket to `write` causes the process to return immediately with an error code of EWOULDBLOCK.

The following example shows how to mark a socket as nonblocking:

```
#include <fcntl.h>
⋮

int s;
⋮

if (fcntl(s, F_SETFL, O_NONBLOCK) < 0)
   perror("fcntl F_SETFL, O_NONBLOCK");
   exit(1);
}
⋮
```

When performing nonblocking I/O on sockets, a program must check for the EWOULDBLOCK error, which is stored in the global value `errno`. The EWOULDBLOCK error occurs when an operation normally blocks, but the socket on which it was performed is marked as nonblocking. The following socket system calls all return the EWOULDBLOCK error code:

- `accept`
- `connect`
- `send`
- `sendto`
- `sendmsg`
- `recv`
- `recvfrom`
- `recvmsg`
- `read`
- `write`

Processes that use these system calls on nonblocking sockets must be prepared to deal with the EWOULDBLOCK return codes.

When an operation, such as a `send`, cannot be completed but partial writes are permissible (for example, when using a SOCK_STREAM socket), the data that can be sent immediately is processed, and the return value indicates the amount of data actually sent.

4.3.2 Binding Names and Addresses

The `bind` system call associates an address with a socket. The domain for the socket is established with the `socket` system call. Regardless of the domain in which the `bind` system call is used, it allows the local process to fill in information about itself, for example, the local port or local pathname. This information allows the server application to be located by a client application.

The following example shows how to use the `bind` system call on a SOCK_STREAM socket created in the Internet domain for the AF_INET address family:

```
#define PORT 3000

int     retval;           /* General return value */
int     s1_descr;         /* Socket 1 descriptor */
.
.
.
struct sockaddr_in  sock1addr; /* Address struct for socket1.*/
.
.
.
s1_descr = socket (AF_INET, SOCK_STREAM, 0);
if (s1_descr < 0)                     /* Call failed */
.
.
.
bzero(&sock1addr, sizeof(sock1addr));
sock1addr.sin_family        = AF_INET;
sock1addr.sin_addr.s_addr   = INADDR_ANY;
sock1addr.sin_port          = htons(PORT);
retval = bind (s1_descr, (struct sockaddr *) &sock1addr, sizeof(sock1addr));
if (retval < 0)          /* Call failed */
.
.
.
```

See `bind`(2) for function syntax, parameters, and errors. See Section 4.7.3 for advanced information on binding names and addresses.

4.3.3 Establishing Connections

Sockets are created in the unconnected state. Client processes use the `connect` system call to connect to a server process or to store a server's address locally, depending on whether the communication is connection-oriented or connectionless. For the Internet domain, the connect system call typically causes the local address, local port, foreign address, and foreign port of an association to be assigned.

The syntax of the connect system call depends on the communication domain. An error is returned if the connection was unsuccessful; any name automatically bound by the system remains, however. Applications should use the close system call to deallocate the socket and descriptor. Common errors associated with sockets are listed in Table 4–6 in Section 4.6. If the connection is successful, the socket is associated with the server and data transfer begins.

See connect(2) for function syntax, parameters, and errors.

Selecting a connection-oriented protocol in the Internet domain means choosing TCP. In such cases, the connect system call builds a TCP connection with the destination, or returns an error if it cannot. Client processes using TCP must call the connect system call to establish a connection before they can transfer data through a reliable stream socket (SOCK_STREAM).

Selecting a connectionless protocol in the Internet domain means choosing UDP. Client processes using connectionless protocols do not have to be connected before they are used. If connect is used under these circumstances, it stores the destination (or server) address locally so that the client process does not need to specify the server's address each time a message is sent. Any data sent on this socket is automatically addressed to the connected server process and only data received from that server process is delivered.

Only one connected address is permitted at any time for each socket; a second connect system call changes the destination address and a connect system call to a null address (for example, AF_INET address INADDR_ANY) causes a disconnect. The connect system call on a connectionless protocol returns immediately, since it results in the operating system recording the server's socket's address (as compared to a connection-oriented protocol, where a connect request initiates establishment of an end-to-end connection).

While a socket using a connectionless protocol is connected, errors from recent send system calls can be returned asynchronously. These errors can be reported on subsequent operations on the socket. A special socket option, SO_ERROR (used with the getsockopt system call), can be used to query the error status. A select system call, issued to determine when more data can be sent or received, will return true when a process has received an error indication.

In any case, the next operation will return the error and clear the error status.

See select(2) for function syntax, parameters, and errors.

The following is an example of the select system call:

```
if ( (ret_val = select(20,&read_mask,NULL,NULL,&tp)) != i )
```

4.3.4 Accepting Connections

A connection-oriented server process normally listens at a well-known address for service requests. That is, the server process remains dormant until a connection is requested by a client's connection to the server's address. Then, the server process wakes up and services the client by performing the actions the client requests.

Connection-oriented servers use the listen and accept system calls to prepare for and then accept connections with client processes.

The listen system call is usually called after the socket and bind system calls. It indicates that the server is ready to receive connect requests from clients.

See listen(2) for function syntax, parameters, and errors.

The server accepts a connection to a client by using the accept system call. An accept call blocks the server until a client requests service. This call returns a failure status if the call is interrupted by a signal such as SIGCHLD. Therefore, the return value from accept is checked to ensure that a connection was established.

See accept(2) for function syntax, parameters, and errors.

When the connection is made, the server normally forks a child process which creates another socket with the same properties as socket the socket on which it is listening. Note in the following example how the socket *s*, used by the parent for queuing connection requests, is closed in the child while the socket *g*, which is created as a result of the accept call, is closed in the parent. The address of the client is also handed to the doit routine because it is required for authenticating clients. After the accept system call creates the new socket, it allows the new socket to service the client's connection request while it continues listening on the original socket; for example:

```
for (;;) {
   int g, len = sizeof (from);

   g = accept(s, (struct sockaddr *)&from, &len);
   if (g < 0) {
      if (errno != EINTR)
         syslog(LOG_ERR, "rlogind: accept: %m");
      continue;
   }
   if (fork() == 0) {    /* Child */
      close(s);
      doit(g, &from);
   }
   close(g);             /* Parent */
}
```

Connectionless servers use the bind system call but, instead of using the accept system call, they use a recvfrom system call and then wait for client requests. No connection is established between the connectionless server and client during the process of exchanging data.

4.3.5 Setting and Getting Socket Options

In addition to binding a socket to a local address or connecting to a destination address, application programs must be able to control the socket. For example, with protocols that use time-out and retransmission, the application program may want to obtain or set the time-out parameters. It may also want to control the allocation of buffer space, determine if the socket allows transmission of a broadcast, or control processing of out-of-band data.

The getsockopt and setsockopt system calls provide the application program with the means to control socket operations. The setsockopt system call allows an application program to set a socket option by using the same set of values obtained with the getsockopt system call.

See setsockopt(2) for function syntax, parameters, and errors.

The following example shows how to set the SO_SNDBUF option on a socket in the Internet communication domain:

```
# include       <sys/socket.h>
.
.
.
int     retval;            /* General return value. */
int     s1_descr;          /* Socket 1 descriptor   */
int     sockbufsize=16384;
.
.
.
retval = setsockopt (s1_descr, SOL_SOCKET, SO_SNDBUF, (void *)

         &sockbufsize, sizeof(sockbufsize));
```

The getsockopt system call allows an application program to request information about the socket options that are set with the setsockopt system call. See getsockopt(2) for function syntax, parameters, and errors.

The following example shows how the getsockopt system call can be used to determine the size of the SO_SNDBUF on an existing socket:

```
#include <sys/socket.h>
⋮

int     retval;            /* General return value. */
int     s1_descr;          /* Socket 1 descriptor   */
int     sbufsize;
size_t  len = sizeof(sbufsize);
⋮

retval = getsockopt (s1_descr, SOL_SOCKET, SO_SNDBUF,
                     (void *)&sbufsize, &len);
```

The SOL_SOCKET parameter indicates that the general socket level code is to interpret the SO_SNDBUF parameter. The SO_SNDBUF parameter indicates the size of the send socket buffer in use on the socket.

Not all socket options apply to all sockets. The options that can be set depend on the address family and protocol the socket uses.

4.3.6 Transferring Data

Most of the work performed by the socket layer is in sending and receiving data. The socket layer itself does not impose any structure on data transmitted or received through sockets. Any data interpretation or structuring is logically isolated in the implementation of the communication domain.

The following are the system calls that an application uses to send and receive data:

- `read`
- `write`
- `send`
- `sendto`
- `recv`
- `recvfrom`
- `sendmsg`
- `recvmsg`

4.3.6.1 Using the read System Call

The `read` system call allows a process to receive data on a socket without receiving the sender’s address.

See `read`(2) for function syntax, parameters, and errors.

4.3.6.2 Using the write System Call

The write system call is used on sockets in the connected state. The destination of data transferred with the write system call is implicitly specified by the connection.

See write(2) for function syntax, parameters, and errors.

4.3.6.3 Using the send, sendto, recv and recvfrom System Calls

The send, sendto, recv, and recvfrom system calls are similar to the read and write system calls, sharing the first three parameters with them; however, additional flags are required. The flags, defined in the <sys/socket.h> header file, can be defined as a nonzero value if the application program requires one or more of the following:

Flag	Description
MSG_OOB	Send or receive out-of-band data.
MSG_PEEK	Look at data without reading. Valid for the recv and recvfrom calls.
MSG_DONTROUTE	Send data without routing packets. Valid for the send and sendto calls.

The MSG_OOB flag signifies out-of-band data, or urgent data, and is specific to stream sockets (SOCK_STREAM). See Section 4.7.4 for more information about out-of-band data.

The MSG_PEEK flag allows an application to preview the data that is available to be read, without having the system discard it after the recv or recvfrom call returns. When the MSG_PEEK flag is specified with a recv system call, any data present is returned to the user but treated as still unread. That is, the next read or recv system call applied to the socket returns the data previously previewed.

The MSG_DONTROUTE flag is currently used only by the routing table management process and is not discussed further.

send

The send system call is used on sockets in the connected state. The send and write system calls function almost identically; the only difference is that send supports the flags described at the beginning of this section.

See send(2) for function syntax, parameters, and errors.

sendto

The sendto system call is used on connected or unconnected sockets. It allows the process explicitly to specify the destination for a message.

See sendto(2) for function syntax, parameters, and errors.

recv

The recv system call allows a process to receive data on a socket without receiving the sender's address. The read and recv system calls function almost identically; the only difference is that recv supports the flags described at the beginning of this section.

See recv(2) for function syntax, parameters, and errors.

recvfrom

The recvfrom system call can be used on connected or unconnected sockets. The recvfrom system call has similar functionality to the recv call but it additionally allows an application to receive the address of a peer with whom it is communicating.

See recvfrom(2) for function syntax, parameters, and errors.

4.3.6.4 Using the sendmsg and recvmsg System Calls

The sendmsg and recvmsg system calls are distinguished from the other send and receive related system calls in that they allow unrelated processes on the local machine to pass file descriptors to each other. These two system calls are the only ones that support the concept of access rights, which means that the system has granted a process the right to access a system-maintained object. Using the sendmsg and recvmsg system calls they can pass that right to another process.

To pass access rights, the sendmsg and recvmsg system calls use the msghdr data structure. The msghdr data structure defines two parameters, the *msg_control* and *msg_controllen* that deal with the passing and receiving of access rights between processes. For more information on the msghdr data structure, see Section 4.2.3.4 and Section 4.5.2.

Although the sendmsg and recvmsg system calls can be used on connection-oriented or connectionless protocols and in the Internet or UNIX domains, for processes to pass descriptors they must be connected with a UNIX domain socket.

sendmsg

The sendmsg system call is used on connected or unconnected sockets. It transfers data using the msghdr data structure. For more information on the msghdr data structure, see Section 4.2.3.4 and Section 4.5.2.

See sendmsg(2) for function syntax, parameters, and errors.

The following is an example of the sendmsg system call:

```
struct msghdr send;
struct iovec saiov;
struct sockaddr destAddress;
char sendbuf[BUFSIZE];
⋮

send.msg_name = (void *)&destAddress;
send.msg_namelen =  sizeof(destAddress);
send.msg_iov = &saiov;
send.msg_iovlen = 1;
saiov.iov_base = sendbuf;
saiov.iov_len = sizeof(sendbuf);
send.msg_control = NULL;
send.msg_controllen = 0;
send.msg_flags = 0;
if ((i = sendmsg(s, &send, 0)) < 0) {
        fprintf(file1,"sendmsg() failed\n");
        exit(1);
}
```

recvmsg

The recvmsg system call is used on connected or unconnected sockets. It transfers data using the msghdr data structure. For more information on the msghdr data structure, see Section 4.2.3.4 and Section 4.5.2.

See recvmsg(2) for function syntax, parameters, and errors.

The following is an example of the recvmsg system call:

```
struct msghdr recv;
struct iovec recviov;
struct sockaddr_in recvaddress;
char recvbuf[BUFSIZE];
⋮

recv.msg_name = (void *) &recvaddress;
recv.msg_namelen = sizeof(recvaddress);
recv.msg_iov = &recviov;
recv.msg_iovlen = 1;
recviov.iov_base = recvbuf;
recviov.iov_len = sizeof(recvbuf);
recv.msg_control = NULL;
recv.msg_controllen = 0
recv.msg_flags = 0
if ((i = recvmsg(r, &recv, 0)) < 0) {
               fprintf(file1,"recvmsg() failed\n");
       exit(1);
}
⋮
```

4.3.7 Shutting Down Sockets

If an application program has no use for any pending data, it can use the `shutdown` system call on the socket prior to closing it. See `shutdown`(2) for function syntax, parameters, and errors.

4.3.8 Closing Sockets

The `close` system call is used to close sockets. See `close`(2) for function syntax, parameters, and errors.

Closing a socket and reclaiming its resources can be complicated. For example, a `close` system call is never expected to fail when a process exits. However, when a socket that is promising reliable delivery of data closes with data still queued for transmission or awaiting acknowledgment of reception, the socket must attempt to transmit the data. When the socket discards the queued data to allow the `close` call to complete successfully, it violates its promise to deliver data reliably. Discarding data can cause naive processes that depend on the implicit semantics of the `close` call to work unreliably in a network environment.

However, if sockets block until all data is transmitted successfully, a `close` system call may never complete in some communication domains.

The socket layer compromises in an effort to address the completion problem and still maintain the semantics of the `close` system call. In normal operation, closing a socket causes any queued but unaccepted connections to be discarded. If the socket is in a connected state, a disconnect is initiated. The socket is marked to indicate that a descriptor is no longer referencing it, and the `close` operation returns successfully. When the disconnect request completes, the network support notifies the socket layer, and the socket resources are reclaimed. The network layer attempts to transmit any data queued in the socket's send buffer, but there is no guarantee that it will succeed.

Alternatively, a socket can be marked explicitly to force the application program to linger when closing until pending data is flushed and the connection shuts down. This option is marked in the socket data structure by using the `setsockopt` system call with the SO_LINGER option.

Note

The `setsockopt` system call, using the linger option, takes a `linger` structure, which is defined in the `<sys/socket.h>` header file.

When an application program indicates that a socket is to linger, it also specifies a duration for the lingering period. If the lingering period expires before the disconnect is completed, the socket layer forcibly shuts down the socket, discarding any data that is still pending.

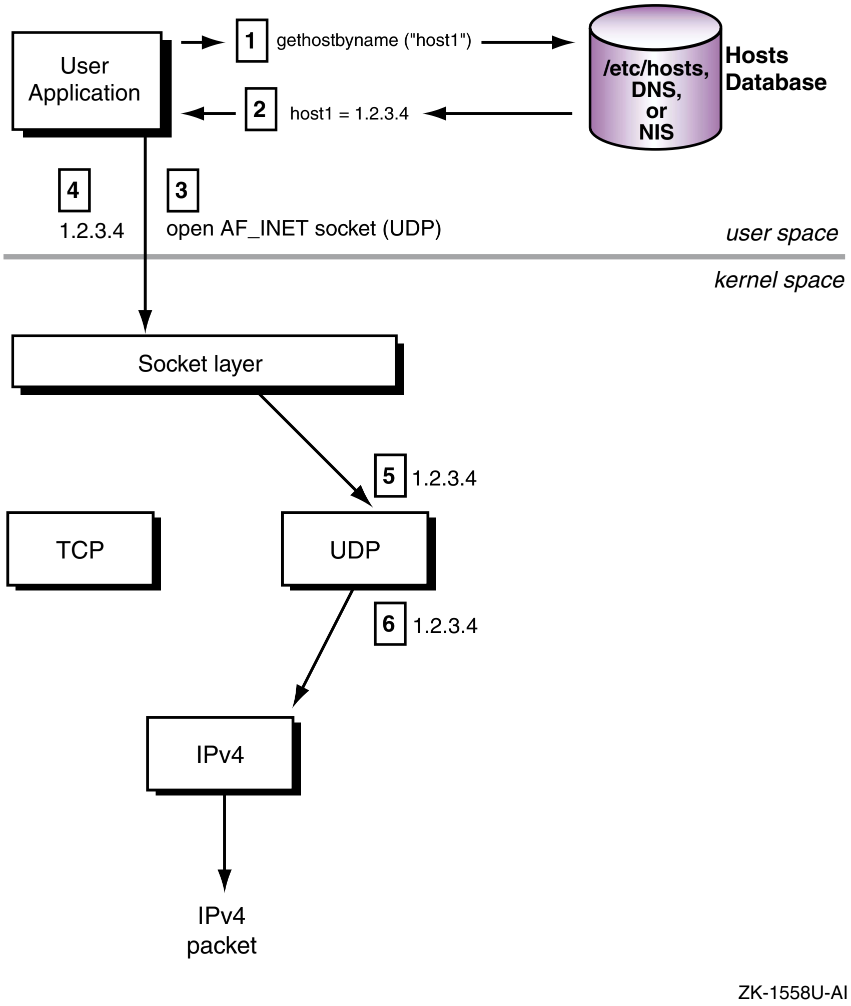
4.4 Creating Internet Applications

This section outlines the steps required to create and use applications in the Internet domain (IPv4 and IPv6) by building upon the information presented in Section 4.3. In addition, it summarizes the address testing macros that an application might use. Use this information here together with the porting guidelines in Section 4.7.1 to develop IPv6 applications.

4.4.1 Creating IPv4 Applications

Internet applications today use AF_INET sockets for IPv4 communications. Figure 4–2 shows an sample sequence of events for a client application that uses an AF_INET socket to send IPv4 packets.

Figure 4–2: Using AF_INET Socket for IPv4 Communications



1 Application calls `gethostbyname` and passes the host name, `host1`.

2 The search finds `host1` in the hosts database and `gethostbyname` returns the IPv4 address `1.2.3.4` in a structure of type `hostent`.

3 The application calls `socket` to create an `AF_INET` socket. The socket is a datagram socket (UDP) in this example, but could be a stream socket (TCP).

4 If the `socket` call is successful, the application fills in a `sockaddr_in` structure and calls `connect` to establish a connection with `host1`.
If the `connect` call is successful, the application calls `send` to send information to the `1.2.3.4` address.

5 The socket layer passes the information and address to the UDP module.

6 The UDP module puts the `1.2.3.4` address into the packet header and passes the information to the IPv4 module for transmission.

From this point, the application can do the following:

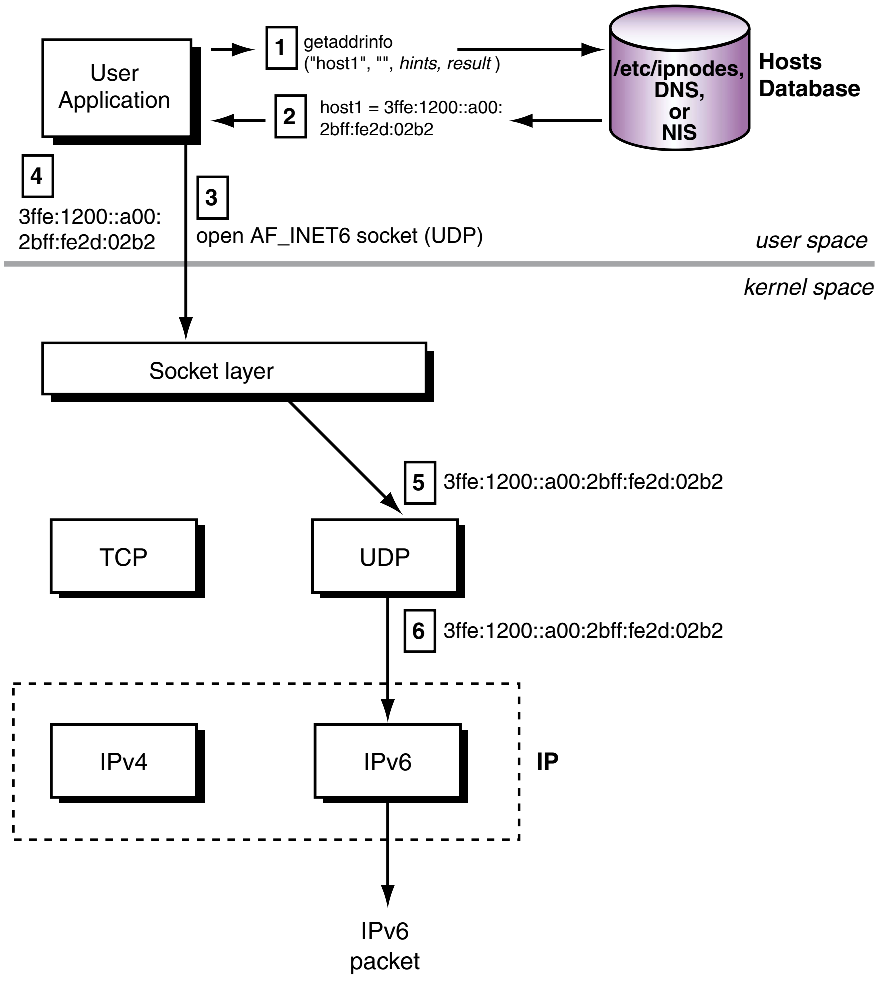
1. Call `recv` to wait for a response from the server system.
2. After it receives a response, call `gethostbyaddr` and pass it the server's address in a `sockaddr_in` structure. After the search finds the address in the hosts database, `gethostbyaddr` returns the host name in a structure of type `hostent`.
3. Call `inet_ntoa` to convert the server address to a text string.

Section C.1.1 contains sample client program code that demonstrates these steps.

4.4.2 Creating IPv6 Applications

Internet applications can use AF_INET6 sockets for IPv6 communication. In addition, the AF_INET6 sockets can also support IPv4 communication. Figure 4–3 shows the sequence of events for a client application that uses an AF_INET6 socket to send IPv6 packets.

Figure 4–3: Using AF_INET6 Socket for IPv6 Communications



1 Application calls `getaddrinfo` and passes the host name (`host1`), the `AF_INET6` address family hint, and the `AI_ADDRCONFIG` and `AI_V4MAPPED` flag hints. The flag hints tell the function that if an IPv6 address is found for `host1`, return it. See `getaddrinfo`(3) for a description of hints fields and values.

2 The search finds an IPv6 address for `host1` in the hosts database and `getaddrinfo` returns the IPv6 address `3ffe:1200::a00:2bff:fe2d:02b2` in one or more structures of type `addrinfo`.

3 The application calls `socket` to create an `AF_INET6` socket, using the address family and socket type contained in the `addrinfo` structure. The socket is a datagram socket (UDP) in this example, but could be a stream socket (TCP).

4 If the `socket` call is successful, the application calls `connect` to establish a connection with `host1`, using the host address and length in the `addrinfo` structure. If the `connect` call is successful, the application sends information to the `3ffe:1200::a00:2bff:fe2d:02b2` address.

> **Note**
>
> After using the information in the `addrinfo` structures, the application calls `freeaddrinfo` to free system resouces used by the structures.

5 The socket layer passes the information and address to the UDP module.

6 The UDP module identifies the IPv6 address and puts the `3ffe:1200::a00:2bff:fe2d:02b2` address into the packet header and passes the information to the IPv6 module for transmission.

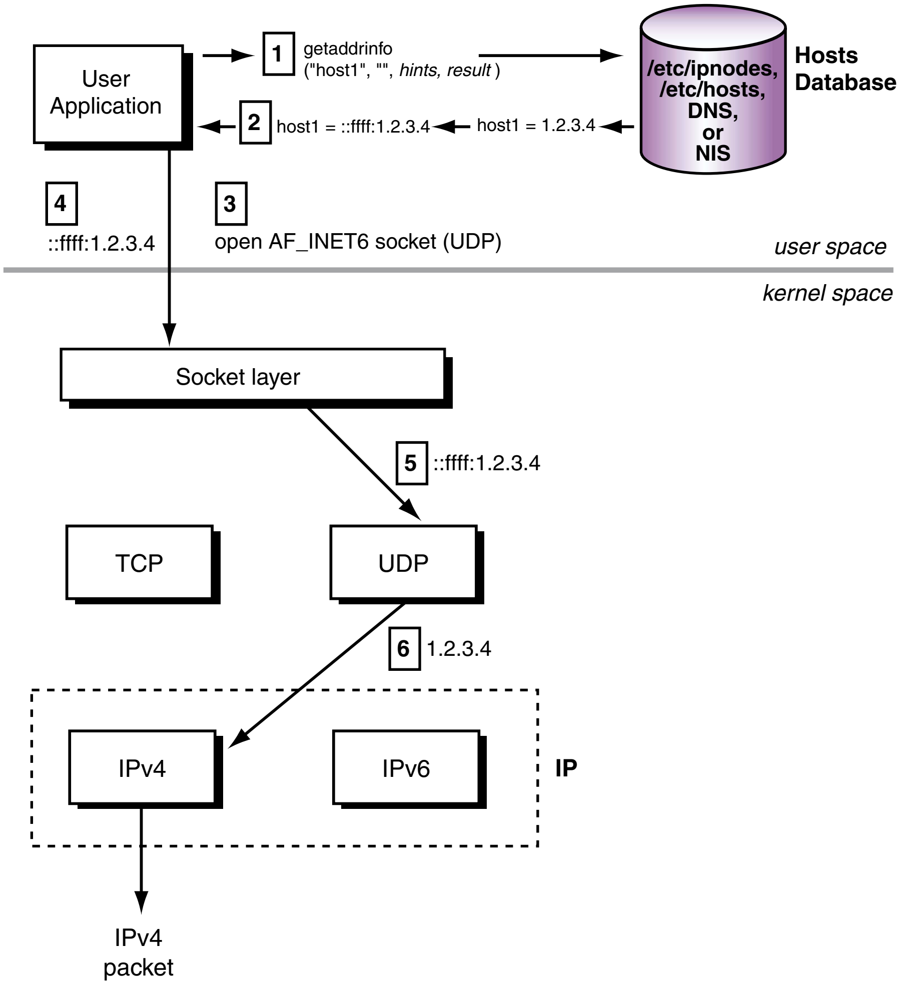
From this point, the application can do the following:

1. Call `recv` to wait for a response from the server system.
2. After it receives a response, call `getpeername` to determine the address of the connected socket. The address is returned in a structure of type `sockaddr_in6`. If you want your application to be protocol independent, you would use the `sockaddr_storage` structure instead of the `sockaddr_in6` structure.
3. Call `getnameinfo` with the `NI_NAMEREQD` flag to obtain the server name.
4. Call `getnameinfo` with the `NI_NUMERICHOST` flag to convert the server address to a text string.

Section C.2.1 contains sample client program code that demonstrates these steps.

You can also use an AF_INET6 socket for IPv4 communications. Figure 4–4 shows the sequence of events for an client application that uses an AF_INET6 socket to send IPv4 packets.

Figure 4–4: Using AF_INET6 Socket for IPv4 Communications (Send)



ZK-1557U-AI

1 Application calls getaddrinfo and passes the host name (host1), the AF_INET6 address family hint, and the AI_ADDRCONFIG and AI_V4MAPPED flag hints. The flag hints tell the function that if an IPv4 address is found for host1, return it as an IPv4–mapped IPv6 address. See getaddrinfo(3) for a description of hints fields and values.

2 The search finds an IPv4 address, 1.2.3.4, for host1 in the hosts database and getaddrinfo returns the IPv4–mapped IPv6 address ::ffff:1.2.3.4 in one or more structures of type addrinfo.

3 The application calls socket to create an AF_INET6 socket, using the address family and socket type contained in the addrinfo structure. The socket is a datagram socket (UDP) in this example, but could be a stream socket (TCP).

4 If the socket call is successful, the application calls connect to establish a connection to host1, using the host address and length in the addrinfo structure. If the connect call is successful, the application sends information to the ::ffff:1.2.3.4 address.

Note

After using the information in the addrinfo structures, the application calls freeaddrinfo to free system resouces used by the structures.

5 The socket layer passes the information and address to the UDP module.

6 The UDP module identifies the IPv4–mapped IPv6 address and puts the 1.2.3.4 address into the packet header and passes the information to the IPv4 module for transmission.

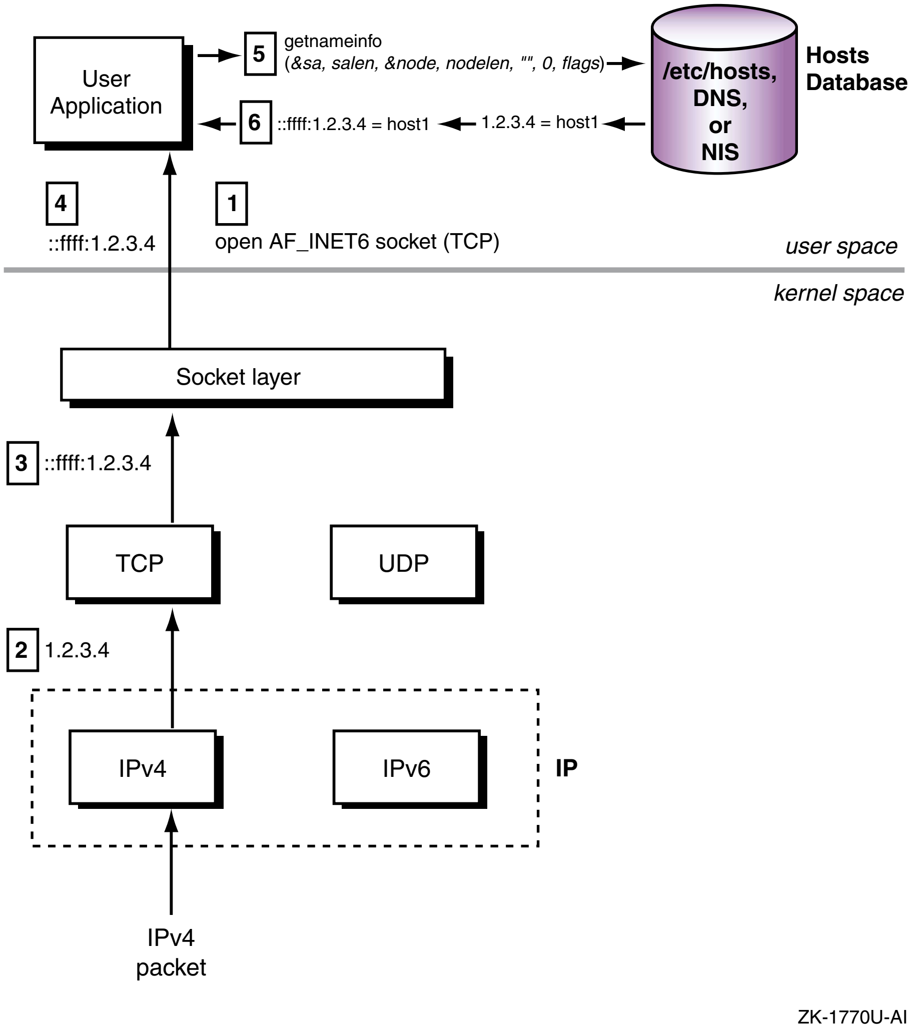
From this point, the application can do the following:

1. Call recv to wait for a response from the server system.
2. After it receives a response, call getpeername to determine the address of the connected socket. The address is returned in a structure of type sockaddr_in6. If you want your application to be protocol independent, you would use the sockaddr_storage structure instead of the sockaddr_in6 structure.
3. Call getnameinfo with the NI_NAMEREQD flag to obtain the server name.
4. Call getnameinfo with the NI_NUMERICHOST flag to convert the server address to a text string.

Section C.2.1 contains sample client program code that demonstrates these steps.

AF_INET6 sockets can receive messages sent to either IPv4 or IPv6 addresses. An AF_INET6 socket uses the IPv4–mapped IPv6 address format to represent IPv4 addresses. Figure 4–5 shows the sequence of events for a server application that uses an `AF_INET6` socket to receive IPv4 packets.

Figure 4–5: Using AF_INET6 Socket for IPv4 Communications (Receive)



1 The application does the following:

1. Calls socket to create an AF_INET6 socket.
2. Initializes a sockaddr_in6 structure and sets the family, address, and port.
3. Calls bind to assign an address to the socket.
4. Calls listen to mark the socket as accepting connections.

2 An IPv4 packet arrives and passes through the IPv4 module.

3 The TCP layer strips off the packet header and passes the information and the IPv4–mapped IPv6 address (::ffff:1.2.3.4) to the socket layer.

4 The application calls accept and retrieves the information from the socket. The information from the socket is passed to the application in a sockaddr_storage structure. This enables the application to be protocol independent.

5 The application calls getnameinfo and passes the ::ffff:1.2.3.4 address and the NI_NAMEREQD flag. The flag tells the function to return the host name for the address. See getnameinfo(3) for a description of the flags bits and their meanings.

6 The search finds the host name for the 1.2.3.4 address in the hosts database and getnameinfo returns the host name.

Section C.2.2 contains sample server program code that demonstrates these steps.

4.4.3 Address Testing Macros

In some cases, an application that uses AF_INET6 socket for communications might need to determine the type of address that is returned in the structure. For this case, the API defines macros to test the addresses. Table 4–5 lists the currently defined address testing macros and the return value for a valid test. To use these macros, include the following file in your application:

```
#include <netinet/in.h>
```

Table 4–5: Summary of Address Testing Macros

Macro	Return
IN6_IS_ADDR_UNSPECIFIED	True, if specified type.
IN6_IS_ADDR_LOOPBACK	True, if specified type.
IN6_IS_ADDR_MULTICAST	True, if specified type.
IN6_IS_ADDR_LINKLOCAL	True, if specified type.

Table 4–5: Summary of Address Testing Macros (cont.)

IN6_IS_ADDR_SITELOCAL	True, if specified type.
IN6_IS_ADDR_V4MAPPED	True, if specified type.
IN6_IS_ADDR_V4COMPAT	True, if specified type.
IN6_IS_ADDR_MC_NODELOCAL	True, if specified scope.
IN6_IS_ADDR_MC_LINKLOCAL	True, if specified scope.
IN6_IS_ADDR_MC_SITELOCAL	True, if specified scope.
IN6_IS_ADDR_MC_ORGLOCAL	True, if specified scope.
IN6_IS_ADDR_MC_GLOBAL	True, if specified scope.
IN6_ARE_ADDR_EQUAL	True, if addresses are equal.

4.5 BSD Socket Interface

In addition to the XNS4.0 socket interface, the operating system also supports the 4.3BSD, 4.4BSD, and POSIX 1003.1g Draft 6.6 socket interfaces. The 4.4BSD socket interface provides a number of changes to 4.3BSD sockets. Most of the changes between the 4.3BSD and 4.4BSD socket interfaces were designed to facilitate the implementation of International Standards Organization (ISO) protocol suites under the sockets framework. The XNS4.0 socket interface provides a standard version of the socket interface.

Note

The availability of the 4.4BSD socket interface does not mean that your site supports ISO protocols. Check with the appropriate personnel at your site.

To use the 4.4BSD socket interface, you must add the following line to your program or makefile:

```
#define _SOCKADDR_LEN
```

The 4.4BSD socket interface includes the following changes from the 4.3BSD interface for application programs:

- A `sockaddr` structure for supporting variable-length (long) network addresses
- A `msghdr` structure to allow receipt of protocol information and status with data

The following sections describe these features.

4.5.1 Variable-Length Network Addresses

The 4.4BSD version of the sockaddr structure supports variable-length network addresses. The structure adds a length field and is defined as follows:

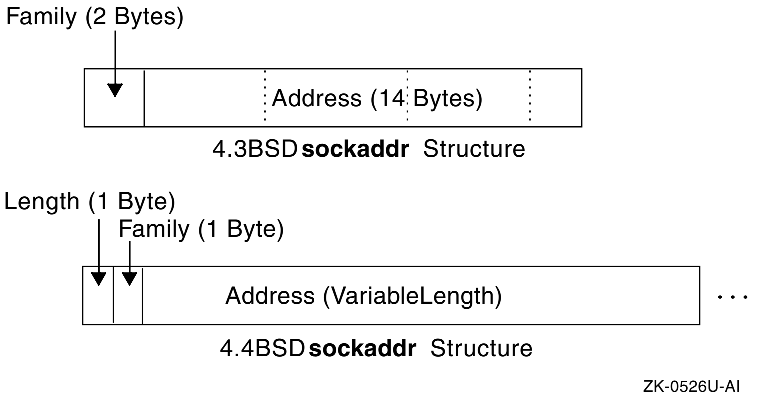
```
/* 4.4BSD sockaddr Structure */

struct sockaddr {
        u_char sa_len;      /* total length */
        u_char sa_family;   /* address family */
        char   sa_data[14]; /* actually longer; address value */
};
```

The 4.3BSD sockaddr structure contains the following fields:

```
u_short sa_family;
char    sa_data[14];
```

Figure 4–6 compares the 4.3BSD and 4.4BSD sockaddr structures.

Figure 4–6: 4.3BSD and 4.4BSD sockaddr Structures



4.5.2 Receiving Protocol Data with User Data

The 4.3BSD version of the msghdr structure (which is the default if you use the cc command) provides the parameters needed for using the optional functions of the sendmsg and recvmsg system calls.

The 4.3BSD msghdr structure is as follows:

```
/* 4.3BSD msghdr Structure */
struct msghdr {
        caddr_t         msg_name;           /* optional address */
        int             msg_namelen;        /* size of address */
        struct iovec   *msg_iov;            /* scatter/gather array */
        int             msg_iovlen;         /* # elements in msg_iov */
        caddr_t         msg_accrights;      /* access rights sent/re-
                                            /* ceived */
        int             msg_accrightslen;
};
```

The *msg_name* and *msg_namelen* parameters are used when the socket is not connected. The *msg_iov* and *msg_iovlen* parameters are used for scatter (read) and gather (write) operations. As stated previously, the *msg_accrights* and *msg_accrightslen* parameters allow the sending process to pass its access rights to the receiving process.

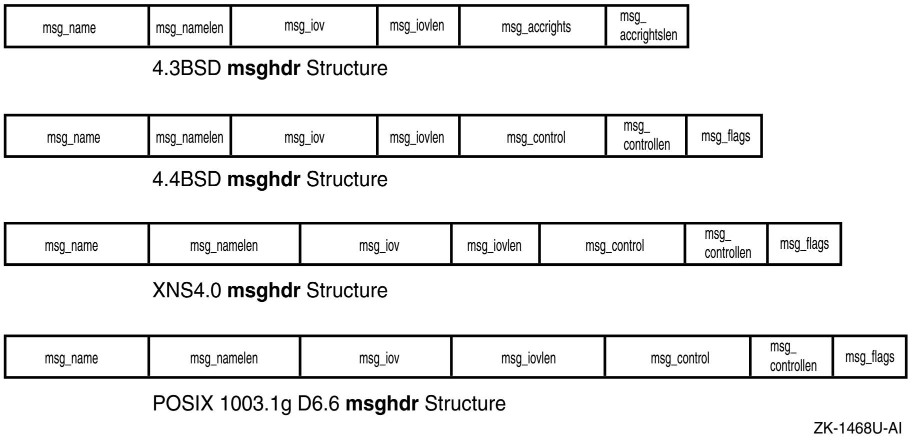
The 4.4BSD structure has additional fields that permit application programs to include protocol information along with user data in messages.

To support the receipt of protocol data together with user data, the operating system provides the `msghdr` structure from the 4.4BSD socket interface. The structure adds a pointer to control data, a length field for the length of the control data, and a flags field, as follows:

```
 /* 4.4BSD msghdr Structure */
struct msghdr {
        caddr_t        msg_name;       /* optional address */
        u_int          msg_namelen;    /* size of address */
        struct iovec  *msg_iov;        /* scatter/gather array */
        u_int          msg_iovlen;     /* # elements in msg_iov */
        caddr_t        msg_control;    /* ancillary data, see below */
        u_int          msg_controllen; /* ancillary data buffer len */
        int            msg_flags;      /* flags on received message */
};
```

The XNS4.0 and POSIX 1003.1g Draft 6.6 `msghdr` data structures have the same fields as 4.4BSD. However, the size of the `msg_namelen` and `msg_controllen` fields are 8 bytes in the XNS4.0 and POSIX 1003.1g Draft 6.6 `msghdr` data structures, as opposed to 4 bytes in the 4.4BSD `msghdr` data structure. In addition, the size of the `msg_iovlen` field is 8 bytes in the POSIX 1003.1g Draft 6.6 `msghdr` data structure, as opposed to 4 bytes long in the 4.4BSD and XNS4.0 `msghdr` data structures. Figure 4–7 shows the 4.3BSD, 4.4BSD, XPG4, and POSIX 1003.1g Draft 6.6 `msghdr` structures.

Figure 4–7: 4.3BSD, 4.4BSD, XNS4.0, and POSIX 1003.1g msghdr Structures



4.3BSD **msghdr** Structure

4.4BSD **msghdr** Structure

XNS4.0 **msghdr** Structure

POSIX 1003.1g D6.6 **msghdr** Structure

In the 4.3BSD version of the `msghdr` data structure, the *`msg_accrights`* and *`msg_accrightslen`* fields permit the sending process to pass its access rights to a system-maintained object, in this case a socket, to the receiving process. In the 4.4BSD, XNS4.0, and POSIX 1003.1g Draft 6.6 versions, this is done using the *`msg_control`* and *`msg_controllen`* fields.

4.6 Common Socket Errors

Table 4–6 lists some common socket error messages and the problems they indicate:

Table 4–6: Common Errors and Diagnostics

Error	Diagnostics
`[EAFNOSUPPORT]`	The protocol family does not support the addresses in the specified address family.
`[EBADF]`	The socket parameter is not valid.
`[ECONNREFUSED]`	The attempt to connect was rejected.
`[EFAULT]`	A pointer does not point to a valid part of user address space.
`[EHOSTDOWN]`	The host is down.
`[EHOSTUNREACH]`	The host is unreachable.
`[EINVAL]`	An invalid argument was used.
`[EMFILE]`	The current process has too many open file descriptors
`[ENETDOWN]`	The network is down.

Table 4–6: Common Errors and Diagnostics (cont.)

Error	Diagnostics
[ENETUNREACH]	The network is unreachable. No route to the network is present.
[ENOMEM]	The system was unable to allocate kernel memory to increase the process descriptor table.
[ENOTSOCK]	The socket parameter refers to a file, not a socket.
[EOPNOTSUPP]	The specified protocol does not permit creation of socket pairs.
[EOPNOTSUPP]	The referenced socket can not accept connections.
[EPROTONOSUPPORT]	This system does not support the specified protocol.
[EPROTOTYPE]	The socket type does not support the specified protocol.
[ETIMEDOUT]	The connection timed out without a response from the remote application.
[EWOULDBLOCK]	The socket is marked nonblocking and the operation could not complete.

4.7 Advanced Topics

This section contains the following information, which is of interest to developers writing complex applications for sockets:

- Porting applications to use AF_INET6 sockets
- Selecting specific protocols
- Binding names and addresses
- Out-of-band data
- IP Multicasting
- Broadcasting and determining network configuration
- The `inetd` daemon
- Input/output multiplexing
- Interrupt-driven socket I/O
- Signals and process groups
- Pseudoterminals

4.7.1 Porting Applications to Use AF_INET6 Sockets

AF_INET6 sockets enable applications to communicate using the IPv6 protocol, IPv4 protocol, or both. For IPv6 communication, RFC 2553, *Basic Socket Interface Extensions for IPv6*, specifies changes to the BSD socket Applications Programming Interface (API). Table 4–7 summarizes these changes.

Table 4–7: Summary of IPv6 Extensions to the BSD Socket API

Category	Changes
Core function calls	None; basic syntax of socket functions stays the same. Applications must cast pointers to the protocol-specific address structures into pointers to the generic `sockaddr` address structure when using the socket functions. See Section 4.4 for information on creating Internet applications.
Socket address structure	Specifies a new `sockaddr_in6` structure for IPv6 communications and a `sockaddr_storage` structure for protocol-independent communication. The `sockaddr_in` structure remains for IPv4 communications. See Section 4.7.1.2 for more information.
Name-to-address translation	Specifies the `getnameinfo` and `getaddrinfo` functions for protocol-independent (IPv4 and IPv6) communication. The `gethostbyaddr` and `gethostbyname` functions remain for IPv4 communications only. See Section 4.7.1.3 for more information.
Address conversion functions	Specifies the `getnameinfo` and `getaddrinfo` functions for protocol-independent (IPv4 and IPv6) address conversion. The `inet_ntoa` and `inet_addr` functions remain for IPv4 address conversion only. See Section 4.7.1.3 for more information.
Socket options	Specifies new socket options for IPv6 multicast. See Section 4.7.5 for more information.

This section describes the following changes that you must make in your existing application code in order to be ready to operate in an IPv6 networking environment. You must also know how an Internet application operates over AF_INET6 sockets. See Section 4.4.2 for more information.

- Name changes
- Structure changes
- Function call changes
- Other application changes

If your application needs to interoperate with existing IPv4 applications, this section also includes guidelines for making changes to your code. After you make these changes, your ported application will be capable of communicating over both IPv4 and IPv6. Any existing IPv4 applications will continue to operate as before, and interoperate with your IPv6 application.

4.7.1.1 Making Name Changes

Most of the changes required are straightforward and mechanical, some may require a bit of code restructuring (for example, a routine that returns an `int` datatype holding an IPv4 address may need to be modified to take as an extra parameter a pointer to an `in6_addr` into which it writes the IPv6 address). Table 4–8 summarizes the changes to make to your application's code:

Table 4–8: Name Changes

Search file for:	Replace with:
`AF_INET`	`AF_INET6`
`PF_INET`	`PF_INET6`
`INADDR_ANY`	`in6addr_any`

4.7.1.2 Making Structure Changes

The structure names and field names have changed for the following structures:

- `in_addr`
- `sockaddr_in`
- `sockaddr`
- `hostent`

The following sections discuss these changes.

in_addr Structure

Applications that use the `in_addr` structure must be changed, as needed, to use the `in6_addr` structure, as shown in the following examples:

AF_INET Structure

```
struct in_addr[1]
  unsigned int s_addr[2]
```

AF_INET6 Structure

```
struct in6_addr[1]
  uint8_t s6_addr [16][2]
```

Make the following changes in your application, as needed:

1 Change the structure name `in_addr` to `in6_addr`.

2 Change the data type from `unsigned int` to `uint8_t` and the field name `s_addr` to `s6_addr`.

If you use the `in6_addr` structure, see Section 4.7.1.4 for additional changes might need to make to your application.

sockaddr_in Structure

Applications that use the 4.4 BSD `sockaddr_in` structure must be changed, as needed, to use the `sockaddr_in6` structure, as shown in the following examples:

AF_INET Structure

```
struct sockaddr_in 1
  unsigned char sin_len 2
  sa_family_t sin_family 3
  in_port_t sin_port 4
  struct in_addr sin_addr 5
```

AF_INET6 Structure

```
struct sockaddr_in6 1
  uint8_t sin6_len 2
  sa_family_t sin6_family 3
  int_port_t sin6_port 4
  struct in6_addr sin6_addr 5
```

Make the following change in your application, as needed:

1 Change structure name `sockaddr_in` to `sockaddr_in6`.

2 Change the data type `unsigned char` to `uint8_t` and the field name `sin_len` to `sin6_len`.

3 Change the field name `sin_family` to `sin6_family`.

4 Change the field name `sin_port` to `sin6_port`.

5 Change the field name `sin_addr` to `sin6_addr`.

Applications that use the 4.3 BSD `sockaddr_in` structure must be changed, as needed, to use the `sockaddr_in6` structure, as shown in the following examples:

AF_INET Structure

```
struct sockaddr_in 1
  u_short sin_family 2
  in_port_t sin_port 3
  struct in_addr sin_addr 4
```

AF_INET6 Structure

```
struct sockaddr_in6 1
  u_short sin6_family 2
  in_port_t sin6_port 3
  struct in6_addr sin6_addr 4
```

Make the following change in your application, as needed:

1 Change structure name `sockaddr_in` to `sockaddr_in6`.

2 Change the field name `sin_family` to `sin6_family`.

3 Change the field name `sin_port` to `sin6_port`.

4 Change the field name `sin_addr` to `sin6_addr`.

Note

In both cases, you should initialize the entire `sockaddr_in6` structure to zero after your structure declarations.

sockaddr Structure

Applications that use the generic socket address structure (`sockaddr`) to hold an AF_INET socket address (`sockaddr_in`) must be changed to use either the AF_INET6 `sockaddr_in6` structure or the `sockaddr_storage` structure.

If your application is to manipulate IPv6 addresses only, make the following changes in your application, as needed:

AF_INET Structure	AF_INET6 Structure
`struct sockaddr` 1	`struct sockaddr_in6` 1

1 Change structure name `sockaddr` to `sockaddr_in6` for those cases where it should be `struct sockaddr_in`. For example, `sizeof(struct sockaddr)`.

If your application is to manipulate addresses from both IPv4 and IPv6 nodes, make the following changes in your application, as needed:

AF_INET Structure	AF_INET6 Structure
`struct sockaddr` 1	`struct sockaddr_storage` 1

1 Change structure name `sockaddr` to `sockaddr_storage` for those cases where it should be `struct sockaddr_in`. For example, `sizeof(struct sockaddr)`.

Note

Both the `sockaddr_in6` and `sockaddr_storage` structures are larger than a `sockaddr` structure.

hostent Structure

Applications that use the `hostent` structure must be changed, as needed, to use the `addrinfo` structure, as shown in the following examples:

AF_INET Structure

```
struct hostent[1]
```

AF_INET6 Structure

```
struct addrinfo[1]
```

Make the following changes in your application, as needed:

[1] Change the structure name `hostent` to `addrinfo`.

4.7.1.3 Making Function Call Changes

You must make changes, as needed, to applications that use the following library routines:

- `gethostbyaddr`
- `gethostbyname`
- `inet_ntoa`
- `inet_addr`

The following sections discuss these changes.

gethostbyaddr Function Call

Applications that use the `gethostbyaddr` function call must be changed to use the `getnameinfo` function call, as shown in the following examples:

AF_INET Call

```
gethostbyaddr(xxx,4,AF_INET)[1]
```

AF_INET6 Call

```
err=getnameinfo(&sockaddr,sockaddr_len, node_name, name_len,
 service, service_len, flags);[1]
```

Make the following changes in your application, as needed:

[1] Change the function name from `gethostbyaddr` to `getnameinfo` and provide a pointer to the socket address structure, a character string for the returned node name, an integer for the length of the returned node name, a character string to receive the returned service name, an integer for the length of the returned service name, and an integer that specifies the type of address processing to be performed. See `getnameinfo`(3) for a description of the flags bits and their meanings.

gethostbyname Function Call

Applications that use the `gethostbyname` function call must be changed to use the `getaddrinfo` function call, as shown in the following examples:

AF_INET Call

```
gethostbyname(name) 1
```

AF_INET6 Call

```
err=getaddrinfo(node_name, service_name, &hints, &result); 1
 .
 .
 .
freeaddrinfo(&result); 2
```

Make the following changes in your application, as needed:

1. Change the function name from `gethostbyname` to `getaddrinfo` and provide a character string that contains the node name, a character string that contains the service name to use, a pointer to a `hints` structure that contains processing options, and a pointer to an `addrinfo` structure or structures for the returned address information. See `getaddrinfo`(3) for a description of hints fields and values.
2. Add a call to the `freeaddrinfo` routine to free the `addrinfo` structure or structures when your application is finished using them.

If your application needs to determine whether an address is an IPv4 address or an IPv6 address, and cannot determine this from the address family, use the `IN6_IS_ADDR_V4MAPPED` macro. See Section 4.4.3 for a list of IPv6 address testing macros.

inet_ntoa Function Call

Applications that use the `inet_ntoa` function call must be changed to use the `getnameinfo` function call, as shown in the following examples:

AF_INET Call

```
inet_ntoa(addr) 1
```

AF_INET6 Call

```
err=getnameinfo(&sockaddr,sockaddr_len, node_name, name_len,
 service, service_len, NI_NUMERICHOST); 1
```

Make the following changes in your application, as needed:

1. Change the function name from `gethostbyaddr` to `getnameinfo` and provide a pointer to the socket address structure, a character string for the returned node name, an integer for the length of the returned node name, a character string to receive the returned service name, an integer for the length of the returned service name, and the NI_NUMERICHOST flag. See `getnameinfo`(3) for a description of the flags bits and their meanings.

inet_addr Function Call

Applications that use the inet_addr function call must be changed to use the getaddrinfo function call, as shown in the following examples:

AF_INET Call

```
result=inet_addr(&string);[1]
```

AF_INET6 Call

```
err=getaddrinfo(node_name, service_name, &hints, &result);[1]
.
.
.
freeaddrinfo(&result);[2]
```

Make the following changes in your application, as needed:

[1] Change the function name from inet_addr to getaddrinfo and provide a character string that contains the node name, a character string that contains the service name to use, a pointer to a hints structure that contains the AI_NUMERICHOST option, and a pointer to an addrinfo structure or structures for the returned address information. See getaddrinfo(3) for a description of hints fields and values.

[2] Add a call to the freeaddrinfo routine to free the addrinfo structure or structures when your application is finished using them.

If your application needs to determine whether an address is an IPv4 address or an IPv6 address, and cannot determine this from the address family, use the IN6_IS_ADDR_V4MAPPED macro. See Section 4.4.3 for a list of IPv6 address testing macros.

4.7.1.4 Making Other Application Changes

In addition to the name changes, you should review your code for specific uses of IP address information and variables.

Comparing IP Addresses

If your application compares IP addresses or tests IP addresses for equality, the in6_addr structure changes you made in Section 4.7.1.2 change the comparison of int quantities to a comparison of structures. This will break the code and cause compiler errors.

Make either of the following changes to your application, as needed:

AF_INET Code

```
(addr1->s_addr == addr2->s_addr)[1]
```

AF_INET6 Code

```
(memcmp(addr1, addr2, sizeof(struct in6_addr)) == 0) [1]
```

[1] Change the equality expression to one that uses the `memcmp` (memory comparison) function.

AF_INET Code

```
(addr1->s_addr == addr2->s_addr) [1]
```

AF_INET6 Code

```
IN6_ARE_ADDR_EQUAL(addr1, addr2) [1]
```

[1] Change the equality expression to one that uses the `IN6_ARE_ADDR_EQUAL` macro. See Section 4.4.3 for a list of IPv6 address testing macros.

Comparing an IP Address to the Wildcard Address

If your application compares an IP address to the wildcard address, the `in6_addr` structure changes you made in Section 4.7.1.2 change the comparison of `int` quantities to a comparison of structures. This will break the code and cause compiler errors.

Make either of the following changes to your application, as needed:

AF_INET Code

```
(addr->s_addr == INADDR_ANY) [1]
```

AF_INET6 Code

```
IN6_IS_ADDR_UNSPECIFIED(addr) [1]
```

[1] Change the equality expression to one that uses the `IN6_IS_ADDR_UNSPECIFIED` macro. See Section 4.4.3 for a list of IPv6 address testing macros.

AF_INET Code

```
(addr->s_addr == INADDR_ANY) [1]
```

AF_INET6 Code

```
(memcmp(addr, in6addr_any, sizeof(struct in6_addr)) == 0) [1]
```

[1] Change the equality expression to one that uses the `memcmp` (memory comparison) function.

Using int Data Types to Hold IP Addresses

If your application uses `int` data types to hold IP addresses, the `in6_addr` structure changes you made in Section 4.7.1.2 changes the assignment. This will break the code and cause compiler errors.

Make the following changes to your application, as needed:

AF_INET Code

```
struct in_addr foo;
int bar; 1
   .
   .
   .
bar = foo.s_addr; 2
```

AF_INET6 Code

```
struct in6_addr foo;
struct in6_addr bar; 1
   .
   .
   .
bar = foo; 2
```

1 Change the data type for `bar` from `int` to a `struct in6_addr`.

2 Change the assignment statement for `bar` to remove the `s_addr` field reference.

Using Functions that Return IP Addresses

If your application uses functions that return IP addresses as `int` data types, the `in6_addr` structure changes you made in Section 4.7.1.2 changes the destination of the return value from an `int` to an array of `char`. This will break the code and cause compiler errors.

Make the following changes to your application, as needed:

AF_INET Code

```
struct in_addr *addr;
addr->s_addr = foo(xxx); 1
```

AF_INET6 Code

```
struct in6_addr *addr;
foo(xxx, addr); 1
```

1 Restructure the function to enable you to pass the address of the structure in the call. In addition, modify the function to write the return value into the structure pointed to by *addr*.

Changing Socket Options

If your application uses IPv4 IP-level socket options, change them to the corresponding IPv6 options. See `ip(7)` for more information.

4.7.2 Selecting Specific Protocols

The syntax of the `socket` system call is described in Section 4.3.1. If the third argument to the `socket` call, the *protocol* argument, is zero (0), the `socket` call selects a default protocol to use with the returned socket descriptor. The default protocol is usually correct and alternate choices are not usually available. However, when using raw sockets to communicate directly with lower-level protocols or hardware interfaces, the protocol argument can be important for setting up demultiplexing.

For example, raw sockets in the Internet family can be used to implement a new protocol above IP and the socket receives packets only for the protocol specified. To obtain a particular protocol, you must determine the protocol number as defined within the communication domain. For the Internet domain, you can use one of the library routines described in Section 4.2.3.2.

The following code shows how to use the `getprotobyname` library call to select the protocol `newtcp` for a SOCK_STREAM socket opened in the Internet domain:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
⋮

struct protent *pp;
⋮

pp = getprotobyname("newtcp");
s = socket(AF_INET, SOCK_STREAM, pp->p_proto);
```

4.7.3 Binding Names and Addresses

The `bind` system call associates an address with a socket.

4.7.3.1 Binding to the Wildcard Address

The local machine address for a socket can be any valid network address of the machine. Because one system can have several valid network addresses, binding addresses to sockets in the Internet domain can be complicated. To simplify local address binding, the constant INADDR_ANY (AF_INET) and in6addr_any (AF_INET6), wildcard addresses, are provided. The wildcard address tells the system that this server process will accept a connection on any of its Internet interfaces, if it has more than one.

The following example shows how to bind the wildcard value INADDR_ANY to a local socket:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>

main()
{
   int s, length;
   struct sockaddr_in name;
   char buf[1024];
⋮

   /* Create name with wildcards. */
   name.sin_family = AF_INET;
   name.sin_len = sizeof(name);
   name.sin_addr.s_addr = INADDR_ANY;
   name.sin_port = 0;
   if (bind(s, (struct sockaddr *)&name, sizeof(name))== -1) {
      perror("binding datagram socket");
      exit(1);
   }
 .
 .
 .
}
```

Sockets with wildcard local addresses can receive messages directed to the specified port number, and send to any of the possible addresses assigned to that host. Note that the socket uses a wildcard value for its local address; a process sending messages to the named socket must specify a valid network address. A process can be willing to receive a message from anywhere, but it cannot send a message anywhere.

An AF_INET socket can only receive messages addressed to an IPv4 address on the system. However, AF_INET6 sockets can receive messages sent to either IPv4 or IPv6 addresses on the system. An AF_INET6 socket uses the IPv4–mapped IPv6 address format to represent IPv4 addresses. See the *Network Administration: Connections* manual for information about IPv6 addresses.

When a server process on a system with more than one network interface wants to allow hosts to connect to only one of its interface addresses, the server process binds the address of the appropriate interface. For example, if a system has two addresses 130.180.123.45 and 131.185.67.89, a server process can bind the address 130.180.123.45. Binding that address ensures that only connections addressed to 130.180.123.45 can connect to the server process.

Similarly, a local port can be left as unspecified (specified as zero), in which case the system selects a port number for it.

4.7.3.2 Binding in the UNIX Domain

Processes that communicate in the UNIX domain (AF_UNIX) are bound by an association that local and foreign pathnames comprises. UNIX domain sockets do not have to be bound to a name but, when bound, there can never be duplicate bindings of a protocol, local pathname, or foreign pathname. The pathnames cannot refer to files existing on the system. The process that binds the name to the socket must have write permission on the directory where the bound socket will reside.

The following example shows how to bind the name socket to a socket created in the UNIX domain:

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

#define NAME "socket"

main()
{
   int s, length;
   struct sockaddr_un name;
   char buf[1024];
⋮

   /* Create name. */
   name.sun_len = sizeof(name.sun_len) +
    sizeof(name.sun_family) +
    strlen(NAME);
   name.sun_family = AF_UNIX;
   strcpy(name.sun_path, NAME);
   if (bind(s, (struct sockaddr *) &name, sizeof(name))==-1) {
      perror("binding name to datagram socket");
      exit(1);
   }
⋮

}
```

4.7.4 Out-of-Band Data

Out-of-band data is a logically independent transmission channel associated with each pair of connected stream sockets. Out-of-band data can be delivered to the socket independently of the normal receive queue or within the receive queue, depending on the status of the SO_OOBINLINE option, set with the setsockopt system call.

The stream socket abstraction specifies that the out-of-band data facilities must support the reliable delivery of at least one out-of-band message at a time. This message must contain at least one byte of data and at least one message can be pending delivery to the user at any one time.

The socket layer supports marks in the data stream that indicate the end of urgent data or out-of-band processing. The socket mechanism does not return data from both sides of a mark in a single system call.

You can use MSG_PEEK to peek at out-of-band data. If the socket has a process group, a SIGURG signal is generated when the protocol is notified of its existence. A process can set the process group or process ID to be informed by the SIGURG signal via the appropriate fcntl call, as described in Section 4.7.9 for SIGIO.

When multiple sockets have out-of-band data awaiting delivery, an application program can use a select call for exceptional conditions to determine which sockets have such data pending. The SIGURG signal or select call notifies the application program that data is pending. The application then must issue the appropriate call actually to receive the data.

In addition to the information passed, a logical mark is placed in the data stream to indicate the point at which the out-of-band data was sent. When a signal flushes any pending output, all data up to the logical mark in the data stream is discarded.

To send an out-of-band message, the MSG_OOB flag is supplied to a send or a sendto system call. To receive out-of-band data, an application program must set the MSG_OOB flag when performing a recvfrom or recv system call.

An application program can determine if the read pointer is currently pointing to the mark in the data stream by using the the SIOCATMARK ioctl:

```
ioctl(s, SIOCATMARK, &yes);
```

If yes is a 1 on return, meaning that no out-of-band data arrived, the next read returns data after the mark. If out-of-band data did arrive, the next read provides data sent by the client prior to transmission of the out-of-band signal. The following program shows the routine used in the remote login process to flush output on receipt of an interrupt or quit signal. This program reads the normal data up to the mark (to discard it), then reads the out-of-band byte:

```
#include <sys/ioctl.h>
#include <sys/file.h>
   .
   .
   .
oob()
{
   int out = FWRITE, mark;
   char waste[BUFSIZ];

   /* flush local terminal output */
   ioctl(1, TIOCFLUSH, (char *)&out);
   for (;;) {
      if (ioctl(rem, SIOCATMARK, &mark) < 0) {
         perror("ioctl");
         break;
      }
      if (mark)
         break;
      (void) read(rem, waste, sizeof (waste));
   }
   if (recv(rem, &mark, 1, MSG_OOB) < 0) {
      perror("recv");
⋮

   }
}
```

A process can also read or peek at the out-of-band data without first reading up to the logical mark. This is difficult when the underlying protocol delivers the urgent in-band data with the normal data and only sends notification of its presence ahead of time; for example, the TCP protocol. With such protocols, when the out-of-band byte has not yet arrived and a `recv` system call is done with the MSG_OOB flag, the call returns an EWOULDBLOCK error. There can be enough in-band data in the input buffer so that normal flow control prevents the peer from sending the urgent data until the buffer is cleared. The process must then read enough of the queued data so that the urgent data can be delivered.

Note

Certain programs that use multiple bytes of urgent data and must handle multiple urgent signals need to retain the position of urgent data within the stream. The socket-level SO_OOBINLINE option provides this capability and it is strongly recommended that you use it.

The socket-level SO_OOBINLINE option retains the position of the urgent data (the logical mark). The urgent data immediately follows the mark within the normal data stream that is returned without the MSG_OOB flag. Reception of multiple urgent indications causes the mark to move, but no out-of-band data is lost.

4.7.5 Internet Protocol Multicasting

Internet Protocol (IP) multicasting provides applications with IP layer access to the multicast capability of Ethernet and Fiber Distribution Data Interface (FDDI) networks. IP multicasting, which delivers datagrams on a best-effort basis, avoids the overhead imposed by IP broadcasting (described in Section 4.7.6) on uninterested hosts; it also avoids consumption of network bandwidth by applications that would otherwise transmit separate packets with identical data to reach several destinations.

IPv4 multicasting achieves efficient multipoint delivery through use of **multicast groups**. A multicast group is a group of zero or more nodes that is identified by a single Class D IP destination address (IPv4) or a single multicast address (IPv6). An IPv4 Class D address has 1110 in the four high-order bits. In dotted decimal notation, IP multicast addresses range from 224.0.0.0 to 239.255.255.255, with 224.0.0.0 being reserved. An IPv6 multicast address has the format prefix of FF00::/8.

A member of a particular multicast group receives a copy of all data sent to the IP address representing that multicast group. Multicast groups can be permanent or transient. A permanent group has a well-known, administratively assigned IP address. In permanent multicast groups, it is the address of the group that is permanent, not its membership. The number of group members can fluctuate, even dropping to zero.

In IPv4, the `all hosts` group is an example of a permanent host group whose assigned address is 224.0.0.1. Tru64 UNIX systems join the all hosts group to participate in the Internet Group Management Protocol (IGMP). (See RFC 1112: *Host Extensions for IP Multicasting* for more information about IGMP and IP multicasting.)

In IPv6, the All Nodes multicast address is an example of a permanent group whose addresses are FF01::1 (node-local, or scope 1) and FF02::1 (link-local, or scope2). See RFC 1884: *IPv6 Addressing Architecture* for more information about IPv6 multicast addresses.

IP addresses that are not reserved for permanent multicast groups are available for dynamic assignment to transient groups. Transient groups exist only as long as they have one or more members.

Note

IP multicasting is not supported over connection-oriented transports such as TCP.

IP multicasting is implemented using options to the `setsockopt` system call, described in the following sections. Definitions required for multicast-related socket options are in the `<netinet/in.h>` header file. Your application must include this header file if you intend it to receive IP multicast datagrams.

4.7.5.1 Sending IPv4 Multicast Datagrams

To send IPv4 multicast datagrams, an application indicates the host group to send to by specifying an IP destination address in the range of 224.0.0.0 to 239.255.255.255 in a `sendto` system call. The system maps the specified IP destination address to the appropriate Ethernet or FDDI multicast address prior to transmitting the datagram.

An application can explicitly control multicast options with arguments to the `setsockopt` system call. The following options can be set by an application using the `setsockopt` system call:

- Time-to-live field (`IP_MULTICAST_TTL`)
- Multicast interface (`IP_MULTICAST_IF`)
- Disabling loopback of local delivery (`IP_MULTICAST_LOOP`)

Note

> The syntax for and arguments to the `setsockopt` system call are described in Section 4.3.5 and `setsockopt`(2). The examples here and in Section 4.7.5.2 illustrate how to use the `setsockopt` options that apply to IPv4 multicast datagrams only.

The `IP_MULTICAST_TTL` option to the `setsockopt` system call allows an application to specify a value between 0 and 255 for the time-to-live (TTL) field. Multicast datagrams with a TTL value of 0 restrict distribution of the multicast datagram to applications running on the local host. Multicast datagrams with a TTL value of 1 are forwarded only to hosts on the local subnet. If a multicast datagram has a TTL value greater than 1 and a multicast router is attached to the sending host's network, then multicast datagrams can be forwarded beyond the local subnet. Multicast routers forward the datagram to known networks that have hosts belonging to the specified multicast group. The TTL value is decremented by each multicast router in the path. When the TTL value is decremented to 0, the datagram is not forwarded further.

The following example shows how to use the `IP_MULTICAST_TTL` option to the `setsockopt` system call:

```
u_char ttl;
ttl=2;

if (setsockopt(sock, IPPROTO_IP, IP_MULTICAST_TTL, &ttl,
                                    sizeof(ttl)) == -1)
       perror("setsockopt");
```

A datagram addressed to an IP multicast destination is transmitted from the default network interface unless the application specifies that an alternate network interface is associated with the socket. The default interface is determined by the interface associated with the default route in the kernel routing table or by the interface associated with an explicit route, if one exists. Using the IP_MULTICAST_IF option to the setsockopt system call, an application can specify a network interface other than that specified by the route in the kernel routing table.

The following example shows how to use the IP_MULTICAST_IF option to the setsockopt system call to specify an interface other than the default:

```
int sock;
struct in_addr ifaddress;
char *if_to_use = "16.141.64.251";
.
.
.

  ifaddress.s_addr = inet_addr(if_to_use);
  if (setsockopt(sock, IPPROTO_IP, IP_MULTICAST_IF, &ifaddress,
                              sizeof(ifaddress)) == -1)
     perror ("error from setsockopt IP_MULTICAST_IF");
  else
     printf ("new interface set for sending multicast datagrams\n");
```

If a multicast datagram is sent to a group of which the sending host is a member, a copy of the datagram is, by default, looped back by the IP layer for local delivery. The IP_MULTICAST_LOOP option to the setsockopt system call allows an application to disable this loopback delivery.

The following example shows how to use the IP_MULTICAST_LOOP option to the setsockopt system call:

```
u_char loop=0;
if (setsockopt( sock, IPPROTO_IP, IP_MULTICAST_LOOP, &loop
         sizeof(loop)) == -1)
 perror("setsockopt");
```

When the value of *loop* is 0, loopback is disabled. When the value of *loop* is 1, it is enabled. For performance reasons, you should disable the default, unless applications on the same host must receive copies of the datagrams.

4.7.5.2 Receiving IPv4 Multicast Datagrams

Before a host can receive IP multicast datagrams destined for a particular multicast group other than the all hosts group, an application must direct the host to become a member of that multicast group. This section describes how an application can direct a host to add itself to and remove itself from a multicast group.

An application can direct the host it is running on to join a multicast group by using the IP_ADD_MEMBERSHIP option to the setsockopt system call as follows:

```
struct ip_mreq mreq;
if (setsockopt( sock, IPPROTO_IP, IP_ADD_MULTICAST, &mreq
         sizeof(mreq)) == -1)
         perror("setsockopt");
```

The *mreq* variable has the following structure:

```
structip_mreq{
struct in_addr imr_multiaddr; /* IP multicast address of group */
struct in_addr imr_interface; /* local IP address of interface */
};
```

Each multicast group membership is associated with a particular interface. It is possible to join the same group on multiple interfaces. The *imr_interface* variable can be specified as `INADDR_ANY`, which allows an application to choose the default multicast interface. Alternatively, specifying one of the host's local addresses allows an application to select a particular, multicast-capable interface. The maximum number of memberships that can be added on a single socket is subject to the `IP_MAX_MEMBERSHIPS` value, which is defined in the `<netinet/in.h>` header file.

To drop membership in a particular multicast group use the `IP_DROP_MEMBERSHIP` option to the `setsockopt` system call:

```
struct ip_mreq mreq;
if (setsockopt( sock, IPPROTO_IP, IP_DROP_MEMBERSHIP, &mreq
         sizeof(mreq))== -1)
 perror("setsockopt");
```

The *mreq* variable contains the same structure values used for adding membership.

If multiple sockets request that a host join a particular multicast group, the host remains a member of that multicast group until the last of those sockets is closed.

To receive multicast datagrams sent to a specific UDP port, the receiving socket must have bound to that port using the `bind` system call. More than one process can receive UDP datagrams destined for the same port if the `bind` system call (described in Section 4.3.2) is preceded by a `setsockopt` system call that specifies the `SO_REUSEPORT` option. The following example illustrates how to use the `SO_REUSEPORT` option to the `setsockopt` system call:

```
int setreuse = 1;
if (setsockopt(sock, SOL_SOCKET, SO_REUSEPORT, &setreuse,
                                sizeof(setreuse)) == -1)
         perror("setsockopt");
```

When the `SO_REUSEPORT` option is set, every incoming multicast or broadcast UDP datagram destined for the shared port is delivered to all sockets bound to that port.

Delivery of IP multicast datagrams to SOCK_RAW sockets is determined by the protocol type of the destination.

4.7.5.3 Sending IPv6 Multicast Datagrams

To send IPv6 multicast datagrams, an application indicates the multicast group to send to by specifying an IPv6 multicast address in a `sendto` system call. The system maps the specified IPv6 destination address to the appropriate Ethernet or FDDI multicast address prior to transmitting the datagram.

An application can explicitly control multicast options with arguments to the `setsockopt` system call. The following options can be set by an application using the `setsockopt` system call:

- Hop limit (`IPV6_MULTICAST_HOPS`)
- Multicast interface (`IPV6_MULTICAST_IF`)
- Disabling loopback of local delivery (`IPV6_MULTICAST_LOOP`)

Note

> The syntax for and arguments to the `setsockopt` system call are described in Section 4.3.5 and `setsockopt`(2). The examples here and in Section 4.7.5.4 illustrate how to use the `setsockopt` options that apply to IPv6 multicast datagrams only.

The `IPV6_MULTICAST_HOPS` option to the `setsockopt` system call allows an application to specify a value between 0 and 255 for the hop limit field. Multicast datagrams with a hop limit value of 0 restrict distribution of the multicast datagram to applications running on the local host. Multicast datagrams with a hop limit value of 1 are forwarded only to hosts on the local link. If a multicast datagram has a hop limit value greater than 1 and a multicast router is attached to the sending host's network, multicast datagrams can be forwarded beyond the local link. Multicast routers forward the datagram to known networks that have hosts belonging to the specified multicast group. The hop limit value is decremented by each multicast router in the path. When the hop limit value is decremented to 0, the datagram is not forwarded further.

The following example shows how to use the `IPV6_MULTICAST_HOPS` option to the `setsockopt` system call:

```
u_char hops;
hops=2;

if (setsockopt(sock, IPPROTO_IPV6, IPV6_MULTICAST_HOPS, &hops,
                                   sizeof(hops)) < 0)
```

```
        perror("setsockopt: IPV6_MULTICAST_HOPS error");
```

A datagram addressed to an IPv6 multicast address is transmitted from the default network interface unless the application specifies that an alternate network interface is associated with the socket. The default interface is determined by the interface associated with the default route in the kernel routing table or by the interface associated with an explicit route, if one exists. Using the IPV6_MULTICAST_IF option to the setsockopt system call, an application can specify a network interface other than that specified by the route in the kernel routing table.

The following example shows how to use the IPV6_MULTICAST_IF option to the setsockopt system call to specify an interface other than the default:

```
u_int if_index = 1;
 .
 .
 .
if (setsockopt(sock, IPPROTO_IPV6, IPV6_MULTICAST_IF, &if_index,
                           sizeof(if_index)) < 0)
   perror ("setsockopt: IPV6_MULTICAST_IF error");
else
   printf ("new interface set for sending multicast datagrams\n");
```

The if_index parameter specifies the interface index of the desired interface or 0 to select a default interface. You can use the if_nametoindex(3) routine to find the interface index.

If a multicast datagram is sent to a group of which the sending node is a member, a copy of the datagram is, by default, looped back by the IP layer for local delivery. The IPV6_MULTICAST_LOOP option to the setsockopt system call allows an application to disable this loopback delivery.

The following example shows how to use the IPV6_MULTICAST_LOOP option to the setsockopt system call:

```
u_char loop=0;
if (setsockopt( sock, IPPROTO_IPV6, IPV6_MULTICAST_LOOP, &loop,
          sizeof(loop)) < 0)
          perror("setsockopt: IPV6_MULTICAST_LOOP error");
```

When the value of *loop* is 0, loopback is disabled. When the value of *loop* is 1, it is enabled. For performance reasons, you should disable the default, unless applications on the same host must receive copies of the datagrams.

4.7.5.4 Receiving IPv6 Multicast Datagrams

Before a node can receive IPv6 multicast datagrams destined for a particular multicast group other than the All Nodes group, an application must direct the node to become a member of that multicast group. This section describes how an application can direct a node to add itself to and remove itself from a multicast group.

An application can direct the node it is running on to join a multicast group by using the IPV6_JOIN_GROUP option to the setsockopt system call as follows:

```
struct ipv6_mreq imr6;
.
.
.

imr6.ipv6mr_interface = if_index;
if (setsockopt( sock, IPPROTO_IPV6, IPV6_JOIN_GROUP,
     (char *)&imr6, sizeof(imr6)) < 0)
     perror("setsockopt: IPV6_JOIN_GROUP error");
```

The *imr6* variable has the following structure:

```
structipv6_mreq {
struct in6_addr ipv6mr_multiaddr; /*IP multicast address of
group*/
unsigned int ipv6mr_interface; /*local interface index*/
};
```

Each multicast group membership is associated with a particular interface. It is possible to join the same group on multiple interfaces. The *ipv6mr_interface* variable can be specified with a value of 0, which allows an application to choose the default multicast interface. Alternatively, specifying one of the host's local interfaces allows an application to select a particular, multicast-capable interface. The maximum number of memberships that can be added on a single socket is subject to the IPV6_MAX_MEMBERSHIPS value, which is defined in the <netinet/in.h> header file.

To drop membership in a particular multicast group use the IPV6_LEAVE_GROUP option to the setsockopt system call:

```
struct ipv6_mreq imr6;
if (setsockopt( sock, IPPROTO_IPV6, IPV6_LEAVE_GROUP, &imr6,
     sizeof(imr6)) < 0)
     perror("setsockopt: IPV6_LEAVE_GROUP error");
```

The *imr6* parameter contains the same structure values used for adding membership.

If multiple sockets request that a node join a particular multicast group, the node remains a member of that multicast group until the last of those sockets is closed.

To receive multicast datagrams sent to a specific UDP port, the receiving socket must have bound to that port using the bind system call. More than one process can receive UDP datagrams destined for the same port if the bind system call (described in Section 4.3.2) is preceded by a setsockopt system call that specifies the SO_REUSEPORT option. See Section 4.7.5.2 for an example of using this option.

Delivery of IP multicast datagrams to SOCK_RAW sockets is determined by the protocol type of the destination.

4.7.6 Broadcasting and Determining Network Configuration

Using a datagram socket, it is possible to send broadcast packets on many networks supported by the system. The network itself must support broadcast; the system provides no simulation of broadcast in the software.

Broadcast messages can place a high load on a network because they force every host on the network to service them. Consequently, the ability to send broadcast packets is limited to sockets that are explicitly marked as allowing broadcasting.

Broadcast is typically used for one of two reasons: to find a resource on a local network without prior knowledge of its address, or to route some information, which requires that information be sent to all accessible neighbors.

Note

Broadcasting is not supported over connection-oriented transports such as TCP.

To send a broadcast message, use the following procedure:

1. Create a datagram socket; for example:

   ```
   s = socket(AF_INET, SOCK_DGRAM, 0);
   ```

2. Mark the socket for broadcasting; for example:

   ```
   int   on = 1;

   if (setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on,
                  sizeof(on)) == -1)
       perror("setsockopt");
   ```

3. Ensure that at least a port number is bound to the socket; for example:

   ```
   sin.sin_len = sizeof(sin);
   sin.sin_family = AF_INET;
   sin.sin_addr.s_addr = htonl(INADDR_ANY);
   sin.sin_port = htons(MYPORT);
   if (bind(s, (struct sockaddr *) &sin, sizeof (sin)) == -1)
       perror("setsockopt");
   ```

The destination address of the message depends on the network or networks on which the message is to be broadcast. The Internet domain supports a shorthand notation for broadcast on the local network, the address is INADDR_BROADCAST (as defined in netinet/in.h).

To determine the list of addresses for all reachable neighbors requires knowledge of the networks to which the host is connected. The operating system provides a method of retrieving this information from the system data structures. The SIOCGIFCONF ioctl call returns the interface configuration of a host in the form of a single ifconf structure. This structure contains a data area that an array of ifreq structures comprises, one for each network interface to which the host is connected. These structures are defined in the <net/if.h> header file, as follows:

```
struct  ifconf {
   int     ifc_len;                /* size of associated buffer */
   union {
      caddr_t ifcu_buf;
      struct  ifreq *ifcu_req;
   } ifc_ifcu;
#define ifc_buf ifc_ifcu.ifcu_buf  /* buffer address */
#define ifc_req ifc_ifcu.ifcu_req  /* array of structures returned */
};

struct  ifreq {
#define IFNAMSIZ        16
   char    ifr_name[IFNAMSIZ];     /* if name, e.g. "en0" */
   union {
      struct  sockaddr ifru_addr;
      struct  sockaddr ifru_dstaddr;
      struct  sockaddr ifru_broadaddr;
      short   ifru_flags;
      int     ifru_metric;
      caddr_t ifru_data;
   } ifr_ifru;
#define ifr_addr      ifr_ifru.ifru_addr      /* address */
#define ifr_dstaddr   ifr_ifru.ifru_dstaddr   /* other end of */
                                              /* p-to-p link */
#define ifr_broadaddr ifr_ifru.ifru_broadaddr /* broadcast address */
#define ifr_flags     ifr_ifru.ifru_flags     /* flags */
#define ifr_metric    ifr_ifru.ifru_metric    /* metric */
#define ifr_data      ifr_ifru.ifru_data      /* for use by */
                                              /* interface */
};
```

The actual call which obtains the interface configuration is as follows:

```
struct ifconf ifc;
char buf[BUFSIZ];

ifc.ifc_len = sizeof (buf);
ifc.ifc_buf = buf;
if (ioctl(s, SIOCGIFCONF, (char *) &ifc) < 0) {
⋮

}
```

After this call, buf contains one ifreq structure for each network to which the host is connected, and ifc.ifc_len is modified to reflect the number of bytes used by the ifreq structures.

Each structure has a set of interface flags that tells whether the network corresponding to that interface flag is up or down, point-to-point or broadcast, and so on. The SIOCGIFFLAGS ioctl retrieves these flags for an interface specified by an ifreq structure, as follows:

```
struct ifreq *ifr;

ifr = ifc.ifc_req;

for (n = ifc.ifc_len / sizeof (struct ifreq); --n >= 0; ifr++) {
   /*
    * We must be careful that we don’t use an interface
    * devoted to an address family other than those intended.
    */
   if (ifr->ifr_addr.sa_family != AF_INET)
      continue;
   if (ioctl(s, SIOCGIFFLAGS, (char *) ifr) < 0) {
.
.
.
   }
   /*
    * Skip irrelevant cases.
    */
   if ((ifr->ifr_flags & IFF_UP) == 0 ||
       (ifr->ifr_flags & IFF_LOOPBACK) ||
       (ifr->ifr_flags & (IFF_BROADCAST | IFF_POINTOPOINT)) == 0)
      continue;
```

Once the flags are obtained, the broadcast address must be obtained. In the case of broadcast networks, this is done via the SIOCGIFBRDADDR ioctl; while, for point-to-point networks, the address of the destination host is obtained with SIOCGIFDSTADDR. For example:

```
struct sockaddr dst;

if (ifr->ifr_flags & IFF_POINTOPOINT) {
   if (ioctl(s, SIOCGIFDSTADDR, (char *) ifr) < 0) {
      ...
   }
   bcopy((char *) ifr->ifr_dstaddr, (char *) &dst,
      sizeof (ifr->ifr_dstaddr));
} else if (ifr->ifr_flags & IFF_BROADCAST) {
   if (ioctl(s, SIOCGIFBRDADDR, (char *) ifr) < 0) {
      ...
   }
   bcopy((char *) ifr->ifr_broadaddr, (char *) &dst,
      sizeof (ifr->ifr_broadaddr));
}
```

After the appropriate ioctl calls obtain the broadcast or destination address (now in dst), the sendto call is used; for example:

```
if (sendto(s, buf, buflen, 0, (struct sockaddr *)&dst, sizeof (dst)) < 0)
    perror("sendto");
```

In the preceding loop, one sendto call occurs for every interface to which the host is connected that supports the notion of broadcast or point-to-point addressing. If a process only wants to send broadcast messages on a given network, code similar to that in the preceding example is used, but the loop needs to find the correct destination address.

4.7.7 The inetd Daemon

The operating system supports the inetd Internet superserver daemon. The inetd daemon, which is invoked at boot time, reads the /etc/inetd.conf file to determine the servers for which it should listen.

Note

Only server applications written to run over sockets can use the inetd daemon in Tru64 UNIX. The inetd daemon in Tru64 UNIX does not support server applications written to run over STREAMS, XTI, or TLI.

For each server listed in /etc/inetd.conf the inetd daemon does the following:

1. Creates a socket and binds the appropriate port number to it.
2. Issues a select system call for read availability and waits for a process to request a connection to the service that corresponds to that socket.
3. Issues an accept system call, forks, duplicates (with the dup call) the new socket to file descriptors 0 and 1 (stdin and stdout), closes other open file descriptors, and executes (with the exec call) the appropriate server.

Servers that use inetd are simplified because inetd takes care of most of the interprocess communication work required to establish a connection. The server invoked by inetd expects the socket connected to its client on file descriptors 0 and 1, and immediately performs any operations such as read, write, send, or recv.

Servers invoked by the inetd daemon can use buffered I/O as provided by the conventions in the <stdio.h> header file, as long as as they remember to use the fflush call when appropriate. See fflush(3) for more information.

The getpeername call, which returns the address of the peer (process) connected on the other end of the socket, is useful for developers writing server applications that use inetd. The following sample code shows how to log the Internet address, in dot notation, of a client connected to a server under `inetd`:

```
struct sockaddr_storage name;
size_t namelen = sizeof (name);
.
.
.
if (getpeername(0, (struct sockaddr *)&name, &namelen) < 0) {
  syslog(LOG_ERR, "getpeername: %m");
  exit(1);
} else
  syslog(LOG_INFO, "Connection from %s", inet_ntoa(name.sin_addr));
.
.
.
```

While the `getpeername` call is especially useful when writing programs to run with `inetd`, it can be used under other circumstances.

4.7.8 Input/Output Multiplexing

Multiplexing is a facility used in applications to transmit and receive I/O requests among multiple sockets. This can be done by using the `select` call, as follows:

```
#include <sys/time.h>
#include <sys/types.h>
.
.
.
fd_set readmask, writemask, exceptmask;
struct timeval timeout;
.
.
.
if (select(nfds, &readmask, &writemask, &exceptmask, &timeout) < 0)
 perror("select");
```

The `select` call takes as arguments pointers to three sets:

1. The set of socket descriptors for which the calling application wants to read data.
2. The socket descriptors to which data is to be written.
3. Exceptional conditions which are pending.

   The corresponding argument to the `select` call must be a null pointer, if the application is not interested in certain conditions; for example, read, write, or exceptions.

Note

Because XTI and TLI are implemented using STREAMS, you should use the `poll` system call instead of the `select` system call on any STREAMS file descriptors.

Each set is actually a structure that contains an array of integer bit masks. The size of the array is set by the FD_SETSIZE definition. The array is long enough to hold one bit for each of the FD_SETSIZE file descriptors.

The FD_SET (*fd, &mask*) and FD_CLR (*fd, &mask*) macros are provided to add and remove the *fd* file descriptor in the *mask* set. The set needs to be zeroed before use and the FD_ZERO (*&mask*) macro is provided to clear the *mask* set.

The *nfds* parameter in the select call specifies the range of file descriptors (for example, one plus the value of the largest descriptor) to be examined in a set.

A time-out value can be specified when the selection will not last more than a predetermined period of time. If the fields in timeout are set to zero (0), the selection takes the form of a poll, returning immediately. If the last parameter is a null pointer, the selection blocks indefinitely. Specifically, a return takes place only when a descriptor is selectable or when a signal is received by the caller, interrupting the system call.

The select call normally returns the number of file descriptors selected; if the select call returns because the time-out expired, then the value 0 is returned. If the select call terminates because of an error or interruption, a -1 is returned with the error number in errno and with the file descriptor masks unchanged.

Assuming a successful return, the three sets indicate which file descriptors are ready to be read from, written to, or have exceptional conditions pending. The status of a file descriptor in a select mask can be tested with the FD_ISSET (*fd, &mask*) macro, which returns a nonzero value if *fd* is a member of the *mask* set or 0 if it is not.

To determine whether there are connections waiting on a socket to be used with an accept call, the select call is used, followed by a FD_ISSET (*fd, &mask*) macro to check for read readiness on the appropriate socket. If FD_ISSET returns a nonzero value, indicating data to read, then a connection is pending on the socket.

Note

In 4.2BSD, the arguments to the select call were pointers to integers instead of pointers to fd_set. This type of call works as long as the number of file descriptors being examined is less than the number of bits in an integer; however, the method shown in the following code is recommended.

The following example shows how an application reads data as it becomes available from sockets s1 and s2 with a 1-second time-out:

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

fd_set read_template;
struct timeval wait;
⋮

for (;;) {
   wait.tv_sec = 1;   /* one second */
   wait.tv_usec = 0;

   FD_ZERO(&read_template);

   FD_SET(s1, &read_template);
   FD_SET(s2, &read_template);

   nb = select(FD_SETSIZE, &read_template, (fd_set *) 0,
      (fd_set *) 0, &wait);
   if (nb <= 0) {
      An error occurred during the select, or
         the select timed out   }

   if (FD_ISSET(s1, &read_template)) {
      Socket #1 is ready to be read from.
   }

   if (FD_ISSET(s2, &read_template)) {
      Socket #2 is ready to be read from.
   }
}
```

The `select` call provides a synchronous multiplexing scheme. Asynchronous notification of output completion, input availability, and exceptional conditions is possible through use of the SIGIO and SIGURG signals described in Section 4.7.10.

4.7.9 Interrupt Driven Socket I/O

The SIGIO signal allows a process to be notified using a signal when a socket (or more generally, a file descriptor) has data waiting to be read. Using the SIGIO facility requires the following three steps:

1. The process must set up a SIGIO signal handler by using the `signal` or `sigvec` calls.
2. The process must set the process ID or process group ID that is to receive notification of pending input to its own process ID or the process

group ID of its process group. (Note that the default process group of a socket is group 0.) This is done by using a fcntl system call.

3. The process must enable asynchronous notification of pending I/O requests with another fcntl system call. The following code shows how to allow a particular process to receive information on pending I/O requests as they occur for socket s. With the addition of a handler for SIGURG, this code can also be used to prepare for receipt of SIGURG signals.

```
    #include <fcntl.h>
⋮

    int   io_handler();
⋮

    signal(SIGIO, io_handler);

    /* Set the process receiving SIGIO/SIGURG signals to us */

    if (fcntl(s, F_SETOWN, getpid()) < 0) {
       perror("fcntl F_SETOWN");
       exit(1);
    }

    /* Allow receipt of asynchronous I/O signals */

    if (fcntl(s, F_SETFL, FASYNC) < 0) {
       perror("fcntl F_SETFL, FASYNC");
       exit(1);
    }
```

4.7.10 Signals and Process Groups

Each socket has an associated process number, the value of which is initialized to zero (0). This number must be redefined with the F_SETOWN parameter to the fcntl system call, as was done in Section 4.7.9, to enable SIGURG and SIGIO signals to be caught. To set the socket's process ID for signals, positive arguments must be given to the fcntl call. To set the socket's process group for signals, negative arguments must be passed to the fcntl call. Note that the process number indicates the associated process ID or the associated process group; it is impossible to specify both simultaneously.

The F_GETOWN parameter to the fcntl call allows a process to determine the current process number of a socket.

The SIGCHLD signal is also useful when constructing server processes. This signal is delivered to a process when any child processes change state.

Typically, servers use the SIGCHLD signal to reap child processes that exited, without explicitly awaiting their termination or periodic polling for exit status. If the parent server process fails to reap its children, a large number of zombie processes may be created. The following code shows how to use the SIGCHLD signal:

```
int reaper();
⋮

signal(SIGCHLD, reaper);
listen(f, 5);
for (;;) {
   int g;
   size_t len = sizeof (from);

   g = accept(f, (struct sockaddr *)&from, &len,);
   if (g < 0) {
      if (errno != EINTR)
         syslog(LOG_ERR, "rlogind: accept: %m");
      continue;
   }
⋮

}
⋮

#include <wait.h>
reaper()
{
   union wait status;

   while (wait3(&status, WNOHANG, 0) > 0)
      ;
}
```

4.7.11 Pseudoterminals

Many programs cannot function properly without a terminal for standard input and output. Since sockets do not provide the semantics of terminals, it is often necessary to have a process communicating over the network do so through a pseudoterminal (`pty`). A pseudoterminal is a pair of devices, master and slave, that allow a process to serve as an active agent in communication between applications and users.

Data written on the slave side of a pseudoterminal is used as input to a process reading from the master side, while data written on the master side is processed as terminal input for the slave. In this way, the process manipulating the master side of the pseudoterminal controls the information read and written on the slave side as if it were manipulating the keyboard and reading the screen on a real terminal. The purpose of the pseudoterminal abstraction is to preserve terminal semantics over a network connection; that is, the slave side appears as a normal terminal to any process reading from or writing to it.

For example, `rlogind`, the remote login server uses pseudoterminals for remote login sessions. A user logging in to a machine across the network is provided a shell with a slave pseudoterminal as standard input, standard output, and standard error. The server process then handles the communication between the programs invoked by the remote shell and the user's local client process. When a user sends a character that generates an interrupt on the remote machine that flushes terminal output, the pseudoterminal generates a control message for the server process. The server then sends an out-of-band message to the client process to signal a flush of data at the real terminal and on the intervening data buffered in the network.

In the operating system, the slave side of a pseudoterminal has a name of the form `/dev/tty`*xy*, where *x* is any single letter, except `d`, and is uppercase or lowercase. The *y* is a hexadecimal digit, meaning it is a single character in the range of 0 to 9 or a to f. The master side of a pseudoterminal has a name of the form `/dev/pty`*xy*, where *x* and *y* correspond to *x* and *y* on the slave side of the pseudoterminal.

The `openpty` and `forkpty` functions were added to the `libc.a` library to make allocating pseudoterminals easier. These functions use the `clone open` call to avoid performing multiple `open` calls.

The `forkpty` function allocates a pseudoterminal. Additionally, it forks a child process and makes the slave pseudoterminal the controlling terminal for the child. The `forkpty` function takes four arguments instead of five, because the slave file descriptor is not passed back to the calling process. Instead, the slave file descriptor is duplicated in the newly created child process as `stdin`, `stdout`, and `stderr`. The other four arguments are identical to those of the `openpty` function.

Both the `openpty` and `forkpty` functions return -1 to signify an error condition. The `openpty` function returns a zero (0) upon sucessful completion, while the `forkpty` returns the pid of the child process. See `openpty`(3) for function syntax, parameters, and errors.

The `openpty` function works as follows:

1. Upon successful completion, the slave side of the pseudoterminal is set to the proper terminal modes. At the time the master and slave sides of the pseudoterminal are opened, the operating system performs the necessary security checks.

2. The process then forks; the child closes the master side of the pseudoterminal and executes (with the `exec` call) the appropriate program.
3. The parent closes the slave side of the pseudoterminal and begins reading and writing from the master side.

The following example makes use of pseudoterminal. The code in this example makes the following assumptions:

- A connection on a socket already exists.
- The socket is connected to a peer that wants a service of some kind.
- The process disassociated itself from any previous controlling terminal.

```
if (openpty(&mast,&slave,NULL,NULL,NULL) {
   syslog(LOG_ERR, "All network ports in use");
   exit(1);
}
ioctl(slave, TIOCGETA, &term);  /* get default slave termios struct */
term.c_iflag |= ICRNL;
term.c_oflag |= OCRNL;
ioctl(slave, TIOCSETA, &term);  /* set slave characteristics        */
i = fork();
if (i < 0) {
   syslog(LOG_ERR, "fork: %m");
   exit(1);
} else if (i) {      /* Parent */
   close(slave);
.
.
.
} else {       /* Child */
   (void) close(s);
   (void) close(master);
   dup2(slave, 0);
   dup2(slave, 1);
   dup2(slave, 2);
   if (slave > 2)
      (void) close(slave);
.
.
.
}
```

See Section 4.3 for information about using sockets.

5

Tru64 UNIX STREAMS

The operating system provides a STREAMS framework as specified by AT&T's System V, Version 4.0 release of STREAMS. This framework, which provides an alternative to traditional UNIX character input/output (I/O), allows you to implement I/O functions in a modular fashion. Modularly developed I/O functions allow applications to build and reconfigure communications services easily.

Note that STREAMS refers to the entire framework whereas Stream refers to the entity created by an application program with the `open` system call.

This chapter contains the following information:

- Overview of the STREAMS framework
- Description of the application interface to STREAMS
- Description of the kernel-level functions
- Instructions on how to configure modules or drivers
- Description of the Tru64 UNIX synchronization mechanism
- Information on how to create device special files
- Description of error and event logging
- Information about STREAMS reference pages

This chapter provides detailed information about areas where the Tru64 UNIX implementation of STREAMS differs from that of AT&T System V, Version 4.0. Where the Tru64 UNIX implementation does not differ significantly from that of AT&T, it provides pointers to the appropriate AT&T documentation.

Note that this chapter does not explain how to program using the STREAMS framework. For detailed programming information you should refer to the *Programmer's Guide: STREAMS*.

5.1 Overview of the STREAMS Framework

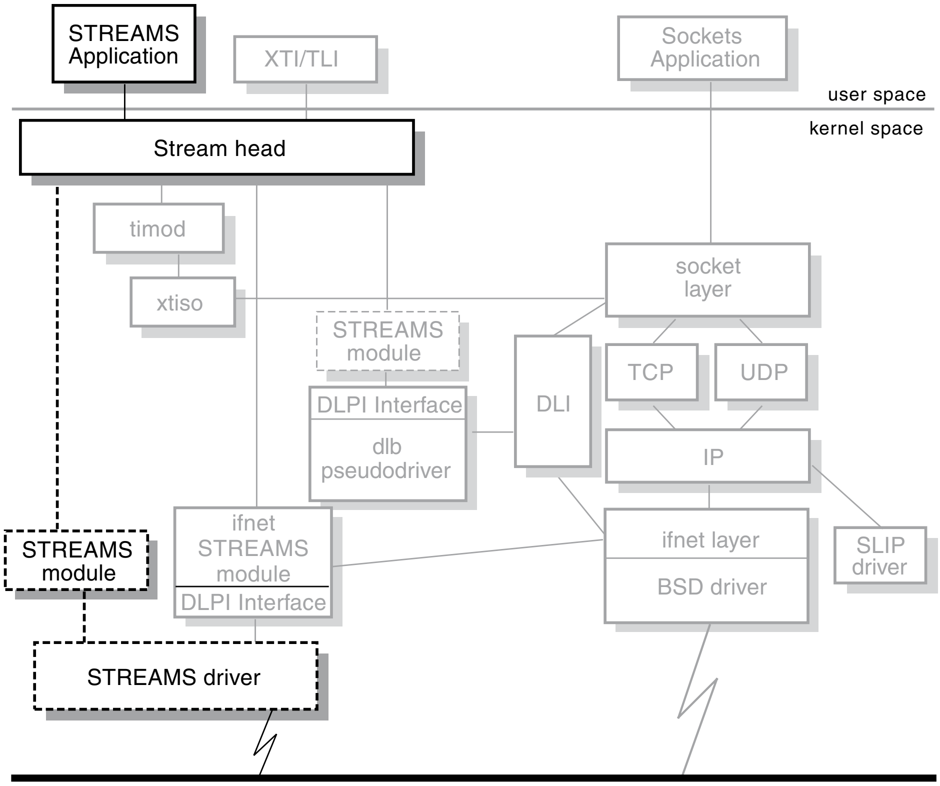
The STREAMS framework consists of:

- A programming interface, or set of system calls, used by application programs to access the STREAMS framework

- Kernel resources, such as the Stream head, and queue data structures used by the Stream
- Kernel utilities that handle tasks such as Stream queue scheduling and flow control, memory allocation, and error logging

Figure 5–1 highlights the STREAMS framework and shows its place in the network programming environment.

Figure 5–1: The STREAMS Framework



ZK-0559U-AI

5.1.1 Review of STREAMS Components

To communicate using STREAMS, an application creates a Stream, which is a full-duplex communication path between a user process and a device driver. The Stream itself is a kernel device and is represented to the application as a character special file. Like any other character special file, the Stream must be opened and otherwise manipulated with system calls.

Every Stream has at least a Stream head at the top and a Stream end at the bottom. Additional modules, which consist of linked pairs of queues, can be inserted between the Stream head and Stream end if they are required for processing the data being passed along the Stream. Data is passed between modules in messages.

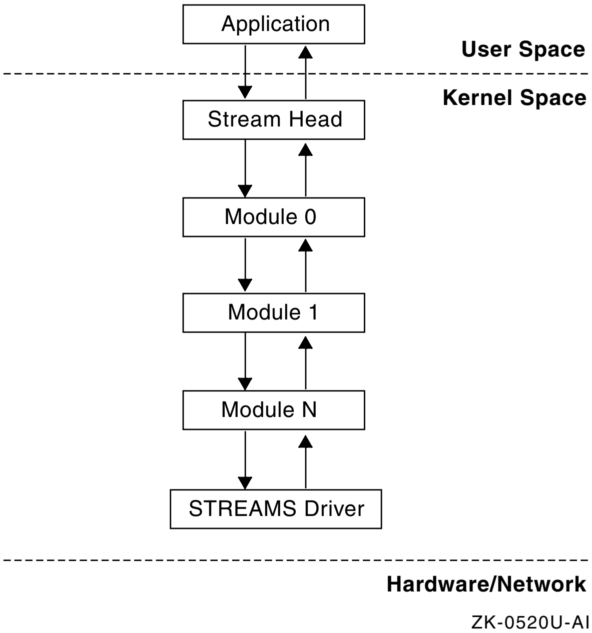
This section briefly describes the following STREAMS components:

- Stream head
- Stream end
- Modules

It also describes messages and their role in the STREAMS framework.

Figure 5–2 illustrates a typical stream. Note that data traveling from the Stream head to the Stream end (STREAMS driver in Figure 5–2) is said to be traveling downstream, or in the write direction. Data traveling from the Stream end to the Stream head is said to be traveling upstream, or in the read direction.

Figure 5–2: Example of a Stream



The Stream head is a set of routines and data structures that provides an interface between user processes and the Streams in the kernel. It is created when your application issues an `open` system call. The following are the major tasks that the Stream head performs:

1. Interprets a standard subset of STREAMS system calls, such as `write` and `putmsg`.

2. Translates them from user space into a standard range of STREAMS messages (such as M_PROTO and M_DATA) which consist of both data and control information.
3. Sends the messages downstream to the next module. Eventually the messages reach the Stream end, or driver.
4. Receives messages sent upstream from the driver and transforms the STREAMS message from kernel space to a format appropriate to the system call (such as `getmsg` or `read`) made by the application. The format varies depending on the system call.

The Stream end is a special form of STREAMS module and can be either a hardware or pseudodevice driver. If a hardware device driver, the Stream end provides communication between the kernel and an external communication device. If a pseudodevice driver, the Stream end is implemented in software and is not related to an external device. Regardless of whether it is a hardware device driver or a pseudodevice driver, the Stream end receives messages sent by the module above it, interprets them, and performs the requested operations. It then returns data and control information to the application by creating a message of the appropriate type which it sends upstream toward the Stream head.

Drivers are like any other STREAMS modules except for the following:

- They can handle interrupts (although they do not have to).

  Device drivers can have one or more interrupt routines. Interrupt routines should queue data on the read side service routine for later processing.

- They can be connected to multiple Streams.

  A driver can be implemented as a multiplexor, meaning that it is connected to multiple Streams in either the upstream or downstream direction. See the *Programmer's Guide: STREAMS* for more information.

- They are initialized and deinitialized by the `open` and `close` system calls. (Other modules use the `I_PUSH` and `I_POP` commands of the `ioctl` system call.)

For detailed information on device drivers and device driver routines, see the *Writing Device Drivers* manual and the *Programmer's Guide: STREAMS*.

Modules process data as it passes from the Stream head to the Stream end and back. A Stream can have zero or more modules on it, depending on the amount and type of processing that the data requires. If the driver can perform all of the necessary processing on the data, no additional modules are required.

Modules consist of a pair of queues that contain data and pointers to other structures that define what each module does. One queue handles data moving downstream toward the driver and the other handles data moving upstream toward the Stream head and application. Pointers link each module's downstream and upstream queues to the next module's downstream and upstream queues.

Depending on their processing requirements, applications request that particular modules be pushed onto the Stream. The Stream head assembles the modules requested by the application and then routes the messages through the pipeline of modules.

Information is passed from module to module using messages. Several different types of messages are defined within the STREAMS environment. All message types, however, fall into the following categories:

- Normal
- High priority

Normal messages, such as M_DATA and M_IOCTL, are processed in the order that they are received, and are subject to STREAMS flow control and queuing mechanisms. Priority messages are passed along the stream in an expedited manner.

For more information on messages and message data structures, see Section 5.3.2

5.1.2 ioctl Processing

In STREAMS, user processes can perform control functions on specific modules and drivers in a Stream with `ioctl` calls. When a user process issues an `ioctl` command, STREAMS blocks the process, forces the Stream head to process the command, and, if necessary, send messages (M_IOCTL) downstream to be received and processed by a specific module or driver. The user process is blocked until one of the following occurs:

- A module or a driver responds with a positive acknowledgement (M_IOCACK) or a negative acknowledgement (M_IOCNAK)
- The request times out (no message is received)
- The user process interrupts the `ioctl`
- An error condition occurs

STEAMS provides two methods of `ioctl` processing: I_STR and transparent.Table 5–1 compares the two methods.

Table 5–1: I_STR and Transparent ioctl Processing Comparison

I_STR Processing	Transparent Processing
Supports user applications written for STREAMS files only	Supports user applications written for either STREAMS or non-STREAMS filess
Commands described in `streamio`(7)	Commands described in `ioctl`(2)
Data format and addressing restrictions	Data format and addressing restrictions depend on the `ioctl`
Requires single pair of messages to complete `ioctl` processing	May require multiple pairs of messages to complete ioctl processing
Timeout value is user-specified or a default	Timeout value is infinite
Implies STREAMS processing	More general

See the *Programmer's Guide: STREAMS* for more information on both methods of `ioctl` processing.

5.2 Application Interface to STREAMS

The application interface to the STREAMS framework allows STREAMS messages to be sent and received by applications. The following sections describe the application interface, including pointers to the STREAMS header files and data types, and descriptions of the STREAMS and STREAMS-related system calls.

5.2.1 Header Files and Data Types

Definitions for the basic STREAMS data types are included in the following header files:

- The `<sys/stream.h>` header file must be included for all modules and Streams applications.
- The `<stropts.h>` header file must be included when an application uses the `ioctl` system call.
- The `<strlog.h>` header file must be included when an application uses the STREAMS error logger and trace facility.

Note

Typically, header file names are enclosed in angle brackets (< >). To obtain the absolute path to the header file, prepend `/usr/include/` to the information enclosed in the angle brackets. In the case of `<sys/stream.h>`, `stream.h` is located in the `/usr/include/sys` directory.

5.2.2 STREAMS Functions

Your application accesses and manipulates STREAMS kernel resources through the following functions:

- open
- close
- read
- write
- ioctl
- mkfifo
- pipe
- putmsg and putpmsg
- getmsg and getpmsg
- poll
- isastream
- fattach
- fdetach

This section briefly describes these functions. For detailed information about these functions, see the reference pages and the *Programmer's Guide: STREAMS*.

5.2.2.1 The open Function

Use the open function to open a Stream.

See open(2) for function syntax, parameters, and errors.

The following example shows how the open function is used:

```
int fd;
fd = open("/dev/streams/echo", O_RDWR);
```

5.2.2.2 The close Function

Use the close function to close a Stream.

See close(2) for function syntax, parameters, and errors.

The last close for a stream causes the stream associated with the file descriptor to be dismantled. Dismantling a stream includes popping any modules on the stream and closing the driver.

5.2.2.3 The read Function

Use the read function to receive the contents of M_DATA messages waiting at the Stream head.

See read(2) for function syntax, parameters, and errors.

The read function fails on message types other than M_DATA, and *errno* is set to EBADMSG.

5.2.2.4 The write Function

Use the write function to create one or more M_DATA messages from the data buffer.

See write(2) for function syntax, parameters, and errors.

5.2.2.5 The ioctl Function

Use the ioctl function to perform a variety of control functions on Streams.

See streamio(7) for function syntax, parameters, and errors for the STREAMS ioctl function.

The following example shows how the ioctl call is used:

```
int fd;
fd = open("/dev/streams/echo", O_RDWR, 0);
ioctl(fd,I_PUSH,"pass");
```

5.2.2.6 The mkfifo Function

Use the STREAMS-based mkfifo function to create a unidirectional STREAMS-based file descriptor.

Note

> The default version of the mkfifo function in the libc library is not STREAMS-based. To use the STREAMS version of the mkfifo function the application must link with the sys5 library. See mkfifo(2) for function syntax, parameters, and errors.
>
> Also note that the mkfifo function requires that the File on File Mount File System (FFM_FS) kernel option is configured. See the *System Administration* manual for information about configuring kernel options.

5.2.2.7 The pipe Function

Use the STREAMS-based pipe function to create a bidirectional, STREAMS-based, communication channel. Non-STREAMS pipes and STREAMS-based pipes differ in the following ways:

- Non-STREAMS pipes are unidirectional
- STREAMS operations (such as streamio and putmsg) can not be performed on them

Note

The default version of the pipe function in the libc library is not STREAMS-based. To use the STREAMS version of the pipe function the application must link with the sys5 library. See pipe(2) for function syntax, parameters, and errors.

5.2.2.8 The putmsg and putpmsg Functions

Use the putmsg and putpmsg functions to generate a STREAMS message block by using information from specified buffers.

See putmsg(2) for function syntax, parameters, and errors.

Use the putpmsg function to send priority banded data down a Stream. The arguments have the same meaning as for the putmsg function.

See putpmsg(2) for function syntax, parameters, and errors.

5.2.2.9 The getmsg and getpmsg Functions

Use the getmsg and getpmsg functions to retrieve the contents of a message located at the Stream head read queue and place them into user specified buffer(s).

See getmsg(2) for function syntax, parameters, and errors.

Use the getpmsg function to receive priority banded data from a Stream. The arguments have the same meaning as for the getmsg function.

See getpmsg(2) for function syntax, parameters, and errors.

5.2.2.10 The poll Function

Use the poll function to identify the Streams to which a user can send data and from which a user can receive data.

See poll(2) for function syntax, parameters, and errors.

5.2.2.11 The isastream Function

Use the isastream function to determine if a file descriptor refers to a STREAMS file.

The following example shows how to use the isastream function to verify that you have opened a STREAMS-based pipe instead of a sockets-based pipe:

```
int fds[2];

pipe(fds);
if (isastream(fds[0]))

       printf("STREAMS based pipe\n");
else
       printf("Sockets based pipe\n");
```

See isastream(3) for function syntax, parameters, and errors.

5.2.2.12 The fattach Function

Use the fattach function to attach a STREAMS-based file descriptor to an object in the file system name space.

The following example shows how to use the fattach function to name a STREAMS-based pipe:

```
int fds[2];

pipe(fds);
fattach(fd[0], "/tmp/pipe1");
```

Note

The fattach function requires that the FFM_FS kernel option be configured. See the *System Administration* manual for information about configuring kernel options.

See fattach(3) for function syntax, parameters, and errors.

5.2.2.13 The fdetach Function

Use the fdetach function to detach a STREAMS-based file descriptor from a file name. A STREAMS-based file descriptor may have been attached by using the fattach function.

Note

The fdetach function requires that the File on File Mount File System (FFM_FS) kernel option is configured. See the *System Administration* manual for information about configuring kernel options.

See fdetach(3) for function syntax, parameters, and errors.

Table 5–2 lists and briefly describes the reference pages that contain STREAMS-related information. For further information about each component, refer to the appropriate reference page.

Table 5–2: STREAMS Reference Pages

Reference Page	Description
autopush(8)	Command that manages the system's database of automatically pushed STREAMS modules.
clone(7)	STREAMS software driver that finds and opens an unused major/minor device on another STREAMS driver.
close(2) [a]	Function that closes the file associated with a designated file descriptor.
dlb(7)	STREAMS pseudodevice driver that provides a communication path between BSD-style device drivers and STREAMS protocol stacks.
fattach(3)	Command that attaches a STREAMS-based file descriptor to a node in the file system.
fdetach(8)	Command that detaches a STREAMS-based file descriptor from a file name.
fdetach(3)	Function that detaches a STREAMS-based file descriptor from a file name.

getmsg(2)
getpmsg(2)
	Functions that reference a message positioned at the Stream head read queue.

ifnet(7)	STREAMS-based module that provides a bridge between STREAMS-based device drivers written to the Data Link Provider Interface (DLPI) and sockets.
isastream(3)	Function that determines if a file descriptor refers to a STREAMS file.
mkfifo(2)	Function that creates a unidirectional STREAMS-based file descriptor.
open(2) [a]	Function that establishes a connection between a file and a file descriptor.

Table 5–2: STREAMS Reference Pages (cont.)

Reference Page	Description
pipe(2)	Function that creates a bidirectional, STREAMS-based, interprocess communication channel.
poll(2)	Function that provides a general mechanism for reporting I/O conditions associated with a set of file descriptors and for waiting until one or more specified conditions becomes true.

putmsg(2)
putpmsg(2)
	Functions that generate a STREAMS message block.

read(2)[a]	Function that reads data from a file into a designated buffer.
strace(8)	Application that retrieves STREAMS event trace messages from the STREAMS log driver.
strchg(1)	Command that alters the configuration of a Stream.
strclean(8)	Command that removes STREAMS error log files.
strconf(1)	Command that queries about a Stream's configuration.
streamio(7)	Command that performs a variety of control functions on Streams.
strerr(3)	Daemon that receives error messages from the STREAMS log driver.
strlog(7)	Interface that tracks log messages used by STREAMS error logging and event tracing daemons.
strsetup(8)	Command that creates the appropriate STREAMS pseudodevices and displays the setup of your STREAMS modules.
timod(7)	Module that converts `ioctl` calls from a transport user supporting the Transport Interface (TI) into messages that a transport protocol provider supporting TI can consume.
tirdwr(7)	Module that provides a transport user supporting the TI with an alternate interface to a transport protocol provider supporting TI.
write(2) [a]	Function that writes data to a file from a designated buffer.

[a] The page is not STREAMS specific.

5.3 Kernel Level Functions

This section contains information with which the kernel programmer who writes STREAMS modules and drivers must be familiar. It contains information about:

- Module data structures
- Message data structures

- STREAMS processing routines for modules and drivers

5.3.1 Module Data Structures

When a module or driver is configured into the system, it must define its read and write queues and other module information.

The qinit, module_info, and streamtab data structures, all of which are located in the <sys/stream.h> header file, define read and write queues. STREAMS modules must fill in these structures in their declaration sections. See Appendix A for an example.

The only external data structure a module must provide is streamtab.

The qinit structure, shown in the following example, defines the interface routines for a queue. The read queue and write queue each have their own set of structures.

```
 struct  qinit {
    int     (*qi_putp)();           /* put routine */
    int     (*qi_srvp)();           /* service routine */
    int     (*qi_qopen)();          /* called on each open */
                                    /* or a push */
    int     (*qi_qclose)();         /* called on last close */
                                    /* or a pop */
    int     (*qi_qadmin)();         /* reserved for future use */
    struct module_info * qi_minfo;  /* information structure */
    struct module_stat * qi_mstat;  /* statistics structure (op-
                                    /* tional) */
};
```

The module_info structure, shown in the following example, contains module or driver identification and limit values:

```
 struct  module_info {
    unsigned short  mi_idnum;       /* module ID number */
    char           *mi_idname;      /* module name */
    long            mi_minpsz;      /* min packet size, for */
                                    /* developer use */
    long            mi_maxpsz;      /* max packet size, for */
                                    /* developer use */
    ulong           mi_hiwat;       /* hi-water mark, for */
                                    /* flow control */
    ulong           mi_lowat;       /* lo-water mark, for */
                                    /* flow control */
};
```

The streamtab structure, shown in the following example, forms the uppermost part of the declaration and is the only part which needs to be visible outside the module or driver:

```
 struct streamtab {
    struct qinit    * st_rdinit;    /* defines read QUEUE */
    struct qinit    * st_wrinit;    /* defines write QUEUE */
    struct qinit    * st_muxrinit;  /* for multiplexing drivers only */
    struct qinit    * st_muxwinit;  /* ditto */
};
```

5.3.2 Message Data Structures

Tru64 UNIX STREAMS messages consist of one or more linked message blocks. Each message block consists of a triplet with the following components:

- A data buffer

  The data buffer contains the binary data that makes up the message. STREAMS imposes no alignment rules on the format of data in the data buffer, aside from those imposed by messages processed at the Stream head.

- A mblk_t control structure

  The mblk_t structure contains information that the message owner can manipulate. Two of its fields are the read and write pointers into the data buffer.

- A dblk_t control structure

  The dblk_t structure contains information about buffer characteristics. For example, two of its fields point to the limits of the data buffer, while others contain the message type.

The Stream head creates and fills in the message data structures when data is traveling downstream from an application. The Stream end creates and fills in the message data structures when data is traveling upstream, as in the case of data coming from an external communications device.

The mblk_t and dblk_t structures, shown in the following examples, are located in the <sys/stream.h> header file:

```
/* message block */
struct  msgb {
        struct msgb *   b_next;    /* next message on queue */
        struct msgb *   b_prev;    /* previous message on queue */
        struct msgb *   b_cont;    /* next message block of message */
        unsigned char * b_rptr;    /* first unread data byte in buffer */
        unsigned char * b_wptr;    /* first unwritten data byte */
        struct datab *  b_datap;   /* data block */
        unsigned char   b_band;    /* message priority */
        unsigned char   b_pad1;
        unsigned short  b_flag;    /* message flags */
        long            b_pad2;
        MSG_KERNEL_FIELDS
};
typedef struct msgb     mblk_t;

/* data descriptor */
struct  datab {
        union {
                struct datab    * freep;
                struct free_rtn * frtnp;
        } db_f;
        unsigned char * db_base;    /* first byte of buffer */
        unsigned char * db_lim;     /* last byte+1 of buffer */
        unsigned char   db_ref;     /* count of messages pointing */
```

```
                                        /* to block */
        unsigned char   db_type;    /* message type */
        unsigned char   db_iswhat;  /* message status */
        unsigned int    db_size;    /* used internally */
        caddr_t         db_msgaddr; /* used internally */
        long            db_filler;
};
#define db_freep        db_f.freep
#define db_frtnp        db_f.frtnp

typedef struct datab    dblk_t;

/* Free return structure for esballoc */
typedef struct free_rtn {
        void    (*free_func)(char *, char *); /* Routine to free buffer */
        char *  free_arg;                     /* Parameter to free_func */
} frtn_t;
```

When a message is on a STREAMS queue, it is part of a list of messages linked by `b_next` and `b_prev` pointers. The `q_next` pointer points to the first message on the queue and the `q_last` pointer points to the last message on the queue.

5.3.3 STREAMS Processing Routines for Drivers and Modules

A module or driver can perform processing on the Stream that an application requires. To perform the required processing, the STREAMS module or driver must provide special routines whose behavior is specified by the STREAMS framework. This section describes the STREAMS module and driver routines, and the following kinds of processing they provide:

- Open processing
- Close processing
- Configuration processing
- Read side put processing
- Write side put processing
- Read side service processing
- Write side service processing

Note

STREAMS modules and drivers must provide open, close, and configuration processing. The other kinds of processing described in this section are optional.

The format used to describe each routine in this section is `XX_`*`routine_name`*. You should substitute the name of a user-written STREAMS module or driver for the XX. For example, the open routine for the user-written STREAMS pseudodevice driver echo would be echo_open.

5.3.3.1 Open and Close Processing

Only the open and close routines provide access to the u_area of the kernel. They are allowed to sleep only if they catch signals.

Open Processing

Modules and drivers must have open routines. The read side qinit structure, st_rdinit defines the open routine in its *qi_qopen* field. A driver's open routine is called when the application opens a Stream. The Stream head calls the open routine in a module when an application pushes the module onto the Stream.

The open routine has the following format:

```
 XX_open(q, devp, flag, sflag, credp)
       queue_t *q;   /* pointer to the read queue */
       dev_t *devp;  /* pointer to major/minor number
                        for devices */
       int flag;     /* file flag */
       int sflag;    /* stream open flag */
       cred_t *credp /* pointer to a credentials structure */
```

The open routine can allocate data structures for internal use by the STREAMS driver or module. A pointer to the data structure is commonly stored in the q_ptr field of the queue_t structure. Other parts of the module or driver can access this pointer later.

Close Processing

Modules and drivers must have close routines. The read side qinit structure, st_rdinit, defines the close routine in its *qi_qclose* field. A driver calls the close routine when the application that opened the Stream closes it. The Stream head calls the close routine in a module when it pops the module from the stack.

The close routine has the following format:

```
XX_close(q, flag, credp)
        queue_t *q;    /* pointer to read queue */
        int flag;      /* file flag */
        cred_t *credp  /* pointer to credentials structure */
```

The close routine may want to free and clean up internally used data structures.

5.3.3.2 Configuration Processing

The configure routine is used to configure a STREAMS module or driver into the kernel. It is specific to Tru64 UNIX and its use is illustrated in Section 5.4.

The configure routine has the following format:

```
XX_configure(op, indata, indatalen, outdata, outdatalen)
    sysconfig_op_t  op;          /* operation - should be */
                                 /* SYSCONFIG_CONFIGURE */
    str_config_t *  indata;      /* for drivers - describes the device */
    size_t          indatalen;   /* sizeof(str_config_t) */
    str_config_t *  outdata;     /* pointer to returned data */
    size_t          outdatalen;  /* sizeof(str_config_t) */
```

5.3.3.3 Read Side Put and Write Side Put Processing

There are both read side and write side XX_Xput routines; XX_wput for write side put processing and XX_rput for read side put processing.

Write Side Put Processing

The write side put routine, XX_wput, is called when the upstream module's write side issues a putnext call. The XX_wput routine is the only interface for messages to be passed from the upstream module to the current module or driver.

The XX_wput routine has the following format:

```
 XX_wput(q, mp)
       queue_t *q;  /* pointer to write queue */
       mblk_t *mp;  /* message pointer */
```

Read Side Put Processing

The read side put routine, XX_rput, is called when the downstream modules read side issues a putnext call. Because there is no downstream module, drivers that are Stream ends do not have read side put routines. The XX_rput routine is the only interface for messages to be passed from the downstream module to the current module.

The XX_rput routine has the following format:

```
 XX_rput(q, mp)
       queue_t *q;  /* pointer to read queue */
       mblk_t *mp;  /* message pointer */
```

The XX_Xput routines must do at least one of the following:

- Process the message
- Pass the message to the next queue (using putnext)

- Delay processing of the message by putting the message on the module's service routine (using putq)

The XX_Xput routine should leave any large amounts of processing to the service routine.

5.3.3.4 Read Side Service and Write Side Service Processing

If an XX_Xput routine receives a message that requires extensive processing, processing it immediately could cause flow control problems. Instead of processing the message immediately, the XX_rput routine (using the putq call) places the message on its read side message queue and the XX_wput places the message on its write queue. The STREAMS module notices that there are messages on these queues and schedules the module's read or write side service routines to process them. If the module's XX_rput routine never calls putq, the module does not require a read side service routine. Likewise, if the module's XX_wput routine never calls putq, the module does not require a write side service routine.

The code for a basic service routine, either read side or write side, has the following format:

```
  XXXsrv(q)
  queue_t *q;
{
        mblk_t *mp;

        while ((mp = getq(q)) != NULL)
        {
             /*
             * If flow control is a  problem, return
             * the message to the queue
             */

             if (!(canput(q->q_next))
                   return putbq(q, mp);
             /*
              * process message
              */

                putnext(q, mp);
        }
        return 0;
}
```

5.3.4 Tru64 UNIX STREAMS Concepts

The following STREAMS concepts are unique to Tru64 UNIX. This section describes these concepts and how they are implemented:

- Synchronization
- Timeout

5.3.4.1 Synchronization

Tru64 UNIX supports the use of more than one kernel STREAMS thread. Exclusive access to STREAMS queues and associated data structures is not guaranteed. Messages can move up and down the same Stream simultaneously, and more than one process can send messages down the same Stream.

To synchronize access to the data structures, each STREAMS module or driver chooses the synchronizaion level it can tolerate. The synchronization level determines the level of parallel activity allowed in the module or driver. Synchronization levels are defined in the `sa.sa_syn_level` field of the `streamadm` data structure which is defined in the module's or driver's configuration routine. The `sa.sa_syn_level` field must have one of the following values:

SQLVL_QUEUE

Queue Level Synchronizaton. This allows one thread of execution to access any instance of the module or driver's write queue at the same time another thread of execution can access any instance of the module or driver's read queue. Queue level synchronization can be used when the read and write queues do not share common data. The SQLVL_QUEUE argument provides the lowest level of synchronization available in the Tru64 UNIX STREAMS framework.

For example, the `q_ptr` field of the read and write queues do not point to the same memory location.

SQLVL_QUEUEPAIR

Queue Pair Level Synchronizaion. Only one thread at a time can access the read and write queues for each instance of this module or driver. This synchronization level is common for most modules or drivers that process data and have only per-stream state.

For example, within an instance of a module, the `q_ptr` field of the read and write queues points to the same memory location. There is no other shared data within the module.

SQLVL_MODULE

Module Level Synchronization. All code within this module or driver is single threaded. No more than one thread of execution can access all instances of the module or driver. For example, all instances of the module or driver are accessing data.

SQLVL_ELSEWHERE

Arbitrary Level Synchronization. The module or driver is synchronized with some other module or driver. This level is used to synchronize a group of modules or drivers that access each other's data. A character string is passed with this option in the sa.sync_info field of the streamadm structure. The character string is used to associate with a set of modules or drivers. The string is decided by convention among the cooperating modules or drivers.

For example, a networking stack such as a TCP module and an IP module which share data might agree to pass the string tcp/ip. No more than one thread of execution can access all modules or drivers synchronized on this string.

SQLVL_GLOBAL

Global Level Synchronization. All modules or drivers under this level are single threaded. Note there may be modules or drivers using other levels not under the same protection. This option is available primarily for debugging.

5.3.4.2 Timeout

The kernel interface to timeout and untimeout is as follows:

```
timeout(func, arg, ticks);
untimeout(func, arg);
```

However, to maintain source compatibilty with AT&T System V Release 4 STREAMS, the <sys/stream.h> header file redefines timeout to be the System V interface, which is:

```
id = timeout(func, arg, ticks);
untimeout(id);
```

The id variable is defined to be an int.

STREAMS modules and drivers must use the System V interface.

5.4 Configuring a User-Written STREAMS-Based Module or Driver in the Tru64 UNIX Kernel

For your system to access any STREAMS drivers or modules that you have written, you must configure the drivers and modules into your system's kernel.

STREAMS modules or drivers are considered to be configurable kernel subsystems; therefore, follow the guidelines in the *Programmer's Guide* for configuring kernel subsystems.

The following sample procedure shows how to add to the kernel a STREAMS-based module (which can be a pushable module or a hardware or pseudodevice driver) called mymod, with it’s source files mymodule1.c and mymodule2.c.

1. Declare a configuration routine in your module source file, in this example, /sys/streamsm/mymodule1.c.

   Example 5–1 shows a module (mymod_configure) that can be used by a module. To use the routine with a driver, do the following:

   a. Remove the comment signs from the following line:

      ```
      /* sa.sa_flags    = STR_IS_DEVICE | STR_SYSV4_OPEN; */
      ```

      This line follows the following comment line:

      ```
      /* driver */
      ```

   b. Comment out the following line:

      ```
      sa.sa_flags       = STR_IS_MODULE | STR_SYSV4_OPEN;
      ```

      This line follows the following comment line:

      ```
      /* module */
      ```

Example 5–1: Sample Module

```
/*
*  Sample mymodule.c
*/
⋮

#include <sys/sysconfig.h>
#include <sys/errno.h>

struct streamtab mymodinfo = { &rinit, &winit };

cfg_subsys_attr_t mymod_attributes[] = {        1
       {,0,0,0,0,0,0}         /* required last element */
};
int
mymod_configure(
  cfg_op_t   op;
  caddr_t       indata;
  ulong         indata_size;
  caddr_t       outdata;
  ulong         outdata_size)
{
        dev_t  devno = NODEV;        2
        struct streamadm sa;
        if  (op != CFG_OP_CONFIGURE)        3
```

Example 5–1: Sample Module (cont.)

```
                   return EINVAL;

        sa.sa_version     = OSF_STREAMS_10;
        /* module */       4
        sa.sa_flags       = STR_IS_MODULE | STR_SYSV4_OPEN;
        /* driver */
        /* sa.sa_flags    = STR_IS_DEVICE | STR_SYSV4_OPEN; */
        sa.sa_ttys        = NULL;
        sa.sa_sync_level  = SQLVL_MODULE;       5
        sa.sa_sync_info   = NULL;
        strcpy(sa.sa_name, "mymod");

        if ((devno = strmod_add(devno, &mymodinfo, &sa)) == NODEV)
        {
                return ENODEV;
        }

        return ESUCCESS;
}
```

$\boxed{1}$ The subroutine in this example supplies an empty attribute table and no attributes are expected to be passed to the subroutine. If you want to develop attributes for your module, refer to the *Programmer's Guide*.

$\boxed{2}$ The first available slot in the `cdevsw` table is automatically allocated for your module. If you wish to reserve a specific device number, you should define it after examining the `cdevsw` table in the `conf.c` program. For more information on the `cdevsw` table and how to add device driver entries to it, see the *Writing Device Drivers* manual.

$\boxed{3}$ This example routine only supports the `CFG_OP_CONFIGURE` option. See the *Programmer's Guide* for information on other configuration routine options.

$\boxed{4}$ The `STR_SYSV4_OPEN` option specifies to call the module's or device's `open` and `close` routines, using the AT&T System V Release 4 calling sequence. If this bit is not specified, the AT&T System V Release 3.2 calling sequence is used.

$\boxed{5}$ Other options for the `sa.sync_level` field are described in Section 5.3.4.

2. Statically link your module with the kernel.

If you want to make the STREAMS module dynamically loadable, see the *Programmer's Guide* for information on configuring kernel subsystems. If the module you are configuring is a hardware device driver, also see the *Writing Device Drivers* manual.

To statically link your module with the kernel, put your module's source files (`mymodule1.c` and `mymodule2.c`) into the `/sys/streamsm` directory and add an entry for each file to the `/sys/conf/files` file. The following example shows the entries in the `/sys/conf/files` file for `mymodule1.c` and `mymodule2.c`:

```
streamsm/mymodule1.c   optional mymod Notbinary
streamsm/mymodule2.c   optional mymod Notbinary
```

Add the `MYMOD` option to the kernel configuration file. The default kernel configuration file is `/sys/conf/`*`HOSTNAME`* (where *`HOSTNAME`* is the name of your system in uppercase letters). For example, if your system is named `TRU64`, add the following line to the `/sys/conf/TRU64` configuration file:

```
options  MYMOD
```

If you are configuring a hardware device driver continue with step 3; if not, go to step 4.

3. If you are configuring a hardware device driver, complete steps 3a to 3d.

   If you are not configuring a hardware device driver, go to step 4.

   If you are configuring a hardware device driver, you should already have an `XXprobe` and an `interrupt` routine defined. See the *Writing Device Drivers* manual for information about defining `probe` and `interrupt` routines.

   a. Add the following line to the top of the device driver configuration file, which for this example is `/sys/streams/mydriver.c`:

      ```
      #include <io/common/devdriver.h>
      ```

   b. Define a pointer to a controller structure; for example:

      ```
      struct controller *XXinfo;
      ```

      For information on the controller structure, see the *Writing Device Drivers* manual.

   c. Declare and initialize a driver structure; for example:

      ```
       struct driver XXdriver =
      {
         XXprobe, 0, 0, 0, 0, XXstd, 0, 0, "XX", XXinfo
      };
      ```

      For information on the driver structure, see the *Writing Device Drivers* manual.

d. Add the controller line to the kernel configuration file. The default kernel configuration file is `/sys/conf/`*`HOSTNAME`* (where *`HOSTNAME`* is the name of your system in uppercase letters). For example, if your system name is `TRU64`, would add a line similar to the following to the `/sys/conf/TRU64` configuration file:

```
controller XX0 at bus vector XXintr
```

For information about the possible values for the bus keyword, see the *System Administration* manual.

4. Reconfigure, rebuild, and boot the new kernel for this system by using the `doconfig` command. See `doconfig`(8) or the *System Administration* manual for information on reconfiguring your kernel.

5. Run the `strsetup -c` command to verify that the device is configured properly:

```
# /usr/sbin/strsetup -c

STREAMS Configuration Information...Wed Jun  2 09:30:11 1994

           Name       Type   Major      Minor  Module ID
           ----       ----   -----      -----  ---------
          clone                 32          0
            ptm     device      37          0       7609
            pts     device       6          0       7608
            log     device      36          0         44
           nuls     device      38          0       5001
           echo     device      39          0       5000
            sad     device      40          0         45
           pipe     device      41          0       5304
          kinfo     device      42          0       5020
       xtisoUDP     device      43          0       5010
       xtisoTCP     device      44          0       5010
            dlb     device      49          0       5010
        bufcall     module                             0
          timod     module                          5006
         tirdwr     module                             0
          ifnet     module                          5501
          ldtty     module                          7701
           null     module                          5003
           pass     module                          5003
           errm     module                          5003
          spass     module                          5007
         rspass     module                          5008
        pipemod     module                          5303

        Configured devices = 11, modules = 11
```

5.5 Device Special Files

This section describes the STREAMS device special files and how they are created. It also provides an overview of the clone device.

All STREAMS drivers must have a character special file created on the system. These files are usually in the /dev/streams directory and are created at installation, or by running the /usr/sbin/strsetup utility.

A STREAMS driver has a device major number associated with it which is determined when the driver is configured into the system. Drivers other than STREAMS drivers usually have a character special file defined for each major and minor number combination. The following is an example of an entry in the /dev/rdisk directory:

```
crw-------   1 root     system      8,   1024 Aug 25 15:38 dsk1a
crw-------   1 root     system      8,   1025 Aug 25 15:38 dsk1b
crw-------   1 root     system      8,   1026 Aug 25 15:38 dsk1c
```

In this example, dsk1a has a major number of 8 and a minor number of 1024. The dsk1b device has a major number of 8 and a minor number of 1025, and dsk1c has a major number of 8 and a minor number 1026.

You can also define character special files for each major and minor number combination for STREAMS drivers. The following is an example of an entry in the /dev/streams directory:

```
crw-rw-rw-   1 root   system   32,  0 Jul 13 12:00 /dev/streams/echo0
crw-rw-rw-   1 root   system   32,  1 Jul 13 12:00 /dev/streams/echo1
```

In this example, echo0 has a major number of 32 and a minor number of 0, while echo1 has a major number of 32, and a minor number of 1.

For an application to open a unique Stream to a device, it must open a minor version of that device that is not already in use. The first application can do an open on /dev/streams/echo0 while the second application can do an open on /dev/streams/echo1. Since each of these devices has a different minor number, each application acquires a unique Stream to the echo driver. This method requires that each device (in this case, echo) have a character special file for each minor device that can be opened to it. This method also requires that the application determine which character special file it should open; it does not want to open one that is already in use.

The clone device offers an alternative to defining device special files for each minor device that can be opened. When the clone device is used, each driver needs only one character special file and, instead of an application having to determine which minor devices are currently available, clone allows a second (or third) device to be opened using its (clone device's) major number. The minor number is associated with the device being opened (in this case, echo). Each time a device is opened using clone device's major number, the STREAMS driver interprets it as a unique Stream.

The strsetup command sets up the entries in the /dev/streams directory to use the clone device. The following is an example entry in the /dev/streams file:

```
crw-rw-rw-   1 root   system   32,  18 Jul 13 12:00 /dev/streams/echo
```

In this example, the system has assigned the major number 32 to the clone device. The number 18 is the major number associated with echo. When an application opens /dev/streams/echo, the clone device intercepts the call. Then, clone calls the open routine for the echo driver. Additionally, clone notifies the echo driver to do a clone open. When the echo driver realizes it is a clone open, it will return its major number, 18, and the first available minor number.

Note

The character special files the /usr/sbin/strsetup command creates are created by default in the /dev/streams directory with clone as the major number. If you configure into your kernel a STREAMS driver that either does not use clone open, or uses a different name, you must modify the /etc/strsetup.conf file described in strsetup.conf(4).

To determine the major number of the clone device on your system, run the strsetup –c command.

5.6 Error and Event Logging

STREAMS error and event logging involves the following:

- The error logger daemon
- The trace logger
- The strclean command

The error logger daemon, strerr, logs in a file any error messages sent to the STREAMS error logging and event tracing facility.

The trace logger, strace, writes to standard output trace messages sent to the STREAMS error logging and event tracing facility.

The strclean command can be run to clean up any old log files generated by the strerr daemon.

A STREAMS module or driver can send error messages and event tracing messages to the STREAMS error logging and event tracing facility through the strlog kernel interface. This involves a call to strlog.

The following example shows a STREAMS driver printing its major and minor device numbers to both the STREAMS error logger and the event tracing facility during its open routine:

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

strlog(MY_DRIVER_ID, 0, 0, SL_ERROR 1 SL_TRACE,
       "My driver:  mydriver_open() - major=%d,minor=%d",
 major(dev,minor(dev));
```

A user process can also send a message to the STREAMS error logging and event tracing facility by opening a Stream to `/dev/streams/log` and calling `putmsg`. The user process must contain code similar to the following to submit a log message to `strlog`:

```
struct strbuf ctl, dat;
struct log_ctl lc;
char *message = "Last edited by <username> on <date>";

ctl_len = ctl.maxlen = sizeof (lc);
ctl.buf = (char *)&lc;

dat.len = dat.maxlen = strlen(message);
dat.buf = message;
lc.level = 0;
lc.flags = SL_ERROR|SL_NOTIFY;

putmsg (log, &ctl, &dat, 0);
```

6

Extensible SNMP Application Programming Interface

The Simple Network Management Protocol (SNMP) is an application layer protocol that allows remote management and data collection from networked devices. A networked device can be anything that is connected to the network, such as a router, a bridge, or a host.

A managed networked device contains software that acts as the SNMP agent for the device. It handles the application layer protocol for SNMP and carries out the management commands. These commands consist of getting information and setting of operational parameters.

There are also network management application programs (usually running on a host somewhere on the network) that send SNMP commands to the various managed devices on the network to perform the management tasks. These tasks can consist of configuration management, network traffic monitoring and network trouble shooting.

The Extensible Simple Network Management Protocol (eSNMP) is the SNMP agent architecture for Tru64 UNIX or earlier versions of DIGITAL UNIX. It includes a master agent process and multiple related processes containing eSNMP subagents. The master agent performs the SNMP protocol handling and the subagents perform the requested management commands. This chapter assumes you are familiar with the following:

- SNMP protocol
- Management Information Base (MIB) definitions and Request For Comments (RFCs)
- Object Identifiers (OIDs) and the International Standards Organization (ISO) registration hierarchy (1.3.6.1.2.1, and so on)
- The C programming language

This chapter provides the following information:

- Overview of eSNMP
- Overview of the eSNMP application programming interface (API)
- Detailed information on the eSNMP routines

6.1 Overview of eSNMP

The following sections describe the components and architecture of the eSNMP agent. It contains information on the following:

- Components of eSNMP
- Architecture
- SNMP Versions

6.1.1 Components of eSNMP

The eSNMP components are as follows:

- `/usr/sbin/snmpd` — The master agent daemon.
- `/usr/sbin/os_mibs` — The host MIB and networking subagent daemon.
- `/usr/sbin/svrMgt_mib` — A server management subagent daemon.
- `/usr/sbin/svrSystem_mib` — A server system subagent daemon.
- `/usr/sbin/mosy` — The MIB compiler.
- `/usr/sbin/snmpi` — The object table code generator.
- `/usr/shlib/libesnmp.so` — The eSNMP library.
- `/usr/include/esnmp.h` — eSNMP definitions.
- `/usr/examples/esnmp/*` — Example code.

The Management Information Base (MIB) defines a set of data elements that relate to network management. Many of these are standardized in the RFCs that are produced as a result of the Internet Engineering Task Force (IETF) working group standardization effort of the Internet Society.

The data elements defined in the RFCs are identified using a naming scheme with a hierarchical structure. Each name at each level of the hierarchy has a number associated with it. You can refer to the data elements in the MIB definitions by name or by their corresponding sequence of numbers, which are called the Object Identifier (OID). You can extend an OID for a specific data element further by adding more numbers to identify a specific instance of the data element. The entire collection of managed data elements is called the MIB tree.

Each SNMP agent implements those MIB elements that pertain to the device being managed, plus a few common MIB elements. These are the supported MIB tree elements. An extensible SNMP agent is one that permits its supported MIB tree to be distributed among various processes and change dynamically.

The eSNMP consists of a single master agent and any number of subagents. The master agent handles the SNMP protocols, supports MIBs related to SNMP itself, and maintains a registry of connected subagents and the MIB subtrees they support. The master agent for eSNMP is the daemon process /usr/sbin/snmpd.

Tru64 UNIX provides subagents that implement several different MIBs, both IETF standard and proprietary to Compaq. Refer to the Software Product Description (SPD), *Cluster Technical Overview*, and the snmpd(8) reference page for more information. These subagents (and any third-party subagents) together with the master agent function as a single SNMP agent for the host.

6.1.2 Architecture

The master agent listens on the preassigned User Datagram Protocol (UDP) port for an incoming SNMP request. When the master agent receives an SNMP request, it authenticates it against the local security database and handles any authentication or protocol errors. If the request is valid, the snmpd daemon consults its MIB registry. (See snmpd(8) for more information.) For each MIB object contained in the request it determines which registered MIB could contain that object and which subagent has registered that MIB. The master agent then builds a series of messages; one for each subagent that will be involved in this SNMP request.

Each subagent program is linked with the shareable library libesnmp.so. This library contains the protocol implementation that enables communication between the master agent and the subagent. This code parses the master agent's message and consults its local object table.

The object table is a data structure that is defined and initialized in code emitted by the snmpi and mosy MIB compiler tools. It contains an entry for each MIB object that is contained in the MIBs implemented in that subagent. One part of an object table entry is the address of a function that services requests for the MIB object. These functions are called method routines.

The eSNMP library code calls into the indicated method routine for each of the MIB variables in the master agent's message. The eSNMP library code creates a response packet based on the function return values and sends it back to the master agent.

The master agent starts a timer and assembles the response packets from all involved subagents. The master agent may rebuild and resend a new set of subagent messages, depending on the specific request; for example, a GetNext request. When the master agent has all required data or error responses or has timed out waiting for a response from a subagent, it builds an SNMP response message and sends it to the originating SNMP application. The interaction between the master agent and subagent is invisible to the requesting SNMP management application.

Subagent programs are linked against the `libesnmp.so` shareable library, which performs all protocol handling and dispatching. Subagent developers need to code method routines for their defined MIB objects only.

6.1.3 SNMP Versions

Extensible SNMP support for SNMPv2c exists in the following areas. This is based on RFCs 1901 through 1908, inclusive:

- The MIB tools (the `mosy` and `snmpi` programs) support SNMPv2c Structure of Management Information for SNMPv2 (SMIv2) and textual conventions.
- The eSNMP library API supports SNMPv2c, variable binding exceptions, and error codes.
- The master agent currently supports SNMPv1 and SNMPv2c in a bilingual manner. All SNMPv2c-specific information from the subagent is mapped, when necessary, into SNMPv1-adherent data according to RFC 2089. For example, if a management application makes a request using SNMPv1 PDUs, the master agent replies using SNMPv1 PDUs, mapping any SNMPv2c SMI items received from subagents. This means that subagents created with a previous version of the eSNMP API do not require any code changes and do not have to be recompiled.

6.1.4 AgentX

All communication by the `libesnmp.so` library to and from the master agent is done through an implementation of RFC 2741, *Agent Extensibility (AgentX) Version 1*. RFC 2741 defines a standard protocol for communications between extensible agent components, a master agent and multiple subagents. This means that subagents that use the eSNMP API will function correctly with no modification if a different vendor's master agent (one that conforms to RFC 2741) is running on the Tru64 UNIX host.

6.2 Overview of the Extensible SNMP Application Programming Interface

The subagent's functions are to establish communications with the master agent, register the MIB subtrees that it intends to handle, and process requests from the master agent. It must also be able to send SNMP traps on behalf of the host application.

The subagent consists of the following:

- A main function written by the developer

- The eSNMP library routines that perform the AgentX protocol work
- The method routines written by the developer that handle specific MIB elements
- The object table structures generated from MIB definition files using the `mosy` and `snmpi` programs

The subagent is usually embedded within an application, such as a router daemon. Subagent processing is only a small part of the work performed by the process. In this case, the main event loop of the application makes the calls into the eSNMP library. In other cases, the subagent is a standalone daemon process that has its own main routine.

While processing a packet received from the master agent, the eSNMP library calls the specified method routine for each requested MIB variable. Each defined MIB variable in the subagent's object table has a pointer to the method routine for handling requests on that MIB variable. Since the object tables are generated by the `mosy` and `snmpi` programs, the method routine names are static.

The eSNMP developer's kit provided with the operating system consists of the following:

- `/usr/sbin/mosy` — MIB compiler utility
- `/usr/sbin/snmpi` — Object table code generator utility
- `/usr/examples/esnmp/mib-converter.sh` — MIB text extraction tool
- `/usr/shlib/libesnmp.so` — eSNMP shared library
- `/usr/include/esnmp.h` — eSNMP definitions file
- `/usr/examples/esnmp/*` — Subagent example source code

The eSNMP shared library (`libesnmp.so`) provides the following services:

- Master-agent to subagent protocol handling routines
- Routines for communicating with the master agent on behalf of the subagent, as follows:
  - `esnmp_init` — Initializes the protocol (performs a handshake with the master agent)
  - `esnmp_allocate` — Requests the master agent to allocate a value for one or more specific index objects (OIDs)
  - `esnmp_deallocate` — Requests the master agent to deallocate a value or values for one or more index objects (OIDs)
  - `esnmp_register` — Registers a MIB subtree with the master agent
  - `esnmp_poll` — Processes a packet from the master agent

  - `esnmp_trap` — Requests the master agent to generate an SNMP trap
  - `esnmp_are_you_there` — Pings the master agent
  - `esnmp_unregister` — Unregisters a MIB subtree
  - `esnmp_term` — Ends communication with the master agent and terminates extensible SNMP
  - `esnmp_sysuptime` — Time handling and synchronization

- Support routines useful for developing method routines. See Section 6.3 for a complete list and description of each eSNMP support routine.
- The `esnmp.h` header file is associated with the eSNMP library. This file defines all data structures, constants, and function prototyes required to implement subagents to this API.

6.2.1 MIB Subtrees

Understanding MIB subtrees is crucial to understanding the eSNMP API and how your subagent will work.

> **Note**
>
> This section assumes that you understand the OID naming structure used in SNMP. If not, refer to RFC 1902: *Structure of Management Information for Version 2 of the Simple Network Management Protocol (SNMPv2).*

The information in SNMP is structured hierarchically like an inverted tree. In this hierarchy, data is only associated with leaf nodes. Each node has a name and a number. Each node can also be identified by an OID, which is a concatenation of the non-negative numbers, called **sub-identifiers**, that exist on the path from the root node down to that node in the tree. An OID must be at least two sub-identifiers in length, and can be at most 128 sub-identifiers in length. Valid sub-identifier 1 values range from 0 to 2, inclusive; sub-identifier 2 values range from 0 to 39, inclusive; and the remaining sub-identifier values can be any non-negative number.

For example, the `chess` MIB provided with the sample code in the `/usr/examples/esnmp` directory has an element with the name `chess`. The OID for the element `chess` is `1.3.6.1.4.1.36.2.15.2.99`, which is derived from its position in the hierarchy of the tree:

```
iso(1)
 org(3)
  dod(6)
   internet(1)
    private(4)
```

```
     enterprise(1)
      digital(36)
       ema(2)
        sysobjects(15)
         decosf(2)
          chess(99)
```

Any node in the MIB hierarchy can define a MIB subtree. All elements within the subtree have an OID that starts with the OID of the subtree base. For example, if we define `chess` to be a MIB subtree base, the elements with the same prefix as the `chess` OID are all within the MIB subtree:

```
chess                  1.3.6.1.4.1.36.2.15.2.99
 chessProductID        1.3.6.1.4.1.36.2.15.2.99.1
 chessMaxGames         1.3.6.1.4.1.36.2.15.2.99.2
 chessNumGames         1.3.6.1.4.1.36.2.15.2.99.3
 gameTable             1.3.6.1.4.1.36.2.15.2.99.4
  gameEntry            1.3.6.1.4.1.36.2.15.2.99.4.1
   gameIndex           1.3.6.1.4.1.36.2.15.2.99.4.1.1
   gameDescr           1.3.6.1.4.1.36.2.15.2.99.4.1.2
gameNumMoves           1.3.6.1.4.1.36.2.15.2.99.4.1.3
   gameStatus          1.3.6.1.4.1.36.2.15.2.99.4.1.4
 moveTable             1.3.6.1.4.1.36.2.15.2.99.5
  moveEntry            1.3.6.1.4.1.36.2.15.2.99.5.1
   moveIndex           1.3.6.1.4.1.36.2.15.2.99.5.1.1
   moveByWhite         1.3.6.1.4.1.36.2.15.2.99.5.1.2
   moveByBlack         1.3.6.1.4.1.36.2.15.2.99.5.1.3
   moveStatus          1.3.6.1.4.1.36.2.15.2.99.5.1.4
 chessTraps            1.3.6.1.4.1.36.2.15.2.99.6
  moveTrap             1.3.6.1.4.1.36.2.15.2.99.6.1
```

It is this MIB subtree base that is registered with the master agent to tell it that this subagent handles all requests related to the elements within the subtree.

The master agent expects a subagent to handle all objects subordinate to the registered MIB subtree. This principle guides your choice of MIB subtrees.

For example, registering a subtree of `chess` is reasonable because it is realistic to assume that the subagent could handle all requests for elements in this subtree. Registering an entire application-specific MIB usually makes sense because the particular application expects to handle all objects defined in the application-specific MIB.

Registering a subtree of `transmission` (under MIB-2) would be a mistake, because it is unlikely that the subagent is prepared to handle every defined MIB object subordinate to transmission (FDDI, Token Ring, and so on).

A subagent may register as many MIB subtrees as it wants. It can register OIDs that overlap with other registrations by itself or other subagents;

however, it cannot register the same OID more than once. Subagents can register and unregister MIB subtrees at any time after communication with the master agent is established.

Normally it is the non-leaf nodes that are registered as a subtree with the master agent. However, leaf nodes (those representing a MIB variable that can have an instance and an associated value), or even specific instances, can be registered as a subtree.

The master agent delivers requests to the subagent that has the MIB subtree with the longest prefix and the best priority.

6.2.2 Object Tables

The `mosy` and `snmpi` utilities are used to generate the C language code that defines the object tables from the MIBs. The object tables are defined in the emitted files *subtree*`_tbl.h` and *subtree*`_tbl.c`, files that are compiled into your subagent.

These modules are created by the utilities; it is recommended that you do not edit them. If the MIBs change or a future version of the eSNMP development utilities require your object tables to be rebuilt, it is easier to rebuild the files and recompile them if you did not edit the files.

6.2.2.1 The subtree_tbl.h File

The *subtree*`_tbl.h` file contains the following information:

- A declaration of the MIB subtree structure
- Index definitions for each MIB variable in the MIB subtree
- Enumeration definitions for MIB variables with enumerated values
- MIB group data structure definitions
- Method routine function prototypes

Declaration of the MIB Subtree Structure

The MIB subtree is automatically initialized by code in the *subtree*`_tbl.c` file. A pointer to this structure is passed to the `esnmp_register` routine to register a MIB subtree with the master agent. All access to the object table for this MIB subtree is through this pointer. The declaration has the following form:

```
extern SUBTREE subtree_subtree;
```

Index Definitions

Index definitions for each MIB variable in the `SUBTREE` have the following:

```
#define I_mib-variable    nnnn
```

These values are unique for each MIB variable within a MIB subtree and are the index into the object table for this MIB variable. These values are also generally used to differentiate between variables that are implemented in the same method routine so they can be used in a switch operation.

Enumeration Definitions

Enumeration definitions for those MIB variables that are defined with “SYNTAX INTEGER” and enumerated values have the following form:

```
#define D_mib-variable_enumeration-name    value
```

Enumeration definitions are useful since they describe the architected value that enumerated integer MIB variables may take on; for example:

```
/* enumerations for gameEntry group */
   #define   D_gameStatus_complete        1
   #define   D_gameStatus_underway        2
   #define   D_gameStatus_delete          3
```

Data Structure Definitions

MIB group data structure definitions have the following form:

```
typedef struct xxx {
     type mib-variable;
    :
     char mib-variable_mark;
    :
    } mib-group_type
```

A data structure is emitted for each MIB group within the MIB subtree. Each structure definition contains a field representing each MIB variable within the group. If the MIB variable name is not unique within the pool of MIBs presented to the snmpi program at the time the *subtree*_tbl.h file is built, the snmpi program does not qualify the name with the name of its parent variable (group name) to make it unique. In addition to the MIB variable fields, the structure includes a 1-byte *mib-variable*_mark field for each variable. You can use these for maintaining status of a MIB variable; for example, the following is the group structure for the chess MIB:

```
typedef struct _chess_type {
    OID   chessProductID;
    int   chessMaxGames;
    int   chessNumGames;

    char chessProductID_mark;
    char chessMaxGames_mark;
    char chessNumGames_mark;
} chess_type;
```

MIB group structures are provided for convenience, but are not mandatory. You can use whatever structure is easiest for you in your method routine.

Method Routine Function Prototypes

Each MIB group within the MIB subtree has a method routine prototype defined. A MIB group is a collection of MIB variables that are leaf nodes and share a common parent node.

There is always a function prototype for the method routine that handles the `Get`, `GetNext`, and `GetBulk` operations. If the group contains any writable variables, there is also a function prototype for the method routine that handles `Set` operations. Pointers to these routines appear in the MIB subtree's object table, which is initialized in the *subtree*`_tbl.c` module. You must write method routines for each prototype that is defined, as follows:

```
extern int mib-group_get(
         METHOD *method);

extern int mib-group_set(
         METHOD *method);
```

For example:

```
extern int chess_get(METHOD *method);
extern int chess_set(METHOD *method);
```

Method routines are discussed in more detail in Section 6.3.2.3.

6.2.2.2 The subtree_tbl.c File

The *subtree*`_tbl.c` file contains the following information:

- An array of integers representing the OIDs for each MIB variable
- An array of `OBJECT` structures. (See `esnmp.h`.)
- The initialized `SUBTREE` structure

Array of Integers

The array of integers used as the OIDs of each MIB variable in the MIB subtree has the following form:

```
static unsigned int elems[] = { ...
```

OBJECT Structures

There is one `OBJECT` for each MIB variable within the MIB subtree. (See `esnmp.h`.)

An `OBJECT` represents a MIB variable and has the following fields:

- object_index — The constant I_*mib-variable* from the *subtree*_tbl.h file.
- oid — The variable's OID (points to a part of elems[]).
- type — The variable's data type.
- getfunc — The address of method routine to call for Get operations.
- setfunc — The address of method routine to call for Set operations.

The master agent has no knowledge of object tables or MIB variables. It only maintains a registry of MIB subtrees. When a request for a particular MIB variable arrives, it is processed as follows. In the following procedure, the MIB variable is mib_var and the MIB subtree is subtree_1:

1. The master agent finds subtree_1 as the authoritative region for the mib_var in the registry of MIB subtrees. The authoritative region is determined as the registered MIB subtree that has the longest prefix and the best priority.
2. The master agent sends an eSNMP message to the subagent that registered subtree_1.
3. The subagent consults its list of registered MIB subtrees and locates subtree_1. It searches the object table of subtree_1 and locates the following:
   - mib_var (for Get and Set requests)
   - The first object lexicographically after mib_var (for GetNext or GetBulk requests)
4. The subagent calls the appropriate method routine. If the method routine completes successfully, the data is returned to the master agent. If not, for Get or Set, an error is returned. For GetNext or GetBulk, the libesnmp library code keeps trying subsequent objects in the object table of subtree_1 until a method routine returns success or the table is exhausted. In either case an appropriate response is returned.
5. If the master agent detects subtree_1 could not return data on a GetNext or GetBulk routine, it iteratively tries the MIB subtree lexicographically after subtree_1 until a subagent returns a value or the registry of MIB subtrees is exhausted.

Initialized SUBTREE Structure

A pointer to the SUBTREE structure is passed to the esnmp_register eSNMP library routine to register the MIB subtree. It is through this pointer that the library routines find the object structures. The following is an example of the chess subtree structure:

```
SUBTREE chess_subtree = { "chess", "1.3.6.1.4.1.36.2.15.2.99",
                          { 11, &elems[0] }, objects, I_moveStatus};
```

The `SUBTREE` structure has the following elements:

- *name* — The name of the base node of the MIB subtree.
- *dots* — The ASCII string representation of the MIB subtree's OID; it is what actually gets registered.
- *oid* — The OID of the base node of the subtree; it points back to the array of integers.
- *object_tbl* — A pointer to the array of objects in the object table. It is indexed by the `I_`*xxxx* definitions found in the *subtree*`_tbl.h` file.
- *last* — The index of the last object in the `object_tbl` file. It is used to determine when the end of the table has been reached.

The final section of the *subtree*`_tbl.c` contains short routines for allocating and freeing the *mib-group*`_type` structures. These are provided as a convenience and are not a required part of the API.

6.2.3 Implementing a Subagent

As a subagent developer, you are usually presented with a UNIX application, daemon, or driver (such as the `gated` daemon or ATM driver) and need to implement an SNMP interface. The following steps explain how you do this:

1. Obtain a MIB specification.

   MIB development starts with a MIB specification, usually in the form of RFCs. For SNMPv1, the specifications are written in concise MIB format according to RFC 1212. For SNMPv2c, the specifications are written in SMIv2 and the textual conventions as specified in RFC 1902 and RFC 1903, respectively. Designing and specifying a MIB is beyond the scope of this document; it is assumed you have a MIB specification.

   The standard RFCs can be obtained from the RFC editor at the following URL:

   `http://www.rfc-editor.org/rfc.html`

   If you have to build your own MIB specification, you can look at a similar MIB written by another vendor. One source for public MIBs is in the archives section of the Network Management page at the following URL:

   `http://smurfland.cit.buffalo.edu/NetMan/index.html`

   You need MIBs for all of the elements you are implementing in the subagent and for any elements referenced by these MIBs (such that all element names resolve to the OID numbers). As a minimum you will need the SMIv2 MIB, `snmp-smi.my`, and the textual conventions, `snmp-tc.my`. These are in the `/usr/examples/esnmp` directory.

2. Compile your MIBs.

   Once you obtain MIB definitions, use them to generate the object tables for your new subagent. The objective is to take the MIB specification text for each of the MIBs, extract the ASN.1 specifications, and compile them into C language modules that contain the local object tables.

   Compile your MIBs using the following tools:

   – `mib-converter.sh`

     The `mib-converter.sh` is an `awk` shell script that extracts the MIB ASN.1 definitions from the RFC text. This step removes the text before and after the MIB definition and removes page headings and footings.

     The `mib-converter.sh` script may not remove everything that needs to be removed; therefore, you may need to remove some things manually, using a text editor. The following is an example of how to use the `mib-converter.sh` script:

     ```
     # /usr/examples/esnmp/mib-converter.sh mib-def.txt > \
     mib-def.my
     ```

     Be careful; some RFCs contain more than one MIB definition. You can only use the `mib-converter.sh` script shell on RFCs that contain a single MIB definition. The `mosy` compiler may not handle it either. If you use an RFC that contains more than one MIB definition, make each one into a separate input file. The resulting output files containing the extracted MIB ASN.1 definitions should be in the following form:

     `mib-def``.my`

   – `mosy`

     The `mosy` compiler parses `.my` files created by the `mib-converter.sh` script and compiles them into `.defs` files. The `.defs` files describe the object hierarchy within the MIB. The `.defs` files are front-ends to several tools. The following is an example of how to use the `mosy` compiler:

     ```
     # mosy mib-def.my
     ```

     The `mosy` compiler produces *`mib-def`*`.defs` files.

     The `mosy` program is taken from ISODE 8.0 (distributed with the 4BSD/ISODE SNMPv2 package).

   – `snmpi`

     The MIB data initializer creation program (`snmpi`) reads a concatenation of the `.def` files compiled by the `mosy` compiler and generates the C code to define the static structures of the object table for a specified MIB subtree.

Note

> The snmpi program supplied with this operating system is different from the snmpi program in 4BSD/ISODE SMUX.

Concatenate the .def files the mosy compiler compiles into the objects.defs file. Be sure to include the compiled versions of snmp-smi.my and snmp-tc.my. The objects.defs file must contain enough MIBs to resolve all MIB names, even if they are not used by your subtrees. Then generate the object table files using the following command:

```
# /usr/sbin/snmpi objects.defs subtree
```

The snmpi program has a print option that allows you to dump the contents of the entire tree generated as a result of the objects it finds into the objects.defs file. If you are having trouble with the subtrees you may find this to be helpful. Use the following command to generate a listing:

```
# /usr/sbin/snmpi -p objects.defs > objects.txt
```

The snmpi program outputs the *subtree*_tbl.c and *subtree*_tbl.h; *subtree* is the name of the base MIB variable name for a MIB subtree. These two files are C code used to initialize the MIB object table for the specified subtree. (This is the local object table.) Repeat this process for each MIB subtree being implemented in your subagent. Note that the snmpi program defaults to using MIB groups as the level of granularity for method routines; that is, the assumption is made that all MIB variables within a group should be serviced by the same method routine. (It also provides the *mib-group*_type data structure to help do this.)

The *mib-group*_type structure is not part of the API; it is provided as a convenience. It is helpful to use the *mib-group* organization of the object table. This is because, generally, those objects are logically related and usually accessed as a group; for example, ipRoutes are returned more or less complete from the kernel routing tables.

3. Code the method routines and the API calls.

   Write the code that calls the eSNMP library API to initialize communications with the master agent (snmpd), and register your MIBs. (See Section 6.2.4.)

   Write the code for the required method routines. (See Section 6.3.) Usually you need one Get method routine and one Set method routine for each MIB group within your registered MIB subtree. The *subtree*_tbl.h files generated in the previous step define the names and function prototype for each method routine you need.

4. Build the subagent.

   An example Makefile, chess.mk, is provided in the /usr/examples/esnmp directory.

5. Execute and test your subagent.

   Run your subagent like any other program or daemon. There are trace facilities built into the eSNMP library routines to assist in the debugging process. Use the set_debug_level routine in the main section to enable the trace.

   Once the subagent has initialized and successfully registered a MIB subtree, you can send SNMP requests using standard applications. For example, Compaq Insight Manager, HP Openview, or any MIB browser. If you do not have access to SNMP applications, you can use the snmp_request, snmp_traprcv, and snmp_trapsnd programs to help debug subagents.

   Note that if you interactively debug, your subagent will probably cause SNMP requests to timeout.

   Normally all error and warning messages are recorded in the system's daemon log. When running the sample chess subagent and the os_mibs subagent, you specify a trace run-time argument, as follows:

   ```
   os_mibs -trace
   ```

   With the trace option active, the program does not run as a daemon and all trace output goes to stdout; it displays each message that is processed.

   You can use this feature in your own subagents by calling the set_debug_level routine and pass it the TRACE parameter.

   Anything passed in the debug macro is sent to stdout, as follows:

   ```
   ESNMP_LOG ((TRACE, ("message_text \n"));
   ```

   To send everything to the daemon log, call the set_debug_level routine and pass it the WARNING || DAEMON_LOG parameter or the set_debug_level routine and pass it the ERROR || DAEMON_LOG parameter to suppress warning messages.

6.2.4 Subagent Protocol Operations

The eSNMP API provides for autonomous subagents that are not closely tied to the master agent (snmpd). Subagents can be part of other subsystems or products and have primary functions not related to SNMP. For instance, the gated daemon is primarily concerned with Internet routing; however it also functions as a subagent.

In particular, the snmpd daemon does not start or stop any subagent daemons during its startup or shutdown procedures. It also does not maintain any on-disk configuration information about subagents. Whenever the snmpd daemon starts, it has no knowledge of previous subagent or MIB subtree registrations.

Typically all daemons on the operating system system are started or stopped together, as the system changes run levels. But subagents should correctly handle situations where they start before the snmpd daemon, or are running while the snmpd daemon is restarted to reload information from its configuration file. In these situations subagents need to restart the eSNMP protocol as described in the following sections.

6.2.4.1 Order of Operations

The sequence of subagent protocol operations are as follows:

1. Initialization (esnmp_init)
2. Allocate any needed indexes for tables shares across multiple subagents (esnmp_allocate)
3. Registration (esnmp_register [esnmp_register ...])
4. Data communication

   The following loop happens continuously:

   ```
   {
      determine sockets with data pending

      if the eSNMP socket has data pending
         esnmp_poll

      periodically call esnmp_are_you_there as required
      during periods of inactivity
   }
   ```
5. Termination (esnmp_term)

   Note that it is very important that subagents call the esnmp_term function when they are stopping. This enables eSNMP to free system resources being used by the subagent.

The example subagent in the /usr/examples/esnmp directory shows how to code subagent protocol operations.

6.2.4.2 Function Return Values

The eSNMP API function return values indicate to a subagent both the success or failure of the requested operation and the state of the master agent. The following list provides a description of each return value and the indicated subagent actions:

- `ESNMP_LIB_OK`

  The operation was successful.

- `ESNMP_LIB_NO_CONNECTION`

  The connection between the subagent and the master agent could not be initiated. This value is returned by the `esnmp_init` function.

  – Causes — The master agent is not running or is not responding.

  – Action — Restart the protocol by calling the `esnmp_init` function again after a suitable delay.

- `ESNMP_LIB_BAD_ALLOC`

  Cannot allocate the index or indexes. This value is returned by the `esnmp_allocate` function.

  – Causes are as follows:

    □ The `esnmp_init` function has not been successfully called prior to calling the `esnmp_allocate` function.

    □ One of the parameters in the `esnmp_allocate` function is invalid. See the log file to determine which parameter is invalid.

  – Action — Call the `esnmp_allocate` function in the proper sequence and with correct arguments.

- `ESNMP_LIB_BAD_DEALLOC`

  Cannot deallocate the index or indexes. This value is returned by the `esnmp_deallocate` function.

  – Causes are as follows:

    □ The `esnmp_init` function has not been successfully called prior to calling the `esnmp_allocate` function.

    □ One of the parameters in the `esnmp_deallocate` function is invalid. See the log file to determine which parameter is invalid.

  – Action — Call the `esnmp_deallocate` function in the proper sequence and with correct arguments.

- `ESNMP_LIB_LOST_CONNECTION`

  Lost communications with the master agent. This value is returned by the `esnmp_register`, `esnmp_poll`, `esnmp_are_you_there`, `esnmp_unregister`, and `esnmp_trap` functions.

  – Causes — An attempt to send a packet to the master agent's socket failed; this is normally due to the master agent terminating abnormally.

  - Action — Restart the protocol by calling the esnmp_init function after a suitable delay.

- ESNMP_LIB_BAD_REG

  The attempt to send a registration failed. This value is returned by the esnmp_register, esnmp_unregister, and esnmp_poll . functions.

  - Causes are as follows:
    - □ The esnmp_init function has not been successfully called prior to calling the esnmp_register function.
    - □ The *timeout* parameter in the esnmp_register function is invalid.
    - □ The subtree passed to the esnmp_register function has already been queued for registration or has been registered by this subagent.
    - □ A previous registration was failed by the master agent (when returned by the esnmp_poll function). See the log file to determine the details regarding why it failed and which subtree was at fault.
    - □ Trying to unregister a subtree that was not registered (esnmp_unregister).
  - Action — Call the esnmp_register function in the proper sequence and with correct arguments.

- ESNMP_LIB_CLOSE

  The master agent is stopping. This value is returned by the esnmp_poll function.

  - Causes — The master agent is beginning an orderly shutdown.
  - Action — Restart the protocol with the esnmp_init function as suited by the subagent.

- ESNMP_LIB_NOTOK

  An eSNMP protocol error occurred and the packet was discarded. This value is returned by the esnmp_poll and esnmp_trap functions.

  - Causes — This indicates a packet-level protocol error within eSNMP, probably due to lack of memory resources within the subagent.
  - Action — Continue.

6.3 Extensible SNMP Application Programming Interface

The following sections provide detailed information on the SNMP Application Programming Interface, which consists of the following:

- Calling interface
- Method routine calling interface
- The `libesnmp` support routines

6.3.1 Calling Interface

The calling interface contains the following routines:

- `esnmp_init`
- `esnmp_allocate`
- `esnmp_deallocate`
- `esnmp_register`
- `esnmp_register2`
- `esnmp_unregister`
- `esnmp_unregister2`
- `esnmp_capabilities`
- `esnmp_uncapabilities`
- `esnmp_poll`
- `esnmp_are_you_there`
- `esnmp_trap`
- `esnmp_term`
- `esnmp_sysuptime`

6.3.1.1 The esnmp_init Routine

The `esnmp_init` routine locally initializes the eSNMP subagent, and initiates communication with the master agent.

This call does not block waiting for a response from the master agent. After calling the `esnmp_init` routine, call the `esnmp_register` routine for each MIB subtree that is to be handled by this subagent.

Call this routine during program initialization or to restart the eSNMP protocol. If you are restarting, the `esnmp_init` routine clears all registrations so each subtree must be reregistered.

You should attempt to create a unique *`subagent_identifier`*, perhaps using the program name (`argv[0]`) and additional descriptive text.

The syntax for the `esnmp_init` routine is as follows:

```
int esnmp_init(
         int *socket,
        char *subagent_identifier);
```

The arguments are defined as follows:

socket

The address of the integer that receives the socket descriptor used by eSNMP.

subagent_identifier

The address of a null-terminated string that identifies this subagent (usually program name).

The return values are as follows:

ESNMP_LIB_NO_CONNECTION

Could not initialize or communicate with the master agent. Try again after a delay.

ESNMP_LIB_OK

The esnmp_init routine has completed successfully.

ESNMP_LIB_NOTOK

Could not allocate memory for the subagent.

The following is an example of the esnmp_init routine:

```
#include <esnmp.h>
int socket;
status = esnmp_init(&socket, "gated");
```

6.3.1.2 The esnmp_allocate Routine

The esnmp_allocate routine requests the allocation of a value for one or more specific index objects.

Index allocation is a service provided by an AgentX-compliant master agent. It provides generic support for sharing MIB conceptual tables among subagents who are assumed to have no knowledge of each other. The master agent maintains a database of index objects (OIDs) and the values that have been allocated for each index. The master agent is unaware of what MIB variables (if any) the index objects represent. By convention, subagent developers should use the MIB variable listed in the INDEX clause as the index object for which values must be allocated.

For tables indexed by multiple variables, values may be allocated for each index although this is frequently unnecessary. The subagent may request one of the following types of allocation:

- A specific index value
- An index value that is not currently allocated
- An index value that has never been allocated

The last two alternatives reflect the uniqueness and constancy requirements present in many MIB specifications for arbitrary integer indexes; for example, `ifIndex` in the IF-MIB (RFC 2233), `snmpFddiSMTIndex` in the FDDI MIB (RFC 1285), and `sysApplInstallPkgIndex` in the System Application MIB (RFC 2287). The need for subagents to share tables using such indexes is the main motivation for an index allocation mechanism.

Index allocation and MIB region registration are not coupled in the master agent. The master agent does not consider its current state of index allocations when processing registration requests and does not consider its current registry when processing index allocation requests. This is mainly to accommodate non-AgentX subagents.

Subagent developers should request the allocation of an index first. Then, they should register the corresponding region. When done in this order, a successful index allocation request gives a subagent a good hint (but no guarantee) of what it should be able to register. The registration might fail because some other subagent has already registered that row of the table.

Subagents should register conceptual rows in a shared table in the following order:

1. Allocate an index value successfully.
2. Use the allocated index value or values in the *`instance`* field in the `ESNMP_REG` structure that is passed to the `esnmp_register2` routine. In this way, the eSNMP subagent developer can fully qualify the MIB region or regions specified by the subtree and any *`range_subid`* and *`range_upper_bound`* fields in that `ESNMP_REG` structure. Subagent developers should set the *`priority`* field to 255 when attempting registrations with instances.
3. If the registration fails with a result code of ESNMP_REG_STATE_REGDUP, deallocate the previously allocated index value or values for this row with the `esnmp_deallocate` routine and begin the process again at step 1.

Note that index allocation is necessary only when an index is an arbitrary value and the subagent developer cannot determine which index values to use. When index values have intrinsic meaning, subagents should not allocate their index values. For example, in RFC 1514 the table of running software processes (hrSWRunTable) is indexed by the system's native process identifier (pid). A subagent implementing the row of hrSWRunTable corresponding to its own process would register the region defining that row's object instances; it is not necessary to allocate index values.

The syntax for the esnmp_allocate routine is as follows:

```
int esnmp_allocate(
        ESNMP_ALLOC *alloc_parm);
```

The arguments are as follows:

alloc_parm

Is a pointer to a ESNMP_ALLOC structure. The caller must keep this structure and its referenced VARBIND list persistent (in memory). This is necessary in order for the eSNMP runtime library to update fields with the result of the index allocation request provided by the master agent. The structure contains the following fields:

vb

A pointer to a variable binding list containing one or more VARBINDs. Each VARBIND in the list contains the name of an index object to be allocated a value. You must always supply a value for each varbind.

When neither the ESNMP_ALLOC_ANY_INDEX nor the ESNMP_ALLOC_NEW_INDEX flags are specified, the master agent uses all supplied values when processing the index allocation request. When either the ESNMP_ALLOC_ANY_INDEX or the ESNMP_ALLOC_NEW_INDEX flag is specified, the master agent ignores all supplied values and generates an appropriate value for each VARBIND when processing the index allocation request.

options

A bitmask of the following values:

ESNMP_ALLOC_CLUSTER

The index allocation request is for the cluster context.

ESNMP_ALLOC_NEW_INDEX

This index allocation request is for a new value. The master agent will generate a value for each VARBIND that has not been used since it started running. These values will be passed back in the successful response.

ESNMP_ALLOC_ANY_INDEX

This index allocation request is for an unused value. The master agent will generate a value for each `VARBIND` that may or may not have been used since it started running. These values will be passed back in the successful response.

`status`

One of the following integer values that provides the caller with asynchronous updates of the state of the index allocation request. After the return of the `esnmp_poll` routine, the caller can inspect this parameter. For the following status codes, the `alloc->error_index` field contains a value of zero (0):

ESNMP_ALLOC_STATE_PENDING

The index allocation request is currently held locally while waiting for a connection to the master agent to become established.

ESNMP_ALLOC_STATE_SENT

The routine sent the index allocation request to the master agent (final status still pending).

ESNMP_ALLOC_STATE_DONE

The master agent successfully processed and acknowledged the index allocation request. The subagent can now use the allocated index value or values in the `esnmp_reg->instance` field that is passed to the `esnmp_register2` call.

For the following status codes, the `alloc->error_index` field contains a non-zero value:

ESNMP_ALLOC_STATE_ALLOCTYPE

The master agent rejected the index allocation request because of a wrong index type or wrong options.

ESNMP_ALLOC_STATE_ALLOCINUSE

The master agent rejected the index allocation request because the supplied index object value or values are currently in use.

ESNMP_ALLOC_STATE_ALLOCAVAIL

The master agent rejected the index allocation request because it had no available value or values for the requested index object. This status is returned only if you specify the ESNMP_ALLOC_NEW_INDEX or ESNMP_ALLOC_ANY_INDEX option.

ESNMP_ALLOC_STATE_ALLOCNOCLU

The master agent rejected the index allocation request because the cluster context option is not supported.

ESNMP_ALLOC_STATE_REJ

The master agent rejected the index allocation request because of other reasons.

error_index

Identifies the VARBIND (starting from 1) in the alloc.vb varbind list to which the status code applies. After the return of the esnmp_poll routine, the caller can inspect this parameter.

The return values are as follows:

ESNMP_LIB_NOTOK

The *alloc_parm* argument was not specified.

ESNMP_LIB_OK

The esnmp_allocate routine has completed successfully.

ESNMP_LIB_LOST_CONNECTION

The subagent lost communication with the master agent.

ESNMP_LIB_BAD_ALLOC

The subagent has not established a connection with the master agent or no valid VARBIND list was dereferenced by the *alloc_parm* argument. A message is also in the log file.

Note that the return value indicates only initiation of an index allocation request. The actual status code returned in the master agent's response will be returned in a subsequent call to the esnmp_poll routine in the alloc->status and alloc->error_index fields.

The following is an example of the esnmp_allocate routine:

```
#define  INDENT_SIZE             4
#define RESPONSE_TIMEOUT         0  /* use the default time set
                                       in esnmp_init message */
#define REGISTRATION_PRIORITY  128  /* priority at which the MIB
                                      subtree will register */
#define RANGE_SUBID             10  /* the identifier position in
                                       oid->elements just after ifEntry */
#define RANGE_UPPER_BOUND       22  /* the identifier for ifSpecific, under ifEntry */

int rc, status;
unsigned int our_ifIndex_instance = 0;
int ready_to_register        = 0;
int have_a_good_registration = 0;
extern SUBTREE ifEntry_subtree;     /* generated by /usr/sbin/snmpi -r ifEntry  */
OBJECT  *ifIndex_object = &ifEntry_subtree.object_tbl[I_ifIndex];
static ESNMP_ALLOC esnmp_alloc_for_ifIndex;  /* retain this structure to obtain status
                                                code and index object values.  Also,
                                                retain this structure for a subsequent
                                                call to esnmp_deallocate  */
static ESNMP_REG esnmp_reg_for_ifEntry;      /* retain this structure for a subsequent
                                                call to esnmp_unregister2  */
static OID ifEntry_instance_oid;
VARBIND  *vb;
\*
 *  initialize the ESNMP_ALLOC structure
 *\
memset(&esnmp_alloc_for_ifIndex, 0, sizeof(ESNMP_ALLOC));

esnmp_alloc_for_ifIndex.options = ESNMP_ALLOC_NEW_INDEX;
esnmp_alloc_for_ifIndex.vb = vb = (VARBIND *)malloc(sizeof(VARBIND));
bzero((char *)vb, sizeof(VARBIND));
clone_oid(&vb->name, &ifIndex_object->oid);
o_integer(vb, ifIndex_object, (unsigned long)0); /* master will return actual
                                                    value assigned */
while(!have_a_good_registration) {
   status = esnmp_allocate(&esnmp_alloc_for_ifIndex );
   if (status != ESNMP_LIB_OK) {
      printf("Could not queue the 'ifIndex' \n");
      printf("index object for index allocation\n")
   }
   :
   :
   rc = esnmp_poll();
   :
   :
   if (esnmp_alloc_for_ifIndex.status > ESNMP_ALLOC_STATE_SENT) {
       /*
        * esnmp_alloc_for_ifIndex.status nows contain the final
        * status from the master agent.
        */
      switch(esnmp_alloc_for_ifIndex.status) {
         case ESNMP_ALLOC_STATE_DONE
            our_ifIndex_instance = esnmp_alloc_for_ifIndex.vb->value.ul;
            ready_to_register = 1;
            printf("\n*** Successful index allocation. Our conceptual row in the");
            printf("\n*** interfaces table was allocated. ifIndex value is %i\n\n",
                        our_ifIndex_instance);
            break;
         case ESNMP_ALLOC_STATE_ALLOCTYPE:
            printf("\n*** Failed index allocation - 'allocation type'");
            printf("\n*** associated with supplied varbind #%i.\n\n",
                          esnmp_alloc_for_ifIndex.error_index);
            break;
         case ESNMP_ALLOC_STATE_ALLOCINUSE:
            printf("\n*** Failed index allocation - 'allocation in use'");
            printf("\n*** associated with supplied varbind #%i.\n\n",
                          esnmp_alloc_for_ifIndex.error_index);
            break;
         case ESNMP_ALLOC_STATE_ALLOCAVAIL:
            printf("\n*** Failed index allocation - 'no available values'");
            printf("\n*** associated with supplied varbind #%i.\n\n",
                          esnmp_alloc_for_ifIndex.error_index);
            break;
         case ESNMP_ALLOC_STATE_ALLOCNOCLU:
            printf("\n*** Failed index allocation - 'cluster context not supported'");
            printf("\n*** associated with supplied varbind #%i.\n\n",
                          esnmp_alloc_for_ifIndex.error_index);
            esnmp_alloc_for_ifIndex.options &= ~ESNMP_ALLOC_CLUSTER;
            break;
         case ESNMP_ALLOC_STATE_REJ:
            printf("\n*** failed index allocation - 'other reasons'");
            printf("\n*** associated with supplied varbind #%i.\n\n",
                          esnmp_alloc_for_ifIndex.error_index);
            break;
      }  /*  End switch */
   }  /*  End if */
   if (ready_to_register) {
      vb = esnmp_alloc_for_ifIndex.vb;
      memset(&esnmp_reg_for_ifEntry, 0, sizeof(ESNMP_REG));
      esnmp_reg_for_ifEntry.subtree           = &ifEntry_subtree;
      esnmp_reg_for_ifEntry.priority          = REGISTRATION_PRIORITY;
      esnmp_reg_for_ifEntry.timeout           = RESPONSE_TIMEOUT;
      esnmp_reg_for_ifEntry.range_subid       = RANGE_SUBID;
      esnmp_reg_for_ifEntry.range_upper_bound = RANGE_UPPER_BOUND;

      ifEntry_instance_oid.nelem              = 1;
      ifEntry_instance_oid.elements           = &our_ifIndex_instance;
      esnmp_reg_for_ifEntry.instance          = (OID *)malloc(sizeof(OID));
      esnmp_reg_for_ifEntry.instance     = clone_oid(esnmp_reg_for_ifEntry.instance,
                                                &ifEntry_instance_oid);
      status = esnmp_register2(&esnmp_reg_for_ifEntry);
      if (status != ESNMP_LIB_OK) {
         printf("Could not queue the registration for 'ifEntry'\n");
      }
      else {
         .
         .
         .
         rc = esnmp_poll();
         .
         .
         .
         if (esnmp_reg_for_ifEntry.state > ESNMP_REG_STATE_SENT) {
             /*
              * esnmp_reg_for_ifEntry.status nows contain the final
              * status from the master agent.
              */
            switch(esnmp_reg_for_ifEntry.state) {
               case ESNMP_REG_STATE_DONE:
                   printf("\n*** Successful registration for conceptual row in");
                   printf("\n*** the interfaces table indexed by an ifIndex");
                   printf("\n*** value of %i.\n\n", our_ifIndex_instance);
                   have_a_good_registration = 1;
                   break;
               case ESNMP_REG_STATE_REGDUP:
                   printf("\n*** Failed registration - Duplicate registration.");
                   printf("\n*** We need to deallocate this ifIndex value and";
                   printf("\n*** allocate a new value\n\n");
                   break;
               case ESNMP_REG_STATE_REGNOCLU:
                   printf("\n*** Failed registration - Cluster context not supported.");
                   printf("\n*** Need to deallocate this ifIndex value and to");
```

```
                   printf("\n*** allocate a new value in the default context\n\n");
                   esnmp_alloc_for_ifIndex.options &= ~ESNMP_ALLOC_CLUSTER;
                   break;
               case ESNMP_REG_STATE_REJ:
                   printf("\n*** Failed registration - Other reasons\n\n");
                   break;
            }
            if (!have_a_good_registration) {
               esnmp_deallocate(&esnmp_alloc_for_ifIndex);
               ready_to_register = 0;
               rc = esnmp_poll();
               free_oid(esnmp_reg_for_ifEntry.instance);
               free(esnmp_reg_for_ifEntry.instance);
               esnmp_reg_for_ifEntry.instance = NULL;
            }
         }
      }
   }
}
```

6.3.1.3 The esnmp_deallocate Routine

The `esnmp_deallocate` routine requests the master agent to deallocate a value or values previously allocated to this subagent for one or more index objects.

Upon receiving an index deallocation request, the master agent updates its database of index objects (OIDs) by marking the specified values as deallocated. These released values are considered available for assignment by subsequent index allocation requests.

The syntax for the `esnmp_deallocate` routine is as follows:

```
int esnmp_deallocate(
            ESNMP_ALLOC *alloc_parm);
```

The arguments are as follows:

`alloc_parm`

Is a pointer to an `ESNMP_ALLOC` structure. Typically, this is the same structure used in a previous call to `esnmp_allocate`. The caller must keep this structure and its referenced `VARBIND` list persistent (in memory). This is necessary in order for the eSNMP runtime library to update fields with the result of the index allocation request provided by the master agent. The structure contains the following fields:

`vb`

A pointer to a variable binding list containing one or more `VARBIND`s. Each `VARBIND` in the list contains the name and the value of an index object previously allocated.

`options`

A bitmask of the following values:

ESNMP_ALLOC_CLUSTER

The index allocation request is for the cluster context.

status

One of the following integer values that provides the caller with asynchronous updates of the state of the index deallocation request. After the return of the `esnmp_poll` routine, the caller can inspect this parameter. For the following status codes, the `alloc->error_index` field contains a value of zero (0):

ESNMP_ALLOC_STATE_SENT

The index deallocation request was sent to the master agent (final status still pending).

ESNMP_ALLOC_STATE_DONE

The master agent successfully processed and acknowledged the index deallocation request. The master agent can now reuse the index value or values when processing a subsequent index allocation request.

For the following status codes, the `alloc->error_index` filed contains a non-zero value:

ESNMP_ALLOC_STATE_DEALLOC_REJ

The master agent rejected the index deallocation request because of an unknown allocation.

ESNMP_ALLOC_STATE_REJ

The master agent rejected the index deallocation request because of other reasons.

error_index

Identifies the `VARBIND` (starting from 1) in the `alloc.vb` list to which the status code applies. After the return of the `esnmp_poll` routine, the caller can inspect this parameter.

The return values are as follows:

ESNMP_LIB_NOTOK

The *alloc_parm* argument was not specified.

ESNMP_LIB_OK

The `esnmp_allocate` routine has completed successfully.

ESNMP_LIB_LOST_CONNECTION

The subagent lost communication with the master agent.

ESNMP_LIB_BAD_DEALLOC

The subagent has not established a connection with the master agent or no valid VARBIND list was dereferenced by the *alloc_parm* argument. See the log file.

Note that the return value indicates only initiation of an index deallocation request. The actual status code returned in the master agent's response will be returned in a subsequent call to the esnmp_poll routine in the alloc->status and alloc->error_index fields.

The following is an example of the esnmp_deallocate routine:

```
#include <esnmp.h>

int status;
static ESNMP_ALLOC esnmp_alloc_for_ifIndex;  /* structure retained from a previous call
                                                to esnmp_allocation().  Retain this
                                                structure for update with final status
                                                from the master agent */
    /* call to unregister2()  goes here */
esnmp_alloc_for_ifIndex.options = 0;   /* clear options */
status = esnmp_deallocate( &esnmp_alloc_for_ifIndex );
if (status != ESNMP_LIB_OK) {
    printf("Could not queue the 'ifIndex' \n");
    printf("index object for index allocation\n");
}
.
.
.

esnmp_poll();
.
.
.

if (esnmp_alloc_for_ifIndex.status > ESNMP_ALLOC_STATE_SENT) {
    /*
     * the final status from the master agent is available
     */
switch(esnmp_alloc_for_ifIndex.status) {
    case ESNMP_ALLOC_STATE_DONE:
        printf("Successful index deallocation for value(s) associated\n");
        printf("with the 'ifIndex' index object.\n");
        free_varbind(esnmp_alloc_forifIndex.vb);
        break;
    case ESNMP_ALLOC_STATE_DEALLOC_REJ:
        printf("Failed index deallocation due to 'unknown allocation'\n");
        printf("associated with supplied varbind #%i.\n",
                                     esnmp_alloc_for_ifIndex.error_index);
        break;
    case ESNMP_ALLOC_STATE_REJ:
        printf("Failed index deallocation due to 'other reasons'\n");
        printf("associated with supplied varbind #%i.\n",
                                     esnmp_alloc_for_ifIndex.error_index);
        break;
    }
}
```

6.3.1.4 The esnmp_register Routine

The esnmp_register routine requests registration of a single MIB subtree. This indicates to the master agent that the subagent instantiates MIB variables within the registered MIB subtree.

The initialization routine (esnmp_init) must be called prior to calling the esnmp_register routine. The esnmp_register function must be called for each SUBTREE structure corresponding to each MIB subtree that it will be handling. At any time MIB subtrees can be unregistered by calling esnmp_unregister and then be reregistered by calling esnmp_register.

When restarting the eSNMP protocol by calling esnmp_init, all MIB subtree registrations are cleared. All MIB subtrees must be reregistered.

A MIB subtree is identified by the base MIB variable name and its corresponding OID. This tuple represents the parent of all MIB variables that are contained in the MIB subtree; for example, the MIB-2 tcp subtree has an OID of 1.3.6.1.2.1.6. All MIB variables subordinate to this (those that have the same first 7 identifiers) are included in the subtree's region. A MIB subtree can also be a single MIB variable (a leaf node) or even a specific instance.

By registering a MIB subtree, the subagent indicates that it will process SNMP requests for all MIB variables (or OIDs) within that MIB subtree's region. Therefore, a subagent should register the most fully qualified (longest) MIB subtree that still contains its instrumented MIB variables.

The master agent requires that a subagent cannot register the same MIB subtree more than once. Other than this one restriction, a subagent may register MIB subtrees that overlap the OID range of MIB subtrees that it previously registered or those of MIB subtrees registered by other subagents.

For example, consider the two Tru64 UNIX daemons, os_mibs and gated. The os_mibs daemon registers the ip MIB subtree (1.3.6.1.2.1.4) and the gated daemon registers the ipRouteEntry MIB subtree (1.3.6.1.2.1.4.21.1). Requests for ip MIB variables within ipRouteEntry, such as ipRouteIfIndex (1.3.6.1.2.1.4.21.1.2), are passed to the gated subagent. Requests for other ip variables, such as ipNetToMediaIfIndex (1.3.6.1.2.1.4.22.1.1), are passed to the os_mibs subagent. If the gated process should terminate or unregister the ipRouteEntry MIB subtree, subsequent requests for ipRouteIfIndex will go to the os_mibs subagent because the ip MIB subtree, which includes all ipRouteEntry MIB variables, would now be the authoritative region of requests for ipRouteIfIndex.

When the master agent receives a SIGUSR1 signal, it puts its MIB registry in to the /var/tmp/snmpd_dump.log file. See snmpd(8) for more information.

The syntax for the `esnmp_register` routine is as follows:

```
int esnmp_register(
        SUBTREE *subtree,
        int timeout,
        int priority);
```

The arguments are defined as follows:

subtree

A pointer to a `SUBTREE` structure corresponding to the MIB subtree to be handled. The `SUBTREE` structures are externally declared and initialized in the code emitted by the `mosy` and `snmpi` utilities (*xxx*`_tbl.c` and *xxx*`_tbl.h`, where *xxx* is the name of the MIB subtree) taken directly from the MIB document.

Note

> All memory pointed to by the *subtree* fields must have permanent storage since it is referenced by `libesnmp` for the duration of the program. You should use the data declarations emitted by the `snmpi` utility.

timeout

The number of seconds the master agent should wait for responses when requesting data in this MIB subtree. This value must be between zero (0) and three hundred (300). If the value is zero (0), the default timeout is used (3 seconds). You should use the default.

priority

The registration priority. The entry with the largest number has the highest priority. The range is 1 to 255. The subagent that has registered a MIB subtree that has the highest priority over a range of Object Identifiers (OIDs) gets all requests for that range of OIDs.

MIB subtrees that are registered with the same priority are considered duplicates, and the registration is rejected by the master agent.

The *priority* argument is a mechanism for cooperating subagents to handle different configurations.

The return values are as follows:

ESNMP_LIB_OK

The `esnmp_register` routine has completed successfully.

ESNMP_LIB_BAD_REG

> The esnmp_init routine has not been called, the timeout parameter is invalid, or this MIB subtree has already been queued for registration.

ESNMP_LIB_LOST_CONNECTION

> The subagent lost communication with the master agent.

Note that the status indicates only the initiation of the request. The actual status returned in the master agent's response will be returned in a subsequent call to the esnmp_poll routine.

The following is an example of the esnmp_register routine:

```
#include <esnmp.h>
#define RESPONSE_TIMEOUT       0  /* use the default time set
                                     in esnmp_init message */
#define REGISTRATION_PRIORITY 10  /* priority at which subtrees
                                     will register */
int status;

extern SUBTREE ipRouteEntry_subtree;

status = esnmp_register( &ipRouteEntry_subtree,
                         RESPONSE_TIMEOUT,
                         REGISTRATION_PRIORITY );
if (status != ESNMP_LIB_OK) {
    printf ("Could not queue the 'ipRouteEntry' \n");
    printf ("subtree for registration\n");
}
```

6.3.1.5 The esnmp_unregister Routine

The esnmp_unregister routine unregisters a MIB subtree with the master agent.

This routine can be called by the application code to tell the eSNMP subagent not to process requests for variables in this MIB subtree anymore. You can later reregister a MIB subtree, if needed, by calling the esnmp_register routine.

The syntax for the esnmp_unregister routine is as follows:

```
int esnmp_unregister(
         SUBTREE *subtree);
```

The arguments are as follows:

subtree

A pointer to the SUBTREE structure for the MIB subtree to be unregistered.

The return values are as follows:

ESNMP_LIB_OK

The routine completed successfully.

ESNMP_LIB_BAD_REG

The MIB subtree was not registered.

ESNMP_LIB_LOST_CONNECTION

The request to unregister the MIB subtree could not be sent. You should restart the protocol.

The following is an example of the esnmp_unregister routine:

```
#include <esnmp.h>
int status

extern SUBTREE ipRouteEntry_subtree;

status = esnmp_unregister( &ipRouteEntry_subtree );

switch (status) {
   case ESNMP_LIB_OK:
    printf ("The esnmp_unregister routine completed successfully.\n");
    break;

   case ESNMP_LIB_BAD_REG:
    printf ("The MIB subtree was not registered.\n");
    break;

   case ESNMP_LIB_LOST_CONNECTION:
   printf("%s%s%s\n", "The request to unregister the ",
                      "MIB subtree could not be sent.  ",
                      "You should restart the protocol.\n");
   break;
}
```

6.3.1.6 The esnmp_register2 Routine

The esnmp_register2 routine offers extensions to the esnmp_register routine.

The initialization routine (esnmp_init) must be called prior to calling the esnmp_register2 routine. The esnmp_register2 function must be called for each subtree structure corresponding to each MIB subtree that it will be handling. At any time MIB subtrees can be unregistered by calling esnmp_unregister2 and then be reregistered by calling esnmp_register2.

When restarting the eSNMP protocol by calling esnmp_init, all MIB subtree registrations are cleared. All MIB subtrees must be reregistered.

A MIB subtree is identified by the base MIB variable name and its corresponding OID. This tuple represents the parent of all MIB variables that are contained in the MIB subtree; for example, the MIB-2 tcp subtree has an OID of 1.3.6.1.2.1.6. All elements subordinate to this (those that have the same first 7 identifiers) are included in the subtree's object table. A MIB subtree can also be a single MIB object (a leaf node) or even a specific instance.

By registering a MIB subtree, the subagent indicates that it will process SNMP requests for all MIB variables (or OIDs) within that MIB subtree's region. Therefore, a subagent should register the most fully qualified (longest) MIB subtree that still contains its instrumented MIB variables.

A subagent using the esnmp_register2 routine can register the same MIB subtree for the local node and for a cluster. To register the MIB subtree for both, you must call the esnmp_register2 routine twice: once with the ESNMP_REG_OPT_CLUSTER bit set in the *options* parameter and once with the ESNMP_REG_OPT_CLUSTER bit clear in the *options* parameter. Alternatively, you can register a MIB subtree for the cluster only or for the local node only, by setting or clearing the ESNMP_REG_OPT_CLUSTER bit, respectively, in the *options* parameter.

A subagent may also register MIB subtrees that overlap the OID range of MIB subtrees that it previously registered or those of MIB subtrees registered by other subagents.

For example, consider the two Tru64 UNIX daemons, os_mibs and gated. The os_mibs daemon registers the ip MIB subtree (1.3.6.1.2.1.4) and the gated daemon registers the ipRouteEntry MIB subtree (1.3.6.1.2.1.4.21.1). Both of these registrations are made with the ESNMP_REG_OPT_CLUSTER bit set in the *options* parameter. Requests for ip MIB variables within ipRouteEntry, such as ipRouteIfIndex (1.3.6.1.2.1.4.21.1.2), are passed to the gated subagent. Requests for other ip variables, such as ipNetToMediaIfIndex (1.3.6.1.2.1.4.22.1.1), are passed to the os_mibs subagent. If the gated process should terminate or unregister the ipRouteEntry MIB subtree, subsequent requests for ipRouteIfIndex will go to the os_mibs subagent because the ip MIB subtree, which includes all `ipRouteEntry` MIB variables, would now be the authoritative region of requests for `ipRouteIfIndex`.

The syntax for the `esnmp_register2` routine is as follows:

```
int esnmp_register2(
        ESNMP_REG *reg);
```

The arguments are defined as follows:

reg

A pointer to a `ESNMP_REG` structure, which contains the following fields:

subtree

A pointer to a `SUBTREE` structure corresponding to the MIB subtree to be handled. The `SUBTREE` structures are externally declared and initialized in the code emitted by the `mosy` and `snmpi` utilities (*xxx*`_tbl.c` and *xxx*`_tbl.h`, where *xxx* is the name of the MIB subtree) taken directly from the MIB document.

Note

All memory pointed to by this field must have permanent storage since it is referenced by `libesnmp` for the duration of the program. You should use the data declarations emitted by the `snmpi` utility.

priority

The registration priority. The entry with the largest number has the highest priority. The range is 1 to 255. The subagent that has registered a MIB subtree that has the highest priority over a range of Object Identifiers (OIDs) gets all requests for that range of OIDs.

MIB subtrees that are registered with the same priority are considered duplicates, and the registration is rejected by the master agent.

The *priority* field is a mechanism for cooperating subagents to handle different configurations.

timeout

The number of seconds the master agent should wait for responses when requesting data in this MIB subtree. This value must be between zero (0) and three hundred (300). If the value is zero (0), the default timeout is used (3 seconds). You should use the default.

range_subid

An integer value that when non-zero and together with the *range_upper_bound* field specifies a range instead of one of the MIB subtree's OID sub-identifiers. The *range_subid* field specifies the OID sub-identifier modified by the *range_upper_bound* field.

range_upper_bound

An integer value that, in conjunction with a non-zero *range_subid* field specifies a range instead of one of the MIB subtree's OID sub-identifiers. The *range_upper_bound* field provides the upper bound of the range and the *range_subid* field provides the lower bound of the range, which is the MIB subtree's OID sub-identifier.

options

An integer value that when set to ESNMP_REG_OPT_CLUSTER indicates that the registration is valid cluster-wide and when set to zero indicates that the registration is valid for the local node.

state

One of the following integer values that provides the caller with asynchronous updates of the state of registration of this MIB subtree. After the return of the `esnmp_poll` routine, the caller can inspect this parameter.

ESNMP_REG_STATE_PENDING

The registration is currently held locally while waiting for connection to the master agent.

ESNMP_REG_STATE_SENT

The registration was sent to the master agent.

ESNMP_REG_STATE_DONE

The wegistration was successfully ackowledged by the master agent.

ESNMP_REG_STATE_REGDUP

The registration was rejected by the master agent because it was a duplicate.

ESNMP_REG_STATE_REGNOCLU

The master agent does not support cluster registrations.

ESNMP_REG_STATE_REJ

The master agent rejected the registration for other reasons.

instance

When non-null, this input parameter specifies a partial or fully qualified instance for the MIB subtree or subtrees in the registration. Use this parameter when registering a row in a table. See Section 6.3.1.2 and snmpi(8)for additional information on registering rows in a table.

The return values are as follows:
ESNMP_LIB_OK

The esnmp_register2 routine has completed successfully.

ESNMP_LIB_BAD_REG

The esnmp_init routine has not been called, the timeout parameter is invalid, a registration slot is not available, or this MIB subtree has already been queued for registration. A message is also in the log file.

ESNMP_LIB_LOST_CONNECTION

The subagent lost communication with the master agent.

Note that the status indicates only the initiation of the request. The actual status returned in the master agent's response will be returned in a subsequent call to the esnmp_poll routine in the reg->*state* field.

The following is an example of the esnmp_register2 routine:

```
#include <esnmp.h>
#define RESPONSE_TIMEOUT         0    /* use the default time set
                                         in esnmp_init message */
#define REGISTRATION_PRIORITY 10     /* priority at which the MIB
                                         subtree will register */
#define RANGE_SUBID             7    /* the identifier position in
                                         oid->elements just after
                                         mib-2 */
#define RANGE_UPPER_BOUND       8    /* the identifier for egp,
                                         under mib-2 */
int status;

extern SUBTREE ip_subtree;
```

```
static ESNMP_REG esnmp_reg_for_ip2egp;  /* retain this structure
                                           for a subsequent call to
                                           esnmp_unregister2  */

/*
 *  initialize the ESNMP_REG structure
 */
memset(&esnmp_reg_for_ip2egp, 0, sizeof(ESNMP_REG));
esnmp_reg_for_ip2egp.subtree           = &ip_subtree;
esnmp_reg_for_ip2egp.priority          = REGISTRATION_PRIORITY;
esnmp_reg_for_ip2egp.timeout           = RESPONSE_TIMEOUT;
esnmp_reg_for_ip2egp.range_subid       = RANGE_SUBID;
esnmp_reg_for_ip2egp.range_upper_bound = RANGE_UPPER_BOUND;

status = esnmp_register2( &esnmp_reg_for_ip2egp );
if (status != ESNMP_LIB_OK) {
    printf("Could not queue the 'ipRouteEntry' \n");
    printf("subtree for registration\n");
}
```

6.3.1.7 The esnmp_unregister2 Routine

The `esnmp_unregister2` routine unregisters a MIB subtree with the master agent. Use this routine only when the MIB subtree was registered using the `esnmp_register2` routine.

This routine can be called by the application code to tell the eSNMP subagent not to process requests for variables in this MIB subtree any more. You can later reregister a MIB subtree, if needed, by calling the `esnmp_register2` routine.

The syntax for the `esnmp_unregister2` routine is as follows:

```
int esnmp_unregister2(
         ESNMP_REG *reg);
```

The arguments are as follows:

reg

A pointer to the `ESNMP_REG` structure that was used when the `esnmp_register2` routine was called.

The return values are as follows:

ESNMP_LIB_OK

The routine completed successfully.

ESNMP_LIB_BAD_REG

The MIB subtree was not registered.

ESNMP_LIB_LOST_CONNECTION

> The request to unregister the MIB subtree could not be sent. You should restart the protocol.

The following is an example of the `esnmp_unregister2` routine:

```
#include <esnmp.h>
int status

extern ESNMP_REG esnmp_reg_for_ip2egp;

status = esnmp_unregister2( &esnmp_reg_for_ip2egp );

switch(status) {
   case ESNMP_LIB_OK:
   printf("The esnmp_unregister2 routine completed successfully.\n");
   break;

   case ESNMP_LIB_BAD_REG:
   printf("The MIB subtree was not registered.\n");
   break;

   case ESNMP_LIB_LOST_CONNECTION:
   printf("%s%s%s\n", "The request to unregister the ",
                      "MIB subtree could not be sent.  ",
                      "You should restart the protocol.\n");
   break;
}
```

6.3.1.8 The esnmp_capabilities Routine

The `esnmp_capabilities` routine adds a subagent's capabilities to the master agent's sysORTable. The sysORTable is a conceptual table that contains an agent's object resources, and is described in RFC 1907.

This routine is called at any point after initializing eSNMP by a call to the `esnmp_init` routine.

The syntax for the `esnmp_capabilities` routine is as follows:

```
void esnmp_capabilities(
         OID *agent_cap_id,
         char *agent_cap_descr);
```

The arguments are as follows:

agent_cap_id

> A pointer to an object identifier that represents an authoritative agent capabilities identifier. This value is used for the sysORID object in the sysORTable for the managed node.

agent_cap_descr

A pointer to a null-terminated character string describing *agent_cap_id*. This value is used for the sysORDescr object in the sysORTable for the managed node.

6.3.1.9 The esnmp_uncapabilities Routine

The esnmp_uncapabilities routine removes a subagent's capabilities from the master agent's sysORTable.

This routine is called if a subagent alters its capabilities dynamically. When a logical connection for a subagent is closed, the master agent removes the related entries in sysORTable.

The syntax for the esnmp_uncapabilities routine is as follows:

```
void esnmp_uncapabilities(
         OID *agent_cap_id);
```

The arguments are as follows:

agent_cap_id

A pointer to an object identifier of the agent capabilities statement to be removed from the sysORTable.

6.3.1.10 The esnmp_poll Routine

The esnmp_poll routine processes a pending message that has been sent by the master agent. This routine is called after the user's select() call has indicated data is ready on the eSNMP socket. (This socket was returned from the call to the esnmp_init routine). If no message is pending on the socket, the esnmp_poll routine blocks until one is received.

If a received message indicates a problem, the routine makes an entry in the syslog file and returns an error status.

If the received message is a request for SNMP data, the routine consults the object table and calls the appropriate method routine or routines.

The syntax for the esnmp_poll routine is as follows:

```
int esnmp_poll( void );
```

The return values are as follows:

ESNMP_LIB_OK

The esnmp_poll routine has completed successfully.

ESNMP_LIB_BAD_REG

A previous registration was failed by the master agent. See the log file.

ESNMP_LIB_DUPLICATE

A duplicate subagent identifier has already been received by the master agent. This is an `esnmp_init` error.

ESNMP_LIB_NO_CONNECTION

The master agent failed to initiate an `esnmp_init` request. Restart after a delay. See the log file.

ESNMP_LIB_CLOSE

A CLOSE message was received.

ESNMP_LIB_NOTOK

An eSNMP protocol error occurred. The packet was discarded.

ESNMP_LIB_LOST_CONNECTION

Communication with master agent was lost. Restart the connection.

6.3.1.11 The esnmp_are_you_there Routine

The `esnmp_are_you_there` routine requests the master agent to report immediately that it is up and functioning. This call does not block waiting for a response. The response is processed by calling the `esnmp_poll` routine.

If no response is received within the timeout period, the application code should restart the eSNMP protocol by calling the `esnmp_init` routine. There are no timers maintained by the eSNMP library.

The syntax for the `esnmp_are_you_there` routine is as follows:

```
int esnmp_are_you_there( void );
```

The return values are as follows:

ESNMP_LIB_OK

The request was sent.

ESNMP_LIB_LOST_CONNECTION

Cannot send the request because the master agent is down.

6.3.1.12 The esnmp_trap Routine

The `esnmp_trap` routine sends a trap message to the master agent. This function can be called at anytime. If the master agent is not running the eSNMP protocol, traps are queued and sent when communication is possible.

The trap message is actually sent to the master agent after the master agent's response to the `esnmp_init` call has been processed. This processing happens within any API call, in most cases during subsequent calls to the `esnmp_poll` routine. The quickest way to send traps to the master agent is to call the `esnmp_init`, `esnmp_poll`, and `esnmp_trap` routines.

The master agent formats the trap into an SNMP trap message and sends it to management stations based on its current configuration. For information on configuring the master agent, see `snmpd`(8) and `snmpd.conf`(4).

There is no response returned from the master agent for a trap.

The syntax for the `esnmp_trap` routine is as follows:

```
int esnmp_trap(
          int generic_trap,
          int specific_trap,
          char *enterprise,
          VARBIND *vb);
```

The arguments are as follows:

generic_trap

A generic trap code. Set to 0 (zero) for SNMPv2 traps.

specific_trap

A specific trap code. Set to 0 (zero) for SNMPv2 traps.

enterprise

An enterprise OID string in dot notation. Set to the object identifier defined by the NOTIFICATION-TYPE macro in the defining MIB specification. This value is passed as the value of `SnmpTrapOID.0` in the SNMPv2–Trap-PDU.

vb

A `VARBIND` list of data (a NULL pointer indicates no data)

The return values are as follows:

ESNMP_LIB_OK

The routine completed successfully.

ESNMP_LIB_LOST_CONNECTION

The routine could not send the trap message to the master agent.

ESNMP_LIB_NOTOK

Something failed and a message could not be generated.

6.3.1.13 The esnmp_term Routine

The esnmp_term routine sends a close message to the master agent and shuts down the eSNMP protocol. All subagents must call this routine when terminating, so that the master agent can update its MIB registry more quickly and that system resources used by eSNMP on the behalf of the subagents can be released.

The syntax for the esnmp_term routine is as follows:

```
void esnmp_term( void );
```

The return value is:

ESNMP_LIB_OK

The esnmp_term routine always returns ESNMP_LIB_OK, even if the packet could not be sent.

6.3.1.14 The esnmp_sysuptime Routine

The esnmp_sysuptime routine converts UNIX system time obtained from gettimeofday into a value with the same timebase as sysUpTime. This can be used as a TimeTicks data type (the time since the master agent started) in units of 1/100 seconds. The time base is obtained from the master agent in response to the esnmp_init routine, so calls to this function before that time will not be accurate.

This provides a general purpose mechanism to convert UNIX timestamps into SNMP TimeTicks. The function returns the appropriate value of sysUpTime for a given UNIX. Passing a null timestamp returns the current value of sysUpTime.

The syntax is as follows:

```
unsigned int esnmp_sysuptime(
         struct timeval *timestamp);
```

The arguments are as follows:

timestamp

Is a pointer to a struct timeval containing a value obtained from the gettimeofday system call. The structure is defined in include/sys/time.h.

A NULL pointer means return the current sysUpTime.

The following is an example of the esnmp_sysuptime routine:

```
#include <sys/time.h>
#include <esnmp.h>
struct timeval timestamp;
```

```
gettimeofday(&timestamp, NULL);
o_integer(vb, object, esnmp_sysuptime(&timestamp));
```

The return is as follows:

`0` Indicates an error (`gettimeofday` failed); otherwise, *`timestamp`* contains the time in 1/100ths seconds since the master agent protocol started.

6.3.2 Method Routine Calling Interface

SNMP requests may contain many `VariableBindings` (encoded MIB variables). The `libsnmp` code executing in a subagent matches each `VariableBinding` with an object table entry. The object table's method routine is then called. Therefore, a method routine is called to service a single MIB variable. Since a single method method routine can handle a number of MIB variables, the same method routine may be called several times during a single SNMP request.

The method routine calling interface contains the following functions:

- `*_get`
- `*_set`

Section 6.3.2.3 provides additional information on method routines.

6.3.2.1 The *_get Routine

The `*_get` routine is a method routine for the specified MIB item, which is typically a MIB group (for example, `system` in MIB-2) or a table entry (for example, `ifEntry` in MIB-2). However, it is up to your discretion. See `snmpi`(8) for more information.

The `libesnmp` routines call whatever routine is specified for `Get` operations in the object table identified by the registered subtree.

This function is pointed to by some number of elements of the subagent object table. When a request arrives for an object, its method routine is called. The `*_get` method routine is called in response to a `Get` SNMP request.

The syntax for the `*_get` routine is as follows:

```
int mib-group_get(
        METHOD *method);
```

The arguments are:

`method`

A pointer to a `METHOD` structure, which contains the following fields:

action

One of ESNMP_ACT_GET, ESNMP_ACT_GETNEXT, or ESNMP_ACT_GETBULK.

serial_num

An integer number that is unique to this SNMP request. Each method routine called while servicing a single SNMP request receives the same value of *serial_num*. New SNMP requests are indicated by a new value of *serial_num*.

repeat_cnt

Used for GetBulk only. This value indicates the current iteration number of a repeating *VARBIND*. This number increments from 1 to max_repetitions, and is 0 for nonrepeating VARBIND structures.

max_repetitions

For GetBulk. The maximum number of repetitions to perform. This will be 0 for nonrepeating VARBIND structures. You can optimize subsequent processing by knowing the maximum number repeat calls will be made.

varbind

A pointer to the VARBIND structure for which we must fill in the OID and data fields. Upon entry of the method routine, the method->varbind->name field is the OID that was requested.

Upon exit of the method routine, the method->varbind field contains the requested data, and the method->varbind->name field is updated to reflect the actual instance OID for the returned VARBIND.

The libsnmp routines (o_integer, o_string, o_oid, and o_octet) are generally used to load data. The libsnmp instance2oid routine is used to update the OID in method->varbind->name field.

object

A pointer to the object table entry for the MIB variable being referenced. The method->object->object_index field is this object's unique index within the object table (useful when one method routine services many objects).

The method->object->oid field is the OID defined for this object in the MIB. The instance requested is derived by comparing this OID with the OID in the request found in the method->varbind->namefield. The oid2instance function is useful for this.

The possible return values for the *_get method routine are as follows:

ESNMP_MTHD_noError

The routine completed successfully.

ESNMP_MTHD_noSuchObject

The requested object cannot be returned or does not exist.

ESNMP_MTHD_noSuchInstance

The requested instance of an object cannot be returned or does not exist.

ESNMP_MTHD_genErr

A general processing error.

6.3.2.2 The *_set Method Routine

The *_set method routine for a specified MIB item, which is typically a MIB group (for example, system in MIB-2) or a table entry (for example, ifEntry in MIB-2). However, it is up to your discretion. See snmpi(8) for more information.

The libesnmp routines call whatever routine is specified for Set operations in the object table identified by the registered subtree.

This function is pointed to by some number of elements of the subagent object table. When a request arrives for an object, its method routine is called. The *_set method routine is called in response to a Set SNMP request.

The syntax for the *_set method routine is as follows:

```
int mib-group_set(
        METHOD *method);
```

The arguments are as follows:

method

A pointer to a METHOD structure, which contains the following fields:

action

The *action* value can be one of the following: ESNMP_ACT_SET, ESNMP_ACT_COMMIT, ESNMP_ACT_UNDO, or ESNMP_ACT_CLEANUP *serial_num*

An integer number that is unique to this SNMP request. Each method routine called while servicing a single SNMP request receives the same value of *serial_num*. New SNMP requests are indicated by a new value of *serial_num*.

varbind

A pointer to the VARBIND structure that contains the MIB variable's supplied data value and name (OID). The instance information has already been extracted from the OID and placed in method->row->*instance* field.

object

A pointer to the object table entry for the MIB variable being referenced. The method->object->*object_index* field is this object's unique index within the object table (useful when one method routine services many objects).

The method->object->*oid* field is the OID defined for this object in the MIB.

flags

A read-only integer bitmask set by libesnmp. If set, the ESNMP_FIRST_IN_ROW bit indicates that this call is the first object to be set in the row. If set, the ESNMP_LAST_IN_ROW bit indicates that this call is the last object to be set in the row. Only METHOD structures with the ESNMP_LAST_IN_ROW bit set are passed to the method routines for commit, undo, and cleanup phases.

row

A pointer to a ROW_CONTEXT structure (defined in the esnmp.h header file). All Set calls to the method routine that refer to the same group and have the same instance number will be presented with the same row structure. The method routines can accumulate information in the row structures during Set calls for use during the commit and undo phases. The accumulated data can be released by the method routines during the cleanup phase.

The ROW_CONTEXT structure contains the following fields:

instance

An address of an array containing the instance OID for this conceptual row. The libesnmp routine builds this array by subtracting the `object oid` from the requested variable `binding oid`.

`instance_len`

The size of the `method->row->`*`instance`* field.

`context`

A pointer to be used privately by the method routine to reference data needed to process this request.

`save`

A pointer to be used privately by the method routine to reference data needed to potentially undo this request.

`state`

An integer to be used privately by the method routine to hold any state information it requires.

The possible returns for the `*_set` method routine are as follows:

ESNMP_MTHD_noError

The routine completed successfully.

ESNMP_MTHD_notWritable

The requested object cannot be set or was not implemented.

ESNMP_MTHD_wrongType

The data type for the requested value is the wrong type.

ESNMP_MTHD_wrongLength

The requested value is the wrong length.

ESNMP_MTHD_wrongEncoding

The requested value is represented incorrectly.

ESNMP_MTHD_wrongValue

The requested value is out of range.

ESNMP_MTHD_noCreation

The requested instance can never be created.

ESNMP_MTHD_inconsistentName

The requested instance cannot currently be created.

ESNMP_MTHD_inconsistentValue

The requested value is not consistent.

ESNMP_MTHD_resourceUnavailable

A failure due to some resource constraint.

ESNMP_MTHD_genErr

A general processing error.

ESNMP_MTHD_commitFailed

The commit phase failed.

ESNMP_MTHD_undoFailed

The undo phase failed.

Overall Processing of the *_set Routine

Every variable binding is parsed and its object is located in the object table. A `METHOD` structure is created for each `VARBIND` structure. These `METHOD` structures point to a `ROW_CONTEXT` structure, which is useful for handling these phases. Objects in the same conceptual row all point to the same `ROW_CONTEXT` structure. This determination is made by checking the following:

- The referenced objects are in the same MIB group.
- The `VARBIND` structures have the same instance OIDs.

Each `ROW_CONTEXT` structure is loaded with the instance information for that conceptual row. The `ROW_CONTEXT` structure *context* and *save* fields are set to NULL, and the *state* field is set to ESNMP_SET_UNKNOWN structure.

The method routine for each object is called, being passed its `METHOD` structure with an action code of ESNMP_ACT_SET.

If all method routines return success, a single method routine (the last one called for the row) is called for each row, with `method->action == ESNMP_ACT_COMMIT`.

If any row reports failure, all rows that were successfully committed are told to undo the phase. This is accomplished by calling a single method routine for each row (the same one that was called for the commit phase), with a `method->action == ESNMP_ACT_UNDO`.

Finally, each row is released. The same single method routine for each row is called with a `method->action == ESNMP_ACT_CLEANUP`. This occurs for every row, regardless of the results of previous processing.

The following list describes the action codes:

ESNMP_ACT_SET

Each object's method routine is called during the `Set` phase, until all objects are processed or a method routine returns an error status value. (This is the only phase during which each object's method routine is called.) For variable bindings in the same conceptual row, `method->row` points to a common `ROW_CONTEXT`.

The `method->flags` bitmask has the ESNMP_LAST_IN_ROW bit set, if this is the last object being called for this `ROW_CONTEXT`. This enables you to do a final consistency check, because you have seen every variable binding for this conceptual row.

The method routine's job in this phase is to determine if the `SetRequest` will work, return the correct SNMP error code if not, and prepare any context data it needs to actually perform the `Set` during the commit phase.

The `method->row->context` is private to the method routine; `libesnmp` does not use it. A typical use is to store the address of an emitted *foo*`_type` structure that has been loaded with the data from the `VARBIND` for the conceptual row.

ESNMP_ACT_COMMIT

Even though several variable bindings may be in a conceptual row, only the last one in order of the `SetRequest` is processed. So, for all the method routines that point to a common row, only the last method routine is called.

This method routine must have available to it all necessary data and context to perform the operation. It must also save a snapshot of current data or whatever it needs to undo the `Set` if required. The `method->row->save` field is intended to hold a pointer to whatever data is needed to accomplish this. A typical use is to store the address of an *xxx* structure that has been loaded with the current data for the conceptual row. The *xxx* structure is one that has been automatically generated by the `snmpi` program.

The `method->row->save` field is also private to the method routine; `libesnmp` does not use it.

If the set operation succeeds, return ESNMP_MTHD_noError; otherwise, back out the commit as best you can and return a value of ESNMP_MTHD_commitFailed.

If any errors were returned during the commit phase, `libesnmp` enters the undo phase; if not, it enters the cleanup phase.

Note

> The undo phase may occur even if the `Set` operation in your subagent is successful because the `SetRequest` spanned subagents and some other subagent failed.

ESNMP_ACT_UNDO

For each conceptual row that was successfully committed, the same method routine is called with `method->action == ESNMP_ACT_UNDO`. The `ROW_CONTEXT` structures that have not yet been called for the commit phase are not called for the undo phase; they are called for cleanup phase.

The method routine should attempt to restore conditions to what they were before it executed the commit phase. (This is typically done using the data pointed to by the `method->row->save` field.)

If successful, return ESNMP_MTHD_noError; otherwise, return ESNMP_MTHD_undoFail.

ESNMP_ACT_CLEANUP

Regardless of what else has happened, at this point each `ROW_CONTEXT` participates in cleanup phase. The same method routine that was called for commit phase is called with `method->action == ESNMP_ACT_CLEANUP`.

This indicates the end of processing for the `SetRequest`. The method routine should perform whatever cleanup is required; for instance, freeing dynamic memory that might have been allocated and stored in `method->row->context` and `method->row->save` fields, and so on.

The function return status value is ignored for the cleanup phase.

6.3.2.3 Method Routine Applications Programming

You must write the code for the method routines declared in the *subtree*`_tbl.h` file. Each method routine has one argument, which is a pointer to the `METHOD` structure, as follows:

```
int mib_group_get(
         METHOD *method,

int mib_group_set(
         METHOD *method);
```

The Get method routines are used to perform Get, GetNext, and GetBulk operations.

Get method routines perform the following tasks:

1. Extract the instance portion of the requested OID. You can do this manually by comparing the *method->object->oid* field (the object's base OID) to the *method->varbind->name* field (the requested OID). You can use the oid2instance libesnmp routine to do this.
2. Determine the instance validity. The instance OID may be null or any length, depending on what was requested and how your object was selected. You may be able to reject the request immediately by checking on the instance OID.
3. Extract the data. Based on the instance OID and *method->action* field, determine what data, if any, is to be returned.
4. Load the response OID back into the method routine's VARBIND structure. Set the *method->varbind* field with the OID of the actual MIB variable instance you are returning. This is usually accomplished by loading an array of integers with the instance OID you wish to return and calling the instance2OID libesnmp routine.
5. Load the response data back into the method routine's VARBIND structure.

   Use one of the libesnmp library routine with the corresponding data type to load the *method->varbind* field with the data to return:

   – o_integer

   – o_string

   – o_octet

   – o_oid

   These routines make a copy of the data you specify. The libesnmp function manages any memory associated with copied data. The method routine must manage the original data's memory.

   The routine does any necessary conversions to the type defined in the object table for the MIB variable and copies the converted data into the *method->varbind* field.

   See the Value Representation section for information on data value representation.
6. Return the correct status value, as follows:

   – ESNMP_MTHD_noError — The routine completed successfully or no errors were found.

- ESNMP_MTHD_noSuchInstance — There is no such instance of the requested object.
- ESNMP_MTHD_noSuchObject — No such object exists.
- ESNMP_MTHD_ genErr — An error occurred and the routine did not complete successfully.

Value Representation

The values in a `VARBIND` structure for each data type are represented as follows. (Refer to the `esnmp.h` file for a definition of the `OCT` and `OID` structures.)

- ESNMP_TYPE_Integer32 (*varbind->value.sl* field)

  This is a 32-bit signed integer. Use the `o_integer` routine to insert an integer value into the `VARBIND`. Note that the prototype for the value argument is unsigned long, so you may need to cast this to a `signed int`.

- ESNMP_TYPE_DisplayString, ESNMP_TYPE_Opaque, ESNMP_TYPE_OctetString (*varbind->value.oct* field)

  This is an octet string. It is contained in the `VARBIND` structure as an `OCT` structure that contains a length and a pointer to a dynamically allocated character array.

  The `DisplayString` is different only in that the character array can be interpreted as ASCII text where the `OctetString` can be anything. If the `OctetString` contains bits or a bitstring, the `OCT` structure contains the following:

  - A length equal to the number of bytes needed to contain the value that is ((qty-bits - 1)/8 + 1)
  - A pointer to a buffer containing the bits of the `bitstring` in the form *bbbbb..bb*, where the *bb* octets represent the bit string itself, where bit zero (0) comes first and so on. Any unused bits in the last octet are set to zero (0).

  Use the `o_string` routine to insert a value into the `VARBIND` structure, which is a buffer and a length. New space will be allocated and the buffer copied into the new space.

  Use the `o_octet` routine to insert a value into the `VARBIND` structure, which is a pointer to an `OCT` structure. New space is allocated and the buffer pointed to by the `OCT` structure is copied.

- ESNMP_TYPE_ObjectId (*varbind->value.oid* and the *varbind->name* fields)

This is an object identifier. It is contained in the `VARBIND` structure as an `OID` structure that contains the number of elements and a pointer to a dynamically allocated array of unsigned integers, one for each element.

The *`varbind->name`* field is used to hold the object identifier and instance information that identifies MIB variable. Use the `OID2Instance` function to extract the instance elements from an incoming OID on a request. Use the `instance2oid` function to combine the instance elements with the MIB variable's base OID to set the `VARBIND` structure's *`name`* field when building a response.

Use the `o_oid` function to insert an object identifier into the `VARBIND` structure when the OID value to be returned as data is in the form of a pointer to an `OID` structure.

Use the `o_string` function to insert an object ID into the `VARBIND` structure when the OID value to be returned as data is in the form of a pointer to an ASCII string containing the OID in dot format; for example `1.3.6.1.2.1.3.1.1.2.0`.

- ESNMP_TYPE_NULL

  This is the NULL or empty type. This is used to indicate that there is no value. The length is 0 and the value union in the `VARBIND` structure is zero-filled.

  The incoming `VARBIND` structures on a `Get`, `GetNext`, and `GetBulk` will have this data type. A method routine should never return such a value. An incoming `Set` request never has such a value in a `VARBIND` structure.

- ESNMP_TYPE_IpAddress (*`varbind->value.oct`* field)

  This is an IP address. It is contained in the `VARBIND` structure in an `OCT` structure that has a length of 4 and a pointer to a dynamically allocated buffer containing the 4 bytes of the IP address in network byte order.

  Use the `o_integer` function to insert an IP address into the `VARBIND` structure when the value is an unsigned integer in network byte order.

  Use the `o_string` function to insert an IP address into the `VARBIND` structure when the value is a byte array (in network byte order). Use a length of 4.

- ESNMP_TYPE_UInteger32 ESNMP_TYPE_Counter32 ESNMP_TYPE_Gauge32 (*`varbind->value.ul`* field)

  The 32-bit counter and 32-bit gauge data types are stored in the `VARBIND` structure as an `unsigned int`.

  Use the `o_integer` function to insert an unsigned value into the `VARBIND` structure.

- ESNMP_TYPE_TimeTicks `(`*`varbind->value.ul`* field)

The 32-bit timeticks type values are stored in the VARBIND structure as an unsigned int.

Use the o_integer function to insert an unsigned value into the VARBIND structure.

- ESNMP_TYPE_Counter64 (*varbind->value.ul64*)

  The 64-bit counter is stored in a VARBIND structure as an unsigned long, which on an Alpha machine has a 64-bit value.

  Use the o_integer function to insert an unsigned long value (64 bits) into the VARBIND structure.

6.3.3 The libsnmp Support Routines

The following sections provide information on the libsnmp support routines, which consist of the following:

- o_integer
- o_octet
- o_oid
- o_string
- str2oid
- sprintoid
- instance2oid
- oid2instance
- inst2ip
- cmp_oid
- cmp_oid_prefix
- clone_oid
- free_oid
- clone_buf
- mem2oct
- cmp_oct
- clone_oct
- free_oct
- free_varbind_data
- set_debug_level
- is_debug_level

- ESNMP_LOG

6.3.3.1 The o_integer Routine

The o_integer routine loads an integer value into the VARBIND structure with the appropriate type.

The syntax is as follows:

```
int o_integer(
        VARBIND *vb,
        OBJECT *obj,
        unsigned long value);
```

The arguments are as follows:

vb

A pointer to the VARBIND structure that is to receive the data. This function does not allocate the VARBIND structure.

obj

A pointer to the OBJECT structure for the MIB variable associated with the OID in the VARBIND structure.

value

The value to be inserted into the VARBIND structure.

The real type as defined in the object structure must be one of the following; otherwise, an error is returned.

If the real type is IpAddress, the assumption is that the 4-byte integer is in network byte order.

ESNMP_TYPE_Integer32:

32-bit INTEGER

ESNMP_TYPE_Counter32:

32-bit Counter (unsigned)

ESNMP_TYPE_Gauge32:

32-bit Gauge (unsigned)

ESNMP_TYPE_TimeTicks:

32-bit TimeTicks (unsigned)

ESNMP_TYPE_UInteger32:

32-bit INTEGER (unsigned)

ESNMP_TYPE_Counter64:

64-bit Counter (unsigned)

ESNMP_TYPE_IpAddress:

IMPLICIT OCTET STRING (4)

The following is an example of the `o_integer` routine:

```
#include <esnmp.h>
#include "ip_tbl.h"  <-- for ipNetToMediaEntry_type definition
VARBIND     *vb       = method->varbind;
OBJECT      *object   = method->object;
ipNetToMediaEntry_type *data;
:
: assume buffer and structure member assignments occur here
:
switch(arg) {
case I_atIfIndex:
return o_integer(vb, object, data->ipNetToMediaIfIndex);
```

The following are the return values:

ESNMP_MTHD_noError

The routine completed successfully.

ESNMP_MTHD_genErr

An error has occurred.

6.3.3.2 The o_octet Routine

The `o_octet` routine loads an octet value into the `VARBIND` structure with the appropriate type.

The syntax is as follows:

```
int o_octet(
         VARBIND *vb,
         OBJECT *obj,
         OCT *oct);
```

The arguments are as follows:

vb

A pointer to the `VARBIND` structure that is to receive the data. This function does not allocate the `VARBIND` structure.

Note

If the original value in the *vb* field is not NULL, this routine attempts to free it. So if you issue the `malloc` command to allocate your own *vb* structure, be sure to fill it with zeros before using it.

obj

A pointer to the `OBJECT` structure for the MIB variable associated with the `OID` in the `VARBIND` structure.

value

The value to be inserted into the `VARBIND` structure.

The real type as defined in the object structure must be one of the following; otherwise, an error is returned:

ESNMP_TYPE_OCTET_STRING

OCTET STRING

ESNMP_TYPE_IpAddress

IMPLICIT OCTET STRING (4) — in octet form, network byte order

ESNMP_TYPE_DisplayString

DisplayString (Textual Convention)

ESNMP_TYPE_Opaque

IMPLICIT OCTET STRING

The following is an example of the `o_octet` routine:

```
#include <esnmp.h>
#include "ip_tbl.h"  <-- for ipNetToMediaEntry_type definition
VARBIND     *vb       = method->varbind;
OBJECT      *object   = method->object;
ipNetToMediaEntry_type *data;
:
: assume buffer and structure member assignments occur here
:
switch(arg) {
 case I_atPhysAddress:
 return o_octet(vb, object, &data->ipNetToMediaPhysAddress);
```

The returns are as follows:

ESNMP_MTHD_noError

The routine completed successfully.

ESNMP_MTHD_genErr

An error condition has occurred.

6.3.3.3 The o_oid Routine

The `o_oid` routine loads an OID value into the `VARBIND` structure with the appropriate type.

The syntax is as follows:

```
int o_oid(
        VARBIND *vb,
        OBJECT *obj,
        OID *oid);
```

The arguments are as follows:

vb

A pointer to the `VARBIND` structure that is to receive the data. This function does not allocate the `VARBIND` structure.

Note

If the original value in the *vb* field is not NULL, this routine attempts to free it; therefore, if you issue the `malloc` command to allocate your own *vb* structure, fill it with zeros (0s) before using it.

obj

A pointer to the `OBJECT` structure for the MIB variable associated with the OID in the `VARBIND` structure.

value

The value to be inserted into the `VARBIND` structure as data. See Section 6.2.1 for OID length and values.

The real type as defined in the object structure must be the following; otherwise, an error is returned:

ESNMP_TYPE_OBJECT_IDENTIFIER

OBJECT IDENTIFIER

The following is an example of the `o_oid` routine:

```
#include <esnmp.h>
#include "ip_tbl.h"  <-- for ipNetToMediaEntry_type definition
VARBIND     *vb       = method->varbind;
OBJECT      *object   = method->object;
ipNetToMediaEntry_type *data;
:
: assume buffer and structure member assignments occur here
:
switch(arg) {
 case I_atObjectID:
  return o_oid(vb, object, &data->ipNetToMediaObjectID);
```

The returns are as follows:

ESNMP_MTHD_noError

The routine completed successfully.

ESNMP_MTHD_genErr

An error condition has occurred.

6.3.3.4 The o_string Routine

The `o_string` routine loads a string value into the `VARBIND` structure with the appropriate type.

The syntax is as follows:

```
int o_string(
         VARBIND *vb,
         OBJECT *obj,
         unsigned char *ptr,
         int len);
```

The arguments are as follows:

vb

A pointer to the `VARBIND` structure that is to receive the data. This function does not allocate the `VARBIND` structure.

Note

If the original value in the *vb* field is not NULL, this routine attempts to free it; therefore, if you issue the `malloc` command to allocate your own *vb* structure, fill it with zeros (0s) before using it.

obj

A pointer to the `OBJECT` structure for the MIB variable associated with the `oid` in the `VARBIND` structure.

ptr

A pointer to the buffer containing data to be inserted into the `VARBIND` structure as data.

len

The length of the data in buffer to which *ptr* field points.

The real type as defined in the object structure must be one of the following; otherwise, an error is returned:

ESNMP_TYPE_OCTET_STRING

OCTET STRING

ESNMP_TYPE_IpAddress

IMPLICIT OCTET STRING (4) — in octet form, network byte order

ESNMP_TYPE_DisplayString

DisplayString (Textual Convention)

ESNMP_TYPE_Opaque

IMPLICIT OCTET STRING

ESNMP_TYPE_OBJECT_IDENTIFIER

OBJECT IDENTIFIER — in dot notation, 1.3.6.1.4.1.3.6

The following is an example of the `o_string` routine:

```
#include <esnmp.h>
#include "ip_tbl.h"  <-- for ipNetToMediaEntry_type definition
VARBIND     *vb       = method->varbind;
OBJECT      *object   = method->object;
ipNetToMediaEntry_type *data;
:
: assume buffer and structure member assignments occur here
:
switch(arg) {
 case I_atPhysAddress:
  return o_string(vb, object, data->ipNetToMediaPhysAddress.ptr,
                              data->ipNetToMediaPhysAddress.len);
```

The return values are as follows:

ESNMP_MTHD_noError

The routine completed successfully.

ESNMP_MTHD_genErr

An error condition has occurred.

6.3.3.5 The str2oid Routine

The `str2oid` routine converts a null-terminated OID string (in dot notation) to an `OID` structure.

It dynamically allocates the elements buffer and inserts its pointer into the `OID` structure passed in. It is the responsibility of the caller to free this buffer. The OID can have a maximum of 128 elements. A null string or empty string returns an OID structure that has one element of zero (0).

Note that the `str2oid` routine does not allocate an `OID` structure.

The syntax is as follows:

```
OID * str2oid(
         OID *oid,
         char *s);
```

The following is an example of the `str2oid` routine:

```
#include <esnmp.h>
OID abc;
if (str2oid(&abc, "1.2.5.4.3.6") == NULL)
    DPRINTF((WARNING,"It did not work...\n"));
```

The returns are as follows:

NULL

An error has occurred; otherwise, the pointer to the `OID` structure (its first argument) is returned.

6.3.3.6 The sprintoid Routine

The `sprintoid` routine converts an `OID` into a null-terminated string in dot notation. An `OID` structure can have up to 128 elements. A full sized `OID` structure can require a large buffer.

The syntax is as follows:

```
char * sprintoid(
         char *buffer,
         OID *oid);
```

The following is an example of the `sprintoid` routine:

```
#include <esnmp.h>
#define SOMETHING_BIG 1024
OID abc;
char buffer[SOMETHING_BIG];
:
```

```
: assume abc gets initialized with some value
:
printf("dots=%s\n", sprintoid(buffer, &abc));
```

The return value points to its first argument.

6.3.3.7 The instance2oid Routine

The `instance2oid` routine makes a copy of the object's base OID and appends a copy of the instance array to make a complete OID for a value. The `instance` is an array of integers and `len` is the number of elements. The instance array may be created by `oid2instance` or constructed from key values as a result of a `get_next` search.

The routine dynamically allocates the elements buffer and inserts its pointer into the `OID` structure passed in the call. The calling program or module is responsible for freeing this buffer.

To use this routine, point to the `OID` structure that is to receive the new `OID` values and call this routine. Any previous value in the `OID` structure is freed (it calls `free_oid` first) and the new values are dynamically allocated and inserted. Be sure the initial value of the new `OID` structure is all zeros, if you do not want it to be freed.

Note that the `instance2oid` routine does not allocate an `OID` structure, only the array containing the elements.

The syntax is as follows:

```
OID * instance2oid(
         OID *new,
         OBJECT *obj,
         unsigned int *instance,
         int len);
```

The arguments are as follows:

new

A pointer to the `OID` structure that is to receive the new OID value.

obj

A pointer to the object table entry for the MIB variable being obtained. The first part of the new OID is the OID from this MIB object table entry.

instance

A pointer to an array of *instance* values. These values are appended to the base OID obtained from the MIB object table entry to construct the new OID.

len

The number of elements in the *instance* array.

The following is an example of the instance2oid routine:

```
#include <esnmp.h>
VARBIND *vb;      <-- filled in
OBJECT  *object;  <-- filled in
unsigned int instance[6];

-- Construct the outgoing OID in a GETNEXT          --
-- Instance is N.1.A.A.A.A where A's are IP address --
instance[0] = data->ipNetToMediaIfIndex;
instance[1] = 1;
for (i = 0; i < 4; i++) {
instance[i+2]=((unsigned char *)(&data->ipNetToMediaNetAddress))[i];
}
instance2oid(&vb->name, object, instance, 6);
```

The returns are as follows:

NULL

An error has occurred; otherwise, the pointer to the OID structure (its first argument) is returned.

6.3.3.8 The oid2instance Routine

The oid2instance routine extracts the instance values from an OID structure and copies them to the specified array of integers. It then returns the number of elements in the array. The instance is the elements of an OID beyond those elements that identify the MIB variable. They are used as indexes to identify a specific instance of a MIB value.

If the OID structure contains more elements than expected (more than specified by the *max_len* parameter), the function copies the number of elements specified by *max_len* only and returns the total number of elements that would have been copied had there been space.

The syntax is as follows:

```
int oid2instance(
        OID *oid,
        OBJECT *obj,
        unsigned int *instance,
        int max_len);
```

The arguments are as follows:

oid

A pointer to an incoming OID structure containing an instance or part of an instance.

obj

A pointer to the object table entry for the MIB variable.

instance

A pointer to an array of unsigned integers where the index will be placed.

max_len

The number of elements available in the instance array.

```
#include <esnmp.h>
OID         *incoming = &method->varbind->name;
OBJECT      *object   = method->object;
int         instLength;
unsigned int instance[6];

-- in a GET operation --
-- Expected Instance is N.1.A.A.A.A where A’s are IP address --
instLength = oid2instance(incoming, object, instance, 6);
if (instLength != 6)
    return ESNMP_MTHD_noSuchInstance;
```

The `N` will be in `instance[0]` and the IP address will be in `instance[2]`, `instance[3]`, `instance[4]`, and `instance[5]`.

The returns are as follows:

- <0 — An error occurred. This is not returned if the object was obtained by looking at this `oid`.
- 0 — There are no instance elements.
- >0 — The number of elements in the index. (This could be larger than the *max_len* parameter).

6.3.3.9 The inst2ip Routine

The `inst2ip` routine returns an IP address derived from an OID instance. For evaluation of an instance for `Get` and `Set` operations use the EXACT mode. For `GetNext` and `GetBulk` operations use the NEXT mode. When using the NEXT mode, this routine’s logic assumes that the search for data will be performed using greater than or equal to matches.

The syntax is as follows:

```
int inst2ip(
         unsigned int *inst,
         int length,
         unsigned int *ipAddr,
         int exact,
        int carry);
```

The arguments are as follows:

inst

A pointer to an array of `unsigned int` containing the instance numbers returned by the `oid2instance` routine to be converted to an IP address.

Each element is in the range 0 to 255. Using the EXACT mode, the routine returns 1 if an element is out of range. Using NEXT mode, a value greater than 255 causes that element to overflow. It is set to 0 and the next most significant element is incremented, so it returns a lexically equivalent value of the next possible `ipAddress`.

length

The number of elements in the instance array. Instances beyond the fourth are ignored. If the length is less than 4, the missing values are assumed to be 0. A negative length results in an *ipAddr* value of 0. For an exact match (such as `Get`) there must be at exactly four elements.

ipAddr

A pointer to where to return the IP address value. It is in network byte order; that is, the most significant element is first.

exact

Either TRUE or FALSE.

`TRUE` means do an EXACT match. If any element is greater than 255 or if there are not exactly 4 elements, return 1. The carry argument is ignored.

`FALSE` means do a NEXT match. That is, return the lexically next IP address if the carry is set and the length is at least 4. If there are fewer than 4 elements, assume the missing values are 0. If any one element contains a value greater than 255, then zero the value and increment the next most significant element. Return 1 only in the case where there is a carry from the more significant (the first) value.

carry

Is the carry to add to the IP address on a NEXT match. If you are trying to determine the next possible IP address, pass in a 1; otherwise, pass in a 0. A length of less than 4 cancels the carry.

The following are examples of the `inst2ip` routine.

The following example converts an instance to an IP address for a `Get` operation, which is an `EXACT` match.

```
#include <esnmp.h>
OID         *incoming = &method->varbind->name;
OBJECT      *object   = method->object;
int instLength;
unsigned int instance[6];
unsigned int ip_addr;
int          iface;

-- The instance is N.1.A.A.A.A where A.A.A.A is the IP address--
instLength = oid2instance(incoming, object, instance, 6);
if (instLength == 6 && !inst2ip(&instance[2], 4, &ip_addr, TRUE,0)) {
    iface = (int) instance[0];
}
else
   return ESNMP_MTHD_noSuchInstance;
```

The following example shows a `GetNext` operation where there is only one key or that the *`ipAddr`* value is the least significant part of the key. This is a NEXT match; therefore, a 1 is passed in for *`carry`* value.

```
#include <esnmp.h>
OID         *incoming = &method->varbind->name;
OBJECT      *object   = method->object;
int instLength;
unsigned int instance[6];
unsigned int ip_addr;
int          iface;

-- The instance is N.1.A.A.A.A where A.A.A.A is the IP address--
instLength = oid2instance(incoming, object, instance, 6);
iface = (instLength < 1) ? 0 :(int) instance[0];

iface += inst2ip(&instance[2], instLength - 2, &ip_addr, FALSE, 1);
```

In the following example, if there is more than one part to a search key and you are doing a `GetNext` operation, you want to find the next possible value for the search key so you can do a cascaded greater-than or equal-to search.

If you have a search key of a number and two *`ipAddr`* values that are represented in the instance part of the OID as *`N.A.A.A.A.B.B.B.B`* with *`N`* as single valued integer and the *`A.A.A.A`* portion making up one IP address and the *`B.B.B.B`* portion making up a second IP address and a total length of 9 if all elements are given, you start by converting the least significant part of the key, (that would be the *`B.B.B.B`* portion). You do that by calling the `inst2ip` routine passing in a 1 for the carry and (length–5) for the length.

If the conversion of the *`B.B.B.B`* portion generated a carry (returned 1), you will pass it on to the next most significant part of the key.

Therefore, convert the *`A.A.A.A`* portion by calling the `inst2ip` routine, passing in (length–1) for the length and the carry returned from the conversion of the *`B.B.B.B`* portion. The most significant element *`N`* is a number; therefore, add the carry from the *`A.A.A.A`* conversion to the number. If that also overflows, then this is not a valid search key.

```
#include <esnmp.h>
OID         *incoming = &method->varbind->name;
OBJECT      *object   = method->object;
int instLength;
unsigned int instance[9];
unsigned int ip_addrA;
unsigned int ip_addrB;
int          iface;
int       carry;

-- The instance is N.A.A.A.A.B.B.B.B --
instLength = oid2instance(incoming, object, instance, 9);
iface = (instLength < 1) ? 0 :(int) instance[0];
carry = inst2ip(&instance[5],instLength - 5,&ip_addrB,FALSE,1);
carry = inst2ip(&instance[1],instLength - 1,&ip_addrA,FALSE,carry);
iface += carry;
if (iface > carry) {
-- a carry caused an overflow in the most significant element
  return ESNMP_MTHD_noSuchInstance;
}
```

The returns are as follows:

- If the *carry* is 0, the routine ended successfully.
- If the *carry* equals 1, it indicates an error if EXACT match or there was a carry for a NEXT match. If there was a carry, the returned *ipAddr* is 0.

6.3.3.10 The cmp_oid Routine

The `cmp_oid` routine compares two `OID` structures. This routine does an element-by-element comparison starting with the most significant element (element 0) and working toward the least significant element. If all other elements are equal, the `OID` structure with the fewest elements is considered less.

The syntax is as follows:

```
int cmp_oid(
            OID *q,
            OID *p);
```

The returns are as follows:

- +1 — The `oid` *q* is greater than `oid` *p*.
- 0 — The `oid` *q* is in `oid` *p*.
- –1 — The `oid` *q* is less than `oid` *p*.

6.3.3.11 The cmp_oid_prefix Routine

The `cmp_oid_prefix` routine compares an `OID` structure against a prefix. A prefix could be the OID on an object in the object table. The elements beyond the prefix are the instance information.

This routine does an element-by-element comparison, starting with the most significant element (element 0) and working toward the least significant element. If all elements of the prefix OID match exactly with corresponding elements of `OID` *q* structure, it is considered a match even if the `OID` *q* structure contains additional elements. The `OID` *q* structure is considered greater than the OID prefix if the first nonmatching element is larger. It is considered smaller if the first nonmatching element is less.

The syntax is as follows:

```
int cmp_oid_prefix(
         OID *q,
         OID *prefix);
```

The following is an example of the `cmp_oid_prefix` routine:

```
#include <esnmp.h>
OID *q;
OBJECT *object;
if (cmp_oid_prefix(q, &object->oid) == 0)
   printf("matches prefix\n");
```

The returns are as follows:

- −1 — The `oid` is less than the prefix.
- 0 — The `oid` is in the prefix.
- +1 — The `oid` is greater than the prefix.

6.3.3.12 The clone_oid Routine

The `clone_oid` routine makes a copy of the `OID` structure.

It dynamically allocates the elements buffer and inserts its pointer into the `OID` structure passed in.

To use this routine, pass in a pointer to the old `OID` structure to be cloned and a pointer to the new `OID` structure that is to receive the duplicated `OID` values.

The calling program or module has the responsibility to free this buffer.

Note that any previous elements buffer pointed to by the new `OID` structure will be freed and pointers to the new, dynamically allocated, buffer will be inserted. Be sure to initialize the new `OID` structure with zeroes (0), unless it contains an element buffer that can be freed.

Also note that this routine does not allocate an `OID` structure.

If the old `OID` structure is NULL or contains a NULL pointer to its elements buffer, a new `OID` of {0.0} is generated.

The syntax is as follows:

```
OID *clone_oid(
         OID *new,
         OID *oid);
```

The arguments are as follows:

new

A pointer to the `OID` structure that is to receive the copy.

old

A pointer to the `OID` structure where the data is to be obtained.

The following is an example of the `clone_oid` routine:

```
#include <esnmp.h>
OID oid1;
OID oid2;
:
: assume oid1 gets assigned a value
:
memset(&oid2, 0, sizeof(OID));
if (clone_oid(&oid2, &oid1) == NULL)
    ESNMP_LOG((WARNING, "It did not work\n"));
```

The returns are as follows:

NULL

An error occurred; otherwise, the pointer to the new OID (its first argument) is returned.

6.3.3.13 The free_oid Routine

The `free_oid` routine frees an `OID` structure's elements buffer.

It frees the buffer pointed to by *oid*`->elements` field then zeros that field and the *oid*`->nelem` field.

Note that this routine does not deallocate the `OID` structure itself, only the elements buffer attached to it.

The syntax is as follows:

```
void free_oid(
         OID *oid);
```

The following is an example of the `free_oid` routine:

```
#include <esnmp.h>
OID oid;
:
: assume oid was assigned a value (perhaps with clone_oid()
: and we are now finished with it.
```

```
:
free_oid(&oid);
```

6.3.3.14 The clone_buf Routine

The clone_buf routine duplicates a buffer in a dynamically allocated space. One extra byte is always allocated on end and filled with \0. If the *len* parameter is less than 0, the duplicate buffer length is set to 0. The routine always returns a buffer pointer, unless there is a malloc error.

The caller has the responsibility to free the allocated buffer.

The syntax is as follows:

```
char *clone_buf(
          char *str,
          int len);
```

The arguments are as follows:

str

A pointer to the buffer to be duplicated.

len

The number of bytes to copy.

The following is an example of the clone_buf routine:

```
#include <esnmp.h>
char *str = "something nice";
char *copy;
copy = clone_buf(str, strlen(str));
```

The returns are as follows:

NULL

A malloc error occurred; otherwise, the pointer to allocated buffer containing a copy of the original buffer is returned.

6.3.3.15 The mem2oct Routine

The mem2oct routine converts a string, (a buffer and length) to an OCT structure.

It dynamically allocates a new buffer, copies the indicated data into it, and updates the OCT structure with the new buffer's address and length.

The caller has the responsibility to free the allocated buffer.

Note this routine does not allocate an OCT structure and that it does not free data previously pointed to in the OCT structure before making the assignment.

The syntax is as follows:

```
OCT * mem2oct(
         OCT *new,
         char *buffer,
         int len);
```

The following is an example of the mem2oct routine:

```
#include <esnmp.h>
char buffer;
int len;
OCT abc;
:
: buffer and len are initialized to something
:
memset(&abc, 0, sizeof(OCT));
if (mem2oct(&abc, buffer, len) == NULL)
    ESNMP_LOG((WARNING,"It did not work...\n"));
```

The following are the return values:

NULL

An error occurred; otherwise, the pointer to the OCT structure (its first argument) is returned.

6.3.3.16 The cmp_oct Routine

The cmp_oct routine compares two octet strings. The two octet strings are compared byte-by-byte for the length of the shortest octet string. If all bytes are equal, the lengths are compared. An octet with a null pointer is considered the same as a zero-length octet.

The syntax is as follows:

```
int cmp_oct(
         OCT *oct1,
         OCT *oct2);
```

The following is an example of the cmp_oct routine:

```
#include <esnmp.h>
OCT abc, efg;
:
: abc and efg are initialized to something
:
if (cmp_oct(&abc, &efg) > 0)
    ESNMP_LOG((WARNING,"octet abc is larger than efg...\n"));
```

The returns are as follows:

- –1 — The string to which the first parameter points is less than the second.
- 0 — The string to which the first parameter points is equal to the second.
- +1 — The string to which the first parameter points is greater than the second.

6.3.3.17 The clone_oct Routine

The `clone_oct` routine makes a copy of an `OCT` structure.

The caller passes in a pointer to the old `OCT` structure to be cloned and a pointer to the new `OCT` structure that is to receive the duplicate `OCT` structure values.

It dynamically allocates the buffer, copies the data, and updates the new `OCT` structure with the buffer's address and length.

The caller has the responsibility to free this allocated buffer.

Note that any previous buffer to which the new `OCT` structure points is freed and pointers to the new, dynamically allocated buffer are inserted. Be sure to initialize the new `OCT` structure with zeros (0), unless it contains a buffer that can be freed.

Also note that this routine does not allocate an `OCT` structure, only the elements buffer pointed to by the `OCT` structure.

The syntax is as follows:

```
OCT *clone_oct(
         OCT *new,
         OCT *old);
```

The arguments are as follows:

new

A pointer to the `OCT` structure that is to receive the copy.

old

A pointer to the `OCT` structure where the data is to be obtained.

The following is an example of the `clone_oct` routine:

```
#include <esnmp.h>
OCT octet1;
OCT octet2;
:
: assume octet1 gets assigned a value
:
```

```
memset(&octet2, 0, sizeof(OCT));
if (clone_oct(&octet2, &octet1) == NULL)
    ESNMP_LOG((WARNING, "It did not work\n"));
```

The return values are as follows:

NULL

An error occurred; otherwise, the pointer to the OCT structure (its first argument) is returned.

6.3.3.18 The free_oct Routine

The free_oct routine frees the buffer attached to an OCT structure.

It frees a dynamically allocated buffer to which the OCT structure points, then zeros (0) the pointer and length fields in the OCT structure. If the OCT structure is already NULL, this routine does nothing. If the buffer attached to the OCT structure is already NULL, this routine sets the length field of the OCT structure to zero (0).

Note that this routine does not deallocate the OCT structure, only the buffer to which it points.

The syntax is as follows:

```
void free_oct(
         OCT *oct);
```

The following is an example of the free_oct routine:

```
#include <esnmp.h>
OCT octet;
:
: assume octet was assigned a value (perhaps with clone_oct()
: and we are now finished with it.
:
free_oct(&octet);
```

6.3.3.19 The free_varbind_data Routine

The free_varbind_data routine frees the dynamically allocated fields within the VARBIND structure.

The routine performs a free_oid(*vb*->name) operation. If the *vb*->type field indicates, it then frees the *vb*->value data using either the free_oct or the free_oid routine.

It does not deallocate the VARBIND structure itself; only the name and data buffers to which it points.

The syntax is as follows:

```
void free_varbind_data(
         VARBIND *vb);
```

The following is an example of the `free_varbind_data` routine:

```
#include <esnmp.h>
VARBIND *vb;
:
: assume oid and data are declared and
: assigned appropriate values
:
vb = (VARBIND*)malloc(sizeof(VARBIND));
clone_oid(&vb->name, oid);
clone_oct(&vb->value.oct, data);
:
: some processing that uses vb occurs here
:
free_varbind_data(vb);
free(vb);
```

6.3.3.20 The set_debug_level Routine

The `set_debug_level` routine sets the logging level, which dictates what log messages are generated. The program or module calls the routine during program initialization in response to run-time options. If not called, WARNING and ERROR messages are sent to `stdout` as the default.

The syntax is as follows:

```
void set_debug_level(
         int stat,
         LOG_CALLBACK_ROUTINE callback_routine);
```

The arguments are as follows:

stat

The log level. The following values can be set individually or in combination:

ERROR

For when a bad error occurred, requiring a restart.

WARNING

For when a packet cannot be handled; this also implies ERROR.

TRACE

For when tracing all packets; this also implies ERROR and WARNING.

DAEMON_LOG

Causes output to go to `syslog` rather than to standard output.

EXTERN_LOG

Causes the callback function to be called to output log messages. If this bit is set, you must provide the second argument, which is a pointer to a user supplied external callback function. If DAEMON_LOG and EXTERN_LOG are not specified, output goes to standard output.

`callback_routine`

A user-supplied external callback function. For example:

```
void callback_function(
        int level,
        char *message);
```

The *`level`* will be `ERROR`, `WARNING`, or `TRACE`. If the EXTERN_LOG bit is set in *`stat`*, the `callback` function will be called whenever an `ESNMP_LOG` macro is executed and the log level indicates that a log message is to be generated.

This facility allows an implementer to control where eSNMP library functions output log messages. If the `EXTERN_LOG` bit will not be set, pass in a NULL pointer for the callback function argument.

The following is an example of the `set_debug_level` routine:

```
#include <esnmp.h>
extern void log_handler(int level, char *message);

if (daemonize)
    set_debug_level(EXTERN_LOG | WARNING, log_handler);
else
    set_debug_level(TRACE, NULL);
```

6.3.3.21 The is_debug_level Routine

The `is_debug_level` routine tests the logging level to see if the specified level is set. You can test the levels as follows:

- ERROR — For when a bad error occurred, requiring restart.
- WARNING — For when a packet cannot be handled; this also implies ERROR.
- TRACE — For when tracing all packets; this also implies ERROR and WARNING.
- DAEMON_LOG — For output going to `syslog`.

- EXTERN_LOG — For the *callback* function to be called to output log messages.

The syntax is as follows:

```
int is_debug_level(
          int type);
```

The return values are as follows:

TRUE The requested level is set and the ESNMP_LOG will generate output, or output will go to the specified destination.

FALSE The logging level is not set.

The following is an example of the is_debug_level routine:

```
#include <esnmp.h>

if (is_debug_level(TRACE))
    dump_packet();
```

6.3.3.22 The ESNMP_LOG Routine

The ESNMP_LOG routine is an error declaration C macro defined in the <esnmp.h> header file. It gathers the information that it can obtain and sends it to the log. If DAEMON_LOG is set, log messages are sent to the daemon log. If EXTERN_LOG is set, log messages are sent to the *callback* function; otherwise, log messages go to standard output.

Note

The esnmp_log routine is called using the ESNMP_LOG macro, which uses the helper routine esnmp_logs to format part of the text. Do not use these functions without the ESNMP_LOG macro.

```
#define ESNMP_LOG(level, x) if (is_debug_level(level)) { \
         esnmp_log(level, esnmp_logs x, __LINE__, __FILE__);}
```

Where *x* is (text):

text - *format, arguments, ...*

For example a printf statement.

level

Can be one of the following:

ERROR Declares an error condition.

WARNING	Declares a warning.
TRACE	Puts in log file if trace is active.

The syntax is as follows:

```
ESNMP_LOG(level,(format, ...))
```

The following is an example of the `ESNMP_LOG` routine:

```
#include <esnmp.h>
ESNMP_LOG( ERROR, ("Cannot open file %s\n", file));
```

7

RSVP Application Programming Interface

The Resource ReSerVation Protocol (RSVP) is an network layer protocol that enables Internet applications to request enhanced quality-of-service (QoS) for specific application data streams or flows, either simplex unicast or multicast. Applications will typically use the RSVP Application Programming Interface (RAPI) to request enhanced QoS when the default best effort delivery is not acceptable (for example, video and audio). The types of QoS applications may request are defined by Internet Integrated Services.

This chapter assumes you are familiar with the following:

- IETF Integrated Services (RFC 1633)
- RSVP protocol (RFC 2205)
- RSVP and Integrated Services (RFC 2210)
- Controlled-Load Service (RFC 2211)
- The C programming language

This chapter provides the following information:

- Overview of network quality of service
- Information about the network QoS components
- Information about the RSVP application programming interface (RAPI) routines

7.1 Network Quality of Service

As applications place ever increasing demands for bandwidth on the Internet network, increasing the network bandwidth is only a temporary solution. Newer real-time applications demand both increased bandwidth and low latency. Clearly, there must be a mechanism to manage bandwidth use in the network.

At present, an IP network with its Best Effort delivery service already performs a form of passive bandwidth management. If an outgoing queue is full, indicating high network traffic and congestion, the packets are quietly dropped. Some upper-level protocols can detect this data loss, others cannot.

Quality of service (QoS) is the phrase commonly associated with the concept of actively managing network bandwidth. In this scenario, all network elements (for example, hosts, applications, and routers) and all network protocol layers cooperate to ensure consistent traffic and service end-to-end in a network. Network bandwidth for real-time applications is reserved, while sufficient bandwidth remains for best-effort traffic.

7.2 Network Quality of Service Components

The major network quality of service components in this operating system are as follows:

- Traffic Control
- RSVP
- RAPI

These components work together to provide network QoS. The following sections describe each of these components.

7.2.1 Traffic Control

On this operating system, the Traffic Control subsystem implements the Controlled-Load service as defined in RFC 2211. This service provides an application data flow with a QoS that approximates Best Effort delivery through unloaded network interfaces. The Traffic Control subsystem consists of the following components:

- Admission control — Verifies that the local system interface has the requested bandwidth and installs flows and filters.
- Packet classifier — Matches the outbound IP header with installed filters.
- Packet rate enforcer — Ensures that each output flow stays within its agreed upon boundaries.
- `iftcntl` utility — Manually sets up flows and views existing flows and filters.

Traffic control is supported on the Ethernet and FDDI interfaces.

7.2.2 RSVP

RSVP is an network signaling protocol that is defined in RFC 2205. It provides a mechanism to reserve bandwidth on the local system and through the network. On this operating system, RSVP is implemented in the form of the `rsvpd` daemon. This daemon is responsible for communicating with RSVP-enabled applications, sending RSVP messages, and receiving and processing RSVP messages. The `rsvpd` daemon uses the Traffic Control subsystem to install and modify flows and filters for a specific network interface.

7.2.3 RAPI

RAPI enables a local application that requires enhanced QoS to communicate with the `rsvpd` daemon. Using the RAPI routines, an application can make resource (bandwidth) reservations on the local system or advertise services to other nodes in the network, or both. This operating system implements RAPI as defined by the Resource ReSerVation Protocol API (RAPI) Technical Standard published by The Open Group. This technical standard is a companion document to this section. See Section 7.5 for a list of supported routines and their descriptions.

7.2.4 Interoperation of Components

In RSVP, there are following roles:

- Senders (or data source) — Defined by an IP source address and a source port
- Receivers — Defined by transport protocol, IP destination address, and destination port

An application can be both a sender and a receiver.

In a typical scenario, a remote host communicates with the local application and asks to receive data. When a local application wants to declare itself a data sender, the following events might occur:

1. The application initiates communication with the `rsvpd` daemon through the RAPI interface.
2. The application passes outgoing traffic or flow characteristics, such as upper and lower bandwidth limits, delay, and jitter, to `rsvpd`. These characteristics are known as a **Tspec**. An Adspec is also passed to the daemon.
3. The `rsvpd` daemon creates a PATH message that contains this information and passes it down to Traffic Control. The interaction between the `rsvpd` daemon and Traffic Control subsystem is invisible to the requesting application.
4. Traffic Control determines if there is sufficient bandwidth on the local adapter for the reservation. If there is, Traffic Control transmits the message. The transmission can be to a unicast address or multicast address.
5. At each RSVP-enabled router, Traffic Control updates the PATH message with the previous source address (next hop upstream to the sender), modifies the Adspec depending on the QoS control services

present on the node, and transmits the message downstream to its destination.

6. At the receiver, `rsvpd` passes the Tspec and Adspec to the receiver application through RAPI.

The preceding list is provided to show a summary of the steps that can occur. Additional steps might be required.

When an local application wants to make a reservation request (declare itself as a receiver), the following events might occur:

1. The application initiates communication with the `rsvpd` daemon through the RAPI interface.
2. The application passes the traffic expected by the receiver (**Tspec**), the QoS level required (**Rspec**), and transport protocol and port number for the packets (**filter spec**) to `rsvpd`. The Rspec and Tspec are considered the flow descriptor or **flowspec**.

   The Rspec is also used to control the packet scheduling mechanism in the router or host. The filter spec also controls packet classification to determine which senders' data packets receive the corresponding QoS.
3. The `rsvpd` daemon creates a RESV message that contains this information and passes it down to Traffic Control. The interaction between the `rsvpd` daemon and Traffic Control subsystem is invisible to the requesting application.
4. Traffic Control determines if there is sufficient bandwidth on the local adapter for the reservation. If there is, Traffic Control transmits the message upstream using the source address in the PATH message.
5. At each RSVP-enabled router, Traffic Control authenticates the reservation request and allocates the necessary resources as requested in the Tspecs and the Flowspec. If the authorization fails or if there are insufficient resources available, the router sends an RSVP error back to the receiver.
6. At the last RSVP-enabled router, if the request is accepted, the router sends an RSVP confirmation message back to the receiver.
7. At each RSVP session data sender, RSVP passes the merged Flowspec object to the application. This informs the sender of the request and describes the properties of the data path.
8. If the sender agrees with the RSVP reservation parameters, it acknowledges the reservation via RSVP and proceeds to send data to receiver over a separate data connection.

The preceding list is provided to show a summary of the steps that can occur. Additional steps might be required.

The detailed manner in which reservations from different receivers are shared in the Internet is controlled by a reservation parameter known as the reservation style. See RFC 2205 for information on the different reservation styles.

7.3 Traffic Control

The Traffic Control subsystem performs the following tasks:

- Implements an admission control mechanism that maintains interface parameters, such as the device's peak output rate, the percentage of bandwidth that can be reserved for the Controlled-Load service, and the maximum number of concurrent flows.
- Ensures that applications do not pace data at a rate faster than the flowspec.
- Interfaces with the `rsvpd` daemon and the `iftcntl` command to install and remove flows and filters
- Matches all outgoing packet headers with any existing filter specs to determine on which output queue to place the packets

See `iftcntl`(8) for more information. See the *Network Administration: Connections* manual for information on enabling traffic control on the system.

7.4 RSVP

RSVP assigns QoS to specific IP data flows or sessions, which can be either multipoint-to-multipoint or point-to-point. An **RSVP session** is defined by a particular transport protocol, IP destination address, and destination port. In order to receive data packets for a particular multicast session, a host must have joined the corresponding IP multicast group. A given session may have multiple senders and if the destination is a multicast address, multiple receivers.

The following sections describe the operating system's RSVP components and the operation of the `rsvpd` daemon.

7.4.1 Components of RSVP

The operating system's RSVP components are as follows:

- `/usr/sbin/rsvpd` — RSVP daemon
- `/usr/sbin/rsvpstat` — A program that displays resource reservation status
- `/usr/shlib/librsvp.so` — RSVP library
- `/usr/include/rapi_lib.h` — RAPI definitions
- `/usr/include/rapi_err.h` — RSVP and RAPI error definitions

7.4.2 rsvpd Daemon

The `rsvpd` daemon performs the following functions:

- Listens on a raw IP socket for an incoming RSVP messages
- Communicates with applications on the local host through the RSVP Application Programming Interface (RAPI)
- Interfaces with the operating system's Traffic Control subsystem

When `rsvpd` receives an RSVP message from the network, it validates it against the local interface database and handles any authentication or protocol errors. If the request is valid, the `rsvpd` daemon then does the following:

- Keeps track of RSVP sessions and flows
- Handles resource reservations against flows
- Passes RSVP messages to the local application, if registered with `rsvpd`
- Routes RSVP packets with a destination address that does not match any interface on the local host to the next hop

See `rsvpd`(8) for information on daemon options. See the *Network Administration: Connections* manual for information on starting and stopping the `rsvpd` daemon.

7.5 RSVP Application Programming Interface

The RSVP Application Programming Interface (RAPI) enables an application to establish communications with the `rsvpd` daemon and to declare itself a data receiver or data sender, or both.

The application consists of the following:

- A main function written by the developer
- The RSVP Library routines that perform the RSVP protocol work

- A function written by the developer to process RSVP callbacks

The following sections describe the supported routines and the steps a developer needs to perform in order to write an RSVP-capable application.

7.5.1 Supported Routines

RAPI consists of the following:

- Client library services — Routines that implement communication between the application program and the `rsvpd` daemon
- RAPI formatting routines — Optional routines for convenience in formatting RAPI objects.

Table 7–1 lists the supported client library services routines and their descriptions. See each routine's reference page for complete description of the routine, its syntax, and its parameters.

Table 7–1: Client Library Services Routines

Routine	Description
`rapi_session`(3)	Creates a RAPI session locally.
`rapi_reserve`(3)	Makes, modifies, or deletes a resource reservation for a session.
`rapi_sender`(3)	Defines, redefines, or deletes a sender's data flow parameters.
`rapi_strerror`(3)	Maps a RAPI error code to an error message string.
`rapi_version`(3)	Returns the RAPI version number being used on the operating system.
`rapi_release`(3)	Closes a RAPI session and deletes all resource reservations.
`rapi_dispatch`(3)	Dispatches a RAPI event.
`rapi_getfd`(3)	Obtains the file descriptor associated with a RAPI session.
`rapi_event_rtn_t`(3)	A user-written routine that receives incoming, asynchronous RSVP events.

RSVP requests contain objects that describe the type of QoS. The RAPI formatting routines enable applications to display the contents of RAPI objects. Table 7–2 lists the supported RAPI formatting routines and their descriptions. See each routine's reference page for complete description of the routine, its syntax, and its parameters.

Table 7–2: RAPI Formatting Routines

Routine	Description
rapi_fmt_adspec(3)	Formats a given RAPI Adspec into a buffer of given address and length.
rapi_fmt_filtspec(3)	Formats a given RAPI filter spec into a buffer of given address and length.
rapi_fmt_flowspec(3)	Formats a given RAPI flowspec into a buffer of given address and length.
rapi_fmt_tspec(3)	Formats a given RAPI Tspec into a buffer of given address and length.

See the Resource ReSerVation Protocol API (RAPI) Technical Standard published by The Open Group for information on RAPI objects, protocol operations, and RAPI return values.

7.5.2 Writing a RAPI-Enabled Application

As an application developer, you are usually presented with a UNIX application and have to implement an RSVP interface.

The following steps explain this process:

1. Include the `rapi.h` header file. This file defines all data structures, constants, and function prototyes required for applications to use RAPI.
2. Code the RAPI calls.

   Write the code that calls RAPI to initialize communications with the `rsvpd` daemon.

   Write a callback routine with which to receive RSVP events.
3. Execute and test your application.

7.5.2.1 Linking Your Application

After you compile your application, you link it with either the `librsvp.so` shareable library or the `librsvp.a` static library. This library contains the protocol implementation (RAPI) that enables communication between the application and the `rsvpd` daemon.

7.5.3 Debugging RAPI Applications

To assist in testing and debugging your applications, you might want to run `rsvpd` with the `-d` flag. This forces the daemon to write error messages and debugging output to the `/var/rsvp/rsvpd_dbg.log` file.

You can also run rsvpstat to verify if connections are getting through or whether reservations are being honored. To monitor active RSVP sessions on the local system, enter the following command:

```
# /usr/sbin/rsvpstat
```

By default, the rsvpstat command displays a list of all RSVP sessions, sender and receiver, active on this system. Information includes the session number, destination address, IP protocol, port number, and the number of PATH and RESV states for the session.

To display sender information, including the contents of the actual PATH message from the sender, enter the following command:

```
# /usr/sbin/rsvpstat -Sv
```

To display receiver information, including the contents of the actual RESV message from the receiver, enter the following command:

```
# /usr/sbin/rsvpstat -Rv
```

See rsvpstat(8) for more information.

8

Tru64 UNIX STREAMS/Sockets Coexistence

This chapter describes the ifnet STREAMS module and dlb STREAMS pseudodriver communication bridges. Before reading it, you should be familiar with basic STREAMS and sockets concepts and have reviewed the information in Chapter 4 and Chapter 5.

The operating system's network programming environment supports the STREAMS and sockets frameworks for network programming. However, there is no native communication path at the data link layer between the two frameworks. The term **coexistence** refers to the ability to exchange data between the sockets and STREAMS frameworks. The term **communication bridge** refers to the software (ifnet STREAMS module or the dlb STREAMS pseudodriver) that enables the two frameworks to exchange data at the data link layer.

Programs written to sockets and STREAMS must intercommunicate for the following reasons:

- A system cannot have two drivers for the same device.
- Programs may need to access STREAMS-based device drivers from BSD protocol stacks or, conversely, may need to access BSD device drivers from STREAMS-based protocol stacks.

  For example, if your system is running a STREAMS device driver and you have an application that uses the TCP/IP implemented on Tru64 UNIX, which is sockets-based, you need a path by which the data gets from the sockets-based protocols stack to the STREAMS device driver and back again. The ifnet STREAMS module allows an application using TCP/IP to exchange data with a STREAMS device driver. Section 8.1 describes the ifnet STREAMS module.

  Conversely, if you have a STREAMS protocol stack implemented on your system but want to use the BSD device driver implemented on Tru64 UNIX, you need a path by which the data gets from the STREAMS protocol stack to the BSD device driver and back again. The dlb STREAMS pseudodriver allows the STREAMS protocol stack to route its data to the BSD device driver. Section 8.2 describes the dlb STREAMS pseudodriver.

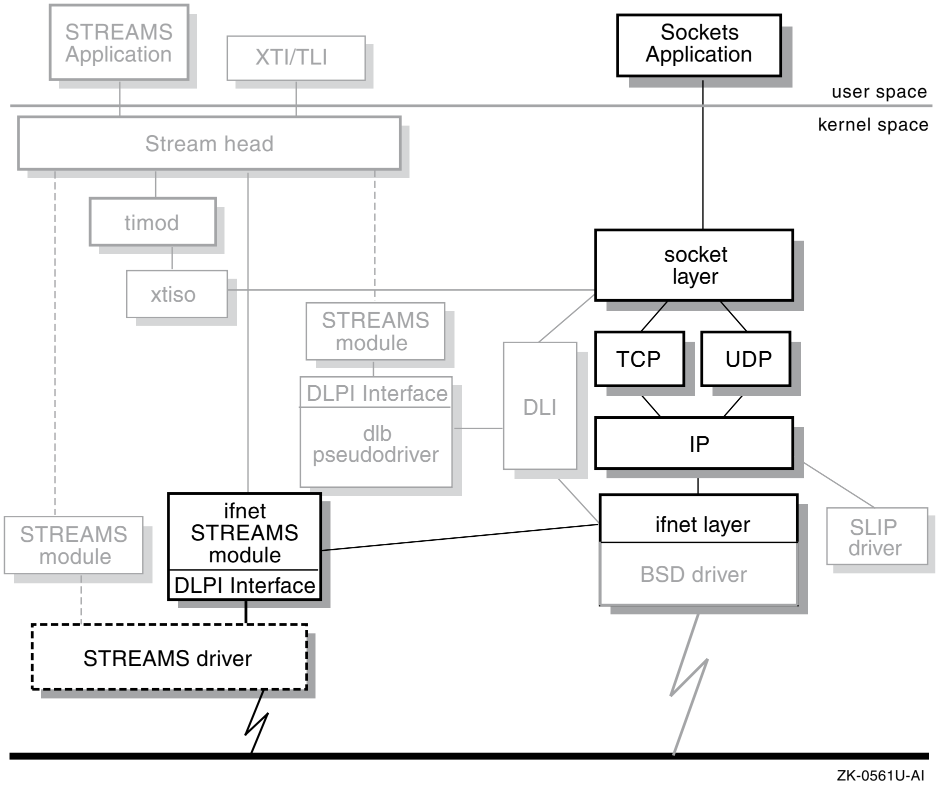
8.1 Bridging STREAMS Drivers to Sockets Protocol Stacks

The ifnet STREAMS module is a communication bridge that allows STREAMS network drivers to access sockets-based network protocols. The ifnet STREAMS module functions like any other STREAMS module, being pushed on the Stream above the STREAMS device driver. Once it is on the Stream, it handles all of the translation required between the DLPI interface of the STREAMS driver and the BSD ifnet layer. The ifnet STREAMS module exports both standard STREAMS interfaces as well as ifnet layer interfaces.

Note that STREAMS network drivers can also continue to use STREAMS-based network protocols while using the ifnet STREAMS module.

Figure 8–1 highlights the ifnet STREAMS module and shows its place in the network programming environment.

Figure 8–1: The ifnet STREAMS module



8.1.1 The STREAMS Driver

This section describes how to prepare the system running the STREAMS driver to use the `ifnet` STREAMS module.

Note

The `ifnet` STREAMS module only supports Ethernet STREAMS device drivers.

This section also lists the DLPI primitives that the STREAMS driver must support in order for the `ifnet` STREAMS module to operate successfully.

8.1.1.1 Using the ifnet STREAMS Module

If your device driver supports the primitives listed in Section 8.1.1.2, no source code changes to either the driver or STREAMS kernel code are needed for you to use the `ifnet` STREAMS module.

To use the `ifnet` STREAMS module, the STRIFNET and DLPI options must be configured in your kernel and you must set up STREAMS for the driver.

The STRIFNET and DLPI options may have been configured into your system at installation time. (For information on configuring options during installation, see the *Installation Guide*.) You can check to see if the options are configured, by issuing the following command:

```
# /usr/sbin/strsetup –c
```

If `ifnet` and `dlb` appear in the Name column, the options are configured in your kernel. If not, you must add them using the `doconfig` command.

To configure STRIFNET and DLPI into your kernel, perform the following steps:

1. Log in as superuser.
2. Enter the `/usr/sbin/doconfig` command. If you have a customized configuration file, you should use the `/usr/sbin/doconfig -c` command. For more information, see `doconfig`(8).
3. Enter a name for the kernel configuration file. It should be the name of your system in uppercase letters, and will probably be the default provided in square brackets ([]); for example:

   ```
   Enter a name for the kernel configuration file.
   [HOST1]: RETURN
   ```

4. Enter `y` when the system asks whether you want to replace the system configuration file; for example:

```
A configuration file with the name 'HOST1' already exists.
Do you want to replace it? (y/n) [n]: y

Saving /sys/conf/HOST1 as /sys/conf/HOST1.bck

*** KERNEL CONFIGURATION AND BUILD PROCEDURE ***
```

5. Select the options you want to include in your kernel.

   Note

   The STRIFNET and DLPI options are not available from this menu. To include these options, you must edit the configuration file, as shown in the following step.

6. Add DLPI and STRIFNET to the options section of the kernel configuration file.

   Enter `y` when the system asks whether you want to edit the kernel configuration file. The `doconfig` command allows you to edit the configuration file with the `ed` editor. For information about using the `ed` editor, see `ed`(1).

   The following `ed` editing session shows how to add the DLPI and STRIFNET options to the kernel configuration file for `host1`. Note that the number of the line after which you append the new lines can differ between kernel configuration files:

   ```
   Do you want to edit the configuration file? (y/n) [n]: y

   Using ed to edit the configuration file.  Press return when ready,
   or type 'quit' to skip the editing session:
   2153

   48a
   options         DLPI
   options         STRIFNET
   .
   1,$w
   2185
   q

   *** PERFORMING KERNEL BUILD ***
   ```

7. After the new kernel is built, you must move it from the directory where `doconfig` places it to the root directory (/) and reboot your system.

   When you reboot, the `strsetup –i` command runs automatically, and creates the device special files for any new STREAMS modules.

8. Run the `strsetup –c` command to verify that the device is configured properly. The following example shows the output from the `strsetup –c` command:

```
# /usr/sbin/strsetup -c
STREAMS Configuration Information...Thu Nov  9 08:38:17 1995

           Name       Type   Major  Module ID
           ----       ----   -----  ---------
          clone                 32          0
            dlb     device      52       5010
           dlpi     device      53        800
          kinfo     device      54       5020
            log     device      55         44
           nuls     device      56       5001
           echo     device      57       5000
            sad     device      58         45
           pipe     device      59       5304
       xtisoUDP     device      60       5010
       xtisoTCP     device      61       5010
      xtisoUDP+     device      62       5010
       xtisoTCP+     device      63       5010
            ptm     device      64       7609
            pts     device       6       7608
            bba     device      65      24880
            lat     device       5          5
          pppif     module               6002
       pppasync     module               6000
        pppcomp     module               6001
        bufcall     module                  0
          ifnet     module               5501
           null     module               5002
           pass     module               5003
           errm     module               5003
           ptem     module               5003
          spass     module               5007
         rspass     module               5008
        pipemod     module               5303
          timod     module               5006
         tirdwr     module                  0
          ldtty     module               7701

        Configured devices = 16, modules = 15
```

For more detailed information on reconfiguring your kernel or the `doconfig` command see the *System Administration* manual and `doconfig`(8).

To set up STREAMS for the driver you must do the following:

1. Write an application program similar to the following:

```
/*
 * Application program to set up the "pifnet" streams for IP
 * and ARP. This must be run prior to ifconfig
 */
#include <stdio.h>
```

```
#include <fcntl.h>
#include <errno.h>
#include <stropts.h>
#include <sys/ioctl.h>
#include <signal.h>
#include "dlpihdr.h"

#define IP_PROTOCOL             0x800
#define ARP_PROTOCOL            0x806
#define PIFNET_IOCTL_UNIT       1236

main(argc, argv)
        int argc;
        char *argv[];
{
        extern char *getenv();
        char *p;
        short unit = 0;
        char devName[256];

        if (argc != 3) usage();
        strcpy(devName, argv[1]);
        unit = atoi(argv[2]);

        sigignore(SIGHUP);
        setupStream(devName, unit, IP_PROTOCOL);
        setupStream(devName, unit, ARP_PROTOCOL);

        /*
         * sleep forever to keep the Streams alive.
         */
        if (fork())  /* detach */
            exit();
        pause();
}

usage()
{
       fprintf(stderr, "usage: pifnetd devname unit-number\n");
       exit(1);
}

setupStream(devName, unit, serviceClass)
  char *devName;
  short unit;
  u_long serviceClass;
{
        int fd, status;
        dl_bind_req_t bindreq;
        dl_bind_ack_t bindack;
        int flags;
        struct strioctl str;
        struct strbuf pstrbufctl, pstrbufdata, gstrbufctl, \
                                               gstrbufdata;
        char ebuf[256];

        /*
         * build the stream
         */
        fd = open(devName, O_RDWR, 0);
        if (fd < 0)
        {
                sprintf(ebuf, " open '%s' failed", devName);
                perror(ebuf);
                exit(1);
        }
        if (ioctl(fd, I_PUSH, "ifnet") < 0)
        {
                sprintf(ebuf, " ioctl I_PUSH failed");
                perror(ebuf);
                exit(1);
        }

        /*
         * tell pifnet the unit number for the device
         */
        str.ic_cmd = PIFNET_IOCTL_UNIT;
        str.ic_timout = 15;
        str.ic_len = sizeof (short);
        str.ic_dp = (char *) &unit;
        status = ioctl(fd, I_STR, &str);
        if (status < 0)
        {
                sprintf(ebuf, " %s - ioctl");
                perror(ebuf);
                exit(1);
        }

        /*
         * bind the stream to a protocol
         */
        bindreq.dl_primitive = DL_BIND_REQ;
        bindreq.dl_sap = serviceClass;
        bindreq.dl_max_conind = 0;
        bindreq.dl_service_mode = DL_CLDLS;
        bindreq.dl_conn_mgmt = 0;
        bindreq.dl_xidtest_flg = 0;
        pstrbufctl.len = sizeof(dl_bind_req_t);
        pstrbufctl.buf = (void *)&bindreq;

        pstrbufdata.buf = (char *)0;
        pstrbufdata.len = -1;
        pstrbufdata.maxlen = 0;

        status = putmsg(fd, &pstrbufctl, (struct strbuf *)0, 0);
        if (status < 0)
        {
                perror("putmsg");
                exit(1);
        }

        /*
         * Check requested binding
         */
        gstrbufctl.buf = (char *)&bindack;
        gstrbufctl.maxlen = sizeof(dl_bind_ack_t);
        gstrbufctl.len = 0;
        status = getmsg(fd, &gstrbufctl, (struct strbuf *)0, &flags);
        if (status < 0)
        {
                perror("getmsg");
                exit(1);
        }

        if (bindack.dl_primitive != DL_BIND_ACK)
        {
                errno = EPROTO;
                perror(" DL_BIND_ACK");
```

```
                exit(1);
        }
}
```

In this sample application the driver's name is /dev/streams/ln. The application creates two Streams; one for the Internet Protocol (IP) and one for the Address Resolution Protocol (ARP). After setting up the Streams, the application must keep running, using the pause command, in order to keep the Streams alive.

Note that, if the driver is a style-2 driver, you must add a DL_ATTACH_REQ primitive to the application program. For more information about the DL_ATTACH_REQ primitive or style-2 drivers, see the DLPI specification in /usr/share/doclib/dlpi/dlpi.ps.

2. Generate an executable file for the application. Compile, link, and debug the program until it runs without errors.

3. Move the executable into a directory that is convenient for you.

   The executable can be located in any directory.

4. Add a line invoking the program to the /sbin/init.d/inet file.

   Although you can manually start the program each time you reboot, it is easiest to add a line to the /sbin/init.d/inet file to run it automatically when the system reboots. Be sure to add the line before the system's ifconfig lines.

   In the following example, each time the system reboots, the /sbin/init.d/inet file runs a program called run_ifnet, which resides in the /etc directory:

```
.
.
.
#
# Enable network
#
case $1 in

        echo "Configuring network"
        /sbin/hostname $HOSTNAME
        echo "hostname: \c"
        /sbin/hostname
        if [ "$NETDEV_0" != ’’ ]; then
                echo  >/tmp/ifconfig_"$NETDEV_0".tmp
# place command invoking executable BEFORE \
       ifconfig lines
                /etc/run_ifnet
                       /sbin/ifconfig $NETDEV_0 $IFCONFIG_0 > \
                                      /tmp/ifconfig_"$NETDEV_0".tmp 2>&1
                      if [ $? != 0 ]; then
                               ERROR=‘cat /tmp/ifconfig_"$NETDEV_0".tmp‘
                               if [ "$ERROR" = "$ERRSTRING" ]; then
                                       /sbin/ifconfig $NETDEV_0 up
                               else
                                       echo "$0: $ERROR"
                               fi
                      fi
```

```
                    rm /tmp/ifconfig_"$NETDEV_0".tmp
                fi
.
.
.
```

5. Reboot the system.

   Use the `/usr/sbin/shutdown -r` command to shut down your system and have it reboot automatically; for example:

   ```
   # /usr/sbin/shutdown -r now
   ```

8.1.1.2 Data Link Provider Interface Primitives

Note that the STREAMS device driver can be a style-1 or a style-2 DLPI provider, as described in the Data Link Provider Interface specification, which is located in `/usr/share/doclib/dlpi/dlpi.ps`. Note that you must have the OSFPGMR*nnn* subset installed to access the DLPI specification on line.

The driver must support the following DLPI primitives. For detailed information about these primitives and how to use them, see the DLPI specification:

`DL_PHYS_ADDR_REQ/DL_PHYS_ADDR_ACK`

`DL_BIND_REQ/DL_BIND_ACK`

`DL_UNBIND_REQ`

`DL_UNITDATA_REQ/DL_UNITDATA_IND/DL_UDERROR_IND`

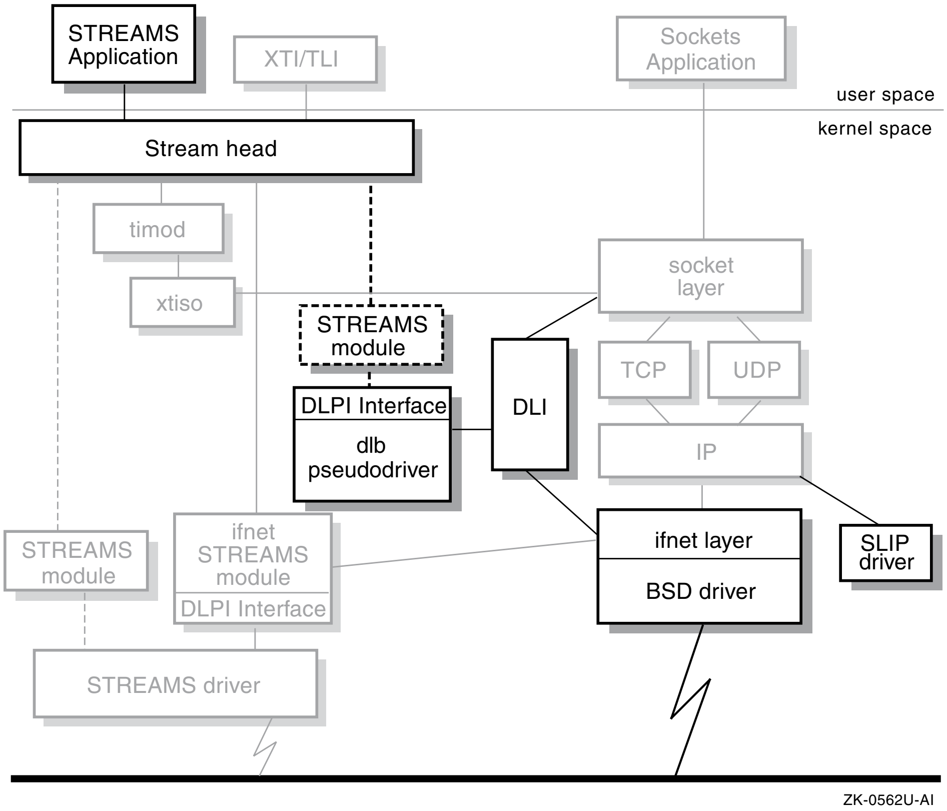
`DL_OK_ACK/DL_ERROR_ACK`

8.2 Bridging BSD Drivers to STREAMS Protocol Stacks

The `dlb` STREAMS pseudodevice driver allows you to bridge BSD-style device drivers and STREAMS protocol stacks. The STREAMS pseudodevice driver is the Stream end in a Stream wanting to communicate with BSD-based drivers. The STREAMS pseudodevice driver provided with this operating system has two interfaces, a subset of the DLPI interface that communicates with STREAMS protocol stacks and another interface that accesses the `ifnet` layer interface of the sockets framework.

Figure 8–2 highlights the `dlb` STREAMS pseudodriver and shows its place in the network programming environment.

Figure 8–2: DLPI STREAMS Pseudodriver



8.2.1 Supported DLPI Primitives and Media Types

The `dlb` STREAMS pseudodriver supports the following connectionless mode primitive and media types. For detailed information about these primitives and how to use them, see the Data Link Provider Interface specification which is in `/usr/share/doclib/dlpi/dlpi.ps`.

`DL_ATTACH_REQ/DL_DETACH_REQ/DL_OK_ACK`

`DL_BIND_REQ/DL_BIND_ACK/DL_UNBIND_REQ`

`DL_ENABMULTI_REQ/DL_DISABLMULTI_REQ`

`DL_PROMISCON_REQ/DL_PROMISCONOFF_REQ`

`DL_PHYS_ADDR_REQ/DL_PHYS_ADDR_ACK`

`DL_SET_PHYS_ADDR_REQ`

`DL_UNITDATA_REQ/DL_UNITDATA_IND`

DL_SUBS_BIND_REQ/DL_SUBS_BIND_ACK

DL_SUBS_UNBIND_REQ/DL_SUBS_UNBIND_ACK

The Ethernet bus (DL_ETHER) is the media type supported by the STREAMS pseudodriver.

8.2.2 Using the STREAMS Pseudodriver

To use the dlb STREAMS pseudodriver the DLPI option must be configured into your kernel. The DLPI option may have been configured into your system at installation time.

You can check to see if the DLPI option is configured by issuing the following command:

```
# /usr/sbin/strsetup -c
```

If dlb appears in the Name column, the option is configured in your kernel. If not, you must add it using the doconfig command.

For a description of how to reconfigure your kernel with the doconfig command, see Section 8.1.1.1.

For more information on reconfiguring your kernel or the doconfig command see the *System Administration* manual and doconfig(8). For information on configuring options during installation, see the *Installation Guide*.

A

Sample STREAMS Module

The spass module is a simple STREAMS module that passes all messages put to it to the putnext() procedure. The spass module delays the call to putnext() for the service procedure to handle. It has flow control code built in, and both the read and write sides share a service procedure.

The following is the code for the spass module:

```
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/sysconfig.h>

static  int     spass_close();
static  int     spass_open();
static  int     spass_rput();
static  int     spass_srv();
static  int     spass_wput();

static struct module_info minfo =  {
        0, "spass", 0, INFPSZ, 2048, 128
};

static struct qinit rinit = {
    spass_rput, spass_srv, spass_open, spass_close, NULL, &minfo
};

static struct qinit winit = {
    spass_wput, spass_srv, NULL, NULL, NULL, &minfo
};

struct streamtab spassinfo = { &rinit, &winit };

cfg_subsys_attr_t bufcall_attributes[]  = {
    {, 0, 0, 0, 0, 0, 0}    /* must be the last element */
};

int
spass_configure(op, indata, indata_size, outdata, outdata_size)
        cfg_op_t        op;
        caddr_t         indata;
        ulong           indata_size;
        caddr_t         outdata;
        ulong           outdata_size;
{
     struct streamadm        sa;
     dev_t                   devno = NODEV;

     if (op != CFG_OP_CONFIGURE)
             return EINVAL;

     sa.sa_version           = OSF_STREAMS_10;
     sa.sa_flags             = STR_IS_MODULE | STR_SYSV4_OPEN;
     sa.sa_ttys              = 0;
     sa.sa_sync_level        = SQLVL_QUEUE;
     sa.sa_sync_info         = 0;
     strcpy(sa.sa_name,      "spass");
     if ( (devno = strmod_add(devno, &spassinfo, &sa)) == NODEV ) {
             return ENODEV;
     }

     return 0;
}

/* Called when module is popped or the Stream is closed */
static int
spass_close (q, credp)
     queue_t * q;
     cred_t * credp;
{
     return 0;
}

/* Called when module is pushed */
static int
spass_open (q, devp, flag, sflag, credp)
     queue_t * q;
     int     * devp;
     int     flag;
     int     sflag;
     cred_t * credp;
{
     return 0;
}

/*
* Called to process a message coming upstream.  All messages
* but flow control messages are put on the read side service
* queue for later processing.
*/
static int
spass_rput (q, mp)
     queue_t * q;
     mblk_t * mp;
{
     switch (mp->b_datap->db_type) {
     case M_FLUSH:
             if (*mp->b_rptr & FLUSHR)
                     flushq(q, 0);
             putnext(q, mp);
             break;
     default:
             putq(q, mp);
             break;
     }
     return 0;
}

/*
* Shared by both read and write sides to process messages put
* on the read or write service queues.  When called from the
* write side, sends all messages on the write side queue
* downstream until flow control kicks in or all messages are
* processed.  When called from the read side sends all messages
* on its read side service queue upstreams until flow control
* kicks in or all messages are processed.
*/
static int
```

```
spass_srv (q)
     queue_t * q;
{
     mblk_t *        mp;

     while (mp = getq(q)) {
             if (!canput(q->q_next))
                  return putbq(q, mp);
             putnext(q, mp);
     }
     return 0;
}

/*
* Called to process a message coming downstream.  All messages but
* flow control messages are put on the write side service queue for
* later processing.
*/
static int
spass_wput (q, mp)
     queue_t * q;
     mblk_t *        mp;
{
     switch (mp->b_datap->db_type) {
     case M_FLUSH:
             if (*mp->b_rptr & FLUSHW)
                     flushq(q, 0);
             putnext(q, mp);
             break;
     default:
             putq(q, mp);
             break;
     }
     return 0;
}
```

B

Socket and XTI Programming Examples

This appendix contains annotated files for a sample server/client[1] credit card authorization program. Clients access a server on the merchant's behalf and request authorization from the server to put a charge on the client's credit card. The server maintains a database of authorized merchants and their passwords, as well as a database of credit card customers, their credit limit, and current balance. It either authorizes or rejects a client request based on the information in its database.

Several variations on the credit card authorization program are presented, including connection-oriented and connectionless modes. The connection-oriented and connectionless modes each contain socket and XTI code for the server and client portions of the program.

Although the program uses network programming in a real world application, it has the following limitations:

- Error handling is not robust
- Accepts only integer amounts
- Performs no child process clean up
- In the case of the connection-oriented protocol examples in Section B.1, for each request received, the server program forks a child process to handle the request. The database information is "detached" in the child process' private data area. When the child process analyzes the request and reduces the customer's credit balance appropriately, it needs to update this information in the original server's data area (and on some persistent storage as well) so that the next request for the same customer is handled correctly. To avoid unnecessary complexity, this logic is not included in the program.

The information is organized as follows:

- Connection-oriented mode programs
  - Socket
    - □ Server
    - □ Client

[1] The term `client` in this appendix refers to the program initiated by the merchant which interacts with the server program.

- – XTI
  - □ Server
  - □ Client
- Connectionless mode programs
  - – Socket
    - □ Server
    - □ Client
  - – XTI
    - □ Server
    - □ Client
- Common files

You can obtain copies of these example programs from /usr/examples/network_programming directory.

B.1 Connection-Oriented Programs

This section contains sockets and XTI variations of the same server and client programs, written for connection-oriented modes communication.

B.1.1 Socket Server Program

Example B–1 implements a server using the socket interface.

Example B–1: Connection-Oriented Socket Server Program

```
/*
 *
 * This file contains the main socket server code
 * for a connection-oriented mode of communication.
 *
 * Usage:       socketserver
 *
 */

#include "server.h"

char                    *parse(char *);
struct transaction      *verifycustomer(char *, int, char *);

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

Example B–1: Connection-Oriented Socket Server Program (cont.)

```
struct sockaddr_in    serveraddr;
struct sockaddr_in    clientaddr;
int                   clientaddrlen = sizeof(clientaddr);
struct hostent        *he;
int                   pid;


signal(SIGCHLD, SIG_IGN);

if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)    1
{
        perror("socket_create");
        exit(1);
}


bzero((char *) &serveraddr,
      sizeof(struct sockaddr_in));                     2
serveraddr.sin_family      = AF_INET;
serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);        3
serveraddr.sin_port        = htons(SERVER_PORT);       4


if ( bind(sockfd,                                      5
          (struct sockaddr *)&serveraddr,
          sizeof(struct sockaddr_in)) < 0) {
        perror("socket_bind");
        exit(2);
}

listen(sockfd, 8);                                     6

while(1) {

        if ((newsockfd =
              accept(sockfd,                           7
                     (struct sockaddr *) &clientaddr,
                     &clientaddrlen)) < 0) {
                if (errno == EINTR) {
                        printf("Bye...\n");
                        exit(0);
                } else {
                        perror("socket_accept");
                        exit(3);
                }
        }
```

Example B–1: Connection-Oriented Socket Server Program (cont.)

```
                pid = fork();

                switch(pid) {
                        case -1:        /* error */
                                perror("dosession_fork");
                                break;
                        default:
                                close(newsockfd);
                                break;
                        case 0:         /* child */

                                close(sockfd);
                                transactions(newsockfd);

                                close(newsockfd);
                                return(0);

                }
        }

}


transactions(int fd)                                           8
{
        int     bytes;
        char    *reply;
        int     dcount;
        char    datapipe[MAXBUFSIZE+1];

        /*
         * Look at the data buffer and parse commands,
         * keep track of the collected data through
         * transaction_status.
         *
         */
         while (1) {
                if ((dcount=recv(fd, datapipe, MAXBUFSIZE, 0))    9
                    < 0) {
                        perror("transactions_receive");
                        break;
                }
                if (dcount == 0) {
                        return(0);
                }

                datapipe[dcount] = '\0';
```

Example B–1: Connection-Oriented Socket Server Program (cont.)

```
                if ((reply=parse(datapipe)) != NULL) {
                        send(fd, reply, strlen(reply), 0);     10
                }
        }
}
```

1 Create a socket with the socket call.

AF_INET specifies the Internet communication domain. Alternatively, if OSI transport were supported, a corresponding constant such as AF_OSI would be required here. The socket type SOCK_STREAM is specified for TCP or connection-oriented communication. This parameter indicates that the socket is connection-oriented. Contrast the socket call with the t_open call in the XTI server example (Section B.1.3).

2 The serveraddr is of type sockaddr_in, which is dictated by the communication domain of the socket (AF_INET). The socket address for the Internet communication domain contains an Internet address and a 16-bit port number, which uniquely identifies an entity on the network. For the TCP/IP and UDP/IP this is the Internet address of the server and the port number on which it is listening.

Note that the information contained in the sockaddr_in structure is dependent on the address family, which is AF_INET in this example. If AF_OSI were used instead of AF_INET, then sockaddr_osi would be required for the bind call instead of sockaddr_in.

3 INADDRANY signifies any attached interface adapter on the system. All numbers must be converted to the network format using appropriate macros. See the following reference pages for more information: htonl(3), htons(3), ntohl(3), and ntohs(3).

4 SERVER_PORT is defined in the common.h header file. It is a short integer, which helps identify the server process from other application processes. Numbers from 0 to 1024 are reserved.

5 Bind the server's address to this socket with the bind call. The combination of the address and port number identify it uniquely on the network.

6 Specify the number of pending connections the server can queue while it finishes processing the previous accept call. This value governs the success rate of connections while the server processes accept calls. Use a larger number to obtain a better success rate if multiple clients are sending the server connect requests. The operating system imposes a ceiling on this value.

7 Accept connections on this socket. For each connection, the server forks a child process to handle the session to completion. The server then resumes listening for new connection requests. This is an example of a concurrent server. You can also have an iterative server, meaning that the server handles the data itself. See Section B.2 for an example of iterative servers.

8 Each incoming message packet is accepted and passed to the `parse` function, which tracks the information provided, such as the merchant's login ID, password, and customer's credit card number. This process is repeated until the `parse` function identifies a complete transaction and returns a response packet, to be sent to the client program.

The client program can send information packets in any order (and in one or more packets), so the `parse` function is designed to remember state information sufficient to deal with this unstructured message stream.

Since the program uses a connection-oriented protocol for data transfer, this function uses `send` and `recv` to send and receive messages, respectively.

9 Receive data with the `recv` call.

10 Send data with the `send` call.

B.1.2 Socket Client Program

Example B–2 implements a client program that can communicate with the `socketserver` interface shown in Example B–1.

Example B–2: Connection-Oriented Socket Client Program

```
/*
 *
 * This generates the client program.
 *
 * usage: client [serverhostname]
 *
 * If a host name is not specified, the local
 * host is assumed.
 *
 */


#include "client.h"

main(int argc, char *argv[])
{
        int                     sockfd;
        struct sockaddr_in      serveraddr;
```

Example B–2: Connection-Oriented Socket Client Program (cont.)

```
struct hostent          *he;
int                     n;
char                    *serverhost = "localhost";
struct hostent          *serverhostp;
char                    buffer[1024];
char                    inbuf[1024];


if (argc>1) {
        serverhost = argv[1];
}

init();

if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)   1
{
        perror("socket_create");
        exit(1);
}


bzero((char *) &serveraddr,
      sizeof(struct sockaddr_in));                    2
serveraddr.sin_family      = AF_INET;

if ((serverhostp = gethostbyname(serverhost)) ==      3
    (struct hostent *)NULL) {
        fprintf(stderr,"gethostbyname on %s failed\n",
                serverhost);
        exit(1);
}
bcopy(serverhostp->h_addr,
      (char *)&(serveraddr.sin_addr.s_addr),
                  serverhostp->h_length);

serveraddr.sin_port        = htons(SERVER_PORT);      4

/* Now connect to the server */
if (connect(sockfd, (struct sockaddr *)&serveraddr,   5
    sizeof(serveraddr)) < 0) {
        perror ("connect");
        exit(2);
}


while(1) {
        /* Merchant record */
```

Example B–2: Connection-Oriented Socket Client Program (cont.)

```
                sprintf(buffer, "%%%%m%s##%%%%p%s##",
                        merchantname, password);

                printf("\n\nSwipe card, enter amount: ");
                fflush(stdout);
                if (scanf("%s", inbuf) == EOF) {
                        printf("bye...\n");
                        exit(0);
                }
                soundbytes();

                sprintf(buffer, "%s%%%%a%s##%%%%n%s##",
                        buffer, inbuf, swipecard());

                if (send(sockfd, buffer, strlen(buffer), 0)    6
                    < 0) {
                        perror("send");
                        exit(1);
                }

                /* receive info */
                if ((n = recv(sockfd, buffer, 1024, 0)) < 0) {    7
                        perror("recv");
                        exit(1);
                }
                buffer[n] = '\0';

                if ((n=analyze(buffer))== 0) {
                        printf("transaction failure,"
                               " try again\n");
                } else if (n<0) {
                        printf("login failed, try again\n");
                        init();
                }
        }

}
```

1 Create a socket with the `socket` call.

AF_INET is the socket type for the Internet communication domain. Note that this parameter must match the protocol and type selected in the corresponding server program.

Contrast the `socket` call with the `t_open` call in the XTI client example (Section B.1.4).

2 The serveraddr is of type sockaddr_in, which is dictated by the communication domain of the socket (AF_INET). The socket address for the Internet communication domain contains an Internet address and a 16-bit port number, which uniquely identifies an entity on the network. For the TCP/IP protocol suite, this is the Internet address of the server and the port number on which it is listening.

Note that the information contained in the sockaddr_in structure is dependent on the address family (or the protocol).

3 Getting information about the server depends on the protocol or the address family. To get the IP address of the server, you can use the gethostbyname routine.

4 SERVER_PORT is defined in the <common.h> header file. It is imperative that the same port number be used to connect to the socket server program. The server and client select the port number, which functions as a well known address for communication.

5 Client issues a connect call to connect to the server. When the connect call is used with a connection-oriented protocol, it allows the client to build a connection with the server before sending data. This is analogous to dialing a phone number.

6 Send data with the send call.

7 Receive data with the recv call.

B.1.3 XTI Server Program

Example B–3 implements a server using the XTI library for network communication. It is an alternative design for a communication program that makes it transport independent. Compare this program with the socket server program in Section B.1.1. This program has the same limitations described at the beginning of the appendix.

Example B–3: Connection-Oriented XTI Server Program

```
/*
 *
 *
 * This file contains the main XTI server code
 * for a connection-oriented mode of communication.
 *
 * Usage:       xtiserver
 *
 */
#include "server.h"

char                    *parse(char *);
struct transaction      *verifycustomer(char *, int, char *);
```

Example B–3: Connection-Oriented XTI Server Program (cont.)

```
main(int argc, char *argv[])
{
        int                     xtifd;
        int                     newxtifd;
        struct sockaddr_in      serveraddr;
        struct hostent          *he;
        int                     pid;
        struct t_bind           *bindreqp;
        struct t_call           *call;


        signal(SIGCHLD, SIG_IGN);


        if ((xtifd = t_open("/dev/streams/xtiso/tcp+", O_RDWR,  1
                            NULL)) < 0) {
                xerror("xti_open", xtifd);
                exit(1);
        }


        bzero((char *) &serveraddr, sizeof(struct sockaddr_in));
        serveraddr.sin_family      = AF_INET;                  2
        serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);        3
        serveraddr.sin_port        = htons(SERVER_PORT);       4

        /* allocate structures for the t_bind and t_listen call */
        if (((bindreqp=(struct t_bind *)
                       t_alloc(xtifd, T_BIND, T_ALL))
                == NULL) ||
            ((call=(struct t_call *)
                       t_alloc(xtifd, T_CALL, T_ALL))
                == NULL)) {
                xerror("xti_alloc", xtifd);
                exit(3);
        }


        bindreqp->addr.buf      = (char *)&serveraddr;
        bindreqp->addr.len      = sizeof(serveraddr);

        /*
         * Specify how many pending connections can be
         * maintained, until finish accept processing
         *
         */
```

Example B–3: Connection-Oriented XTI Server Program (cont.)

```
bindreqp->qlen          = 8;                          5

if (t_bind(xtifd, bindreqp, (struct t_bind *)NULL)    6
    < 0) {
        xerror("xti_bind", xtifd);
        exit(4);
}

/*
 * Now the socket is ready to accept connections.
 * For each connection, fork a child process in charge
 * of the session, and then resume accepting connections.
 *
 */

while(1) {

        if (t_listen(xtifd, &call) < 0) {             7
                if (errno == EINTR) {
                        printf("Bye...\n");
                        exit(0);
                } else {
                        xerror("t_listen", xtifd);
                        exit(4);
                }
        }

        /*
         * Create a new transport endpoint on which
         * to accept a connection
         *
         */
        if ((newxtifd=t_open("/dev/streams/xtiso/tcp+",8
                             O_RDWR, NULL)) < 0) {
                xerror("xti_newopen", xtifd);
                exit(5);
        }

        /* accept connection */
        if (t_accept(xtifd, newxtifd, call) < 0) {   9
                xerror("xti_accept", xtifd);
                exit(7);
        }
        pid = fork();

        switch(pid) {
                case -1:        /* error */
```

Example B–3: Connection-Oriented XTI Server Program (cont.)

```
                                xerror("dosession_fork", xtifd);
                                break;
                        default:
                                t_close(newxtifd);
                                break;
                        case 0:         /* child */

                                t_close(xtifd);
                                transactions(newxtifd);
                                if ((t_free((char *)bindreqp,
                                     T_BIND) < 0) ||
                                     (t_free((char *)call,
                                     T_CALL) < 0)) {
                                          xerror("xti_free", xtifd);
                                          exit(3);
                                }

                                t_close(newxtifd);
                                return(0);

                }
        }

}


transactions(int fd)                                          10
{
        int     bytes;
        char    *reply;
        int     dcount;
        int     flags;
        char    datapipe[MAXBUFSIZE+1];

        /*
         * Look at the data buffer and parse commands, if more data
         * required go get it
         * Since the protocol is SOCK_STREAM oriented, no data
         * boundaries will be preserved.
         *
         */
         while (1) {
                if ((dcount=t_rcv(fd, datapipe, MAXBUFSIZE,   11
                                  &flags)) < 0){
                        /* if disconnected bid a goodbye */
                        if (t_errno == TLOOK) {
                                int tmp = t_look(fd);
```

Example B–3: Connection-Oriented XTI Server Program (cont.)

```
                                if (tmp != T_DISCONNECT) {
                                        t_scope(tmp);
                                } else {
                                        exit(0);
                                }
                        }
                        xerror("transactions_receive", fd);
                        break;
                }
                if (dcount == 0) {
                        /* consolidate all transactions */
                        return(0);
                }

                datapipe[dcount] = ’ ’;

                if ((reply=parse(datapipe)) != NULL) {
                        if (t_snd(fd, reply, strlen(reply), 0) 12
                            < 0) {
                                xerror("xti_send", fd);
                                break;
                        }
                }
        }
}
```

[1] The `t_open` call specifies a device special file name; for example `/dev/streams/xtiso/tcp+`. This file name provides the necessary abstraction for the TCP transport protocol over IP. Unlike the socket interface, where you specify the address family (for example, AF_INET), this information is already represented in the choice of the device special file. The `/dev/streams/xtiso/tcp+` file implies both TCP transport and IP. See the Chapter 5 for information about STREAMS devices.

As mentioned in Section B.1.1, if the OSI transport were available you would use a device such as `/dev/streams/xtiso/cots`.

Contrast the `t_open` call with the `socket` call in Section B.1.1.

[2] Selection of the address depends on the choice of the transport protocol. Note that in the socket example the address family was the same as used in the `socket` system call. With XTI, the choice is not obvious and you must know the appropriate mapping from the transport protocol to `sockaddr`. See Chapter 3 for more information.

3 INADDRANY signifies any attached interface adapter on the system. All numbers must be converted to the network format using appropriate macros. See the following reference pages for more information: `htonl`(3), `htons`(3), `ntohl`(3), `ntohs`(3).

4 SERVER_PORT is defined in the `<common.h>` header file. It has a data type of `short integer` which helps identify the server process from other application processes. Numbers from 0 to 1024 are reserved.

5 Specify the number of pending connections the server can queue while it processes the last request.

6 Bind the server's address with the `t_bind` call. The combination of the address and port number uniquely identify it on the network. After the server process' address is bound, the server process is registered on the system and can be identified by the lower level kernel functions to which to direct any requests.

7 Listen for connection requests with the `t_listen` function.

8 Create a new transport endpoint with another call to the `t_open` function.

9 Accept the connection request with the `t_accept` function.

10 Each incoming message packet is accepted and passed to the `parse` function, which tracks the information provided (such as the merchant's login ID, password, and customer's credit card number). This process is repeated until the `parse` function identifies a complete transaction and returns a response packet to be sent to the client program.

The client program can send information packets in any order (and in one or more packets), so the `parse` function is designed to remember state information sufficient to deal with this unstructured message stream.

Since the program uses a connection-oriented protocol for data transfer, this function uses `t_snd` and `t_rcv` to send and receive data, respectively.

11 Receive data with the `t_rcv` function.

12 Send data with the `t_snd` function.

B.1.4 XTI Client Program

Example B–4 implements a client program that can communicate with the `xtiserver` interface shown in Section B.1.3. Compare this program with the socket client program in Example B–3.

Example B–4: Connection-Oriented XTI Client Program

```
/*
 *
 * This file contains the main XTI client code
 * for a connection-oriented mode of communication.
 *
 * Usage: xticlient [serverhostname]
 *
 * If a host name is not specified, the local
 * host is assumed.
 *
 */

#include "client.h"

main(int argc, char *argv[])
{
        int                     xtifd;
        struct sockaddr_in      serveraddr;
        int                     n;
        char                    *serverhost = "localhost";
        struct hostent          *serverhostp;
        char                    buffer[1024];
        char                    inbuf[1024];
        struct t_call           *sndcall;
        int                     flags = 0;


        if (argc>1) {
                serverhost = argv[1];
        }

        init();

        if ((xtifd = t_open("/dev/streams/xtiso/tcp+", O_RDWR, 1
                            NULL)) < 0) {
                xerror("xti_open", xtifd);
                exit(1);
        }


        bzero((char *) &serveraddr,                          2
              sizeof(struct sockaddr_in));
        serveraddr.sin_family      = AF_INET;                3
        if ((serverhostp = gethostbyname(serverhost)) ==     4
            (struct hostent *)NULL) {
                fprintf(stderr,"gethostbyname on %s failed\n",
                        serverhost);
                exit(1);
```

Example B–4: Connection-Oriented XTI Client Program (cont.)

```
}
bcopy(serverhostp->h_addr,
      (char *)&(serveraddr.sin_addr.s_addr),
                  serverhostp->h_length);
serveraddr.sin_port        = htons(SERVER_PORT);     5

if (t_bind(xtifd, (struct t_bind *)NULL,             6
           (struct t_bind *)NULL) < 0) {
        xerror("bind", xtifd);
        exit(2);
}

/* Allocate structures for the t_bind and t_listen calls */
if ((sndcall=(struct t_call *)t_alloc(xtifd, T_CALL,
      T_ALL)) == NULL) {
      xerror("xti_alloc", xtifd);
      exit(3);
}

sndcall.opt.maxlen      = 0;
sndcall.udata.maxlen    = 0;
sndcall.addr.buf        = (char *)&serveraddr;
sndcall.addr.len        = sizeof(serveraddr);

if (t_connect(xtifd, sndcall,                        7
              (struct t_call *)NULL) < 0) {
        xerror ("t_connect", xtifd);
        exit(3);
}


while(1) {
        /* Merchant record */
        sprintf(buffer, "%%%%m%s##%%%%p%s##",
                merchantname, password);

        printf("\n\nSwipe card, enter amount: ");
        fflush(stdout);
        if (scanf("%s", inbuf) == EOF) {
                printf("bye...\n");
                exit(0);
        }
        soundbytes();

        sprintf(buffer, "%s%%%%a%s##%%%%n%s##",
                buffer, inbuf, swipecard());
```

Example B–4: Connection-Oriented XTI Client Program (cont.)

```
                if (t_snd(xtifd, buffer, strlen(buffer), 0)    8
                    < 0) {
                        xerror("t_snd", xtifd);
                        exit(1);
                }

                if ((n = t_rcv(xtifd, buffer, 1024, &flags))   9
                    < 0) {
                        xerror("t_rcv", xtifd);
                        exit(1);
                }

                buffer[n] = ’\0’;

                if ((n=analyze(buffer))== 0) {
                        printf("transaction failure,"
                               " try again\n");
                } else if (n<0) {
                        printf("login failed, try again\n");
                        init();
                }
        }
        if (t_free((char *)sndcall, T_CALL) < 0) {
                xerror("xti_free", xtifd);
                exit(3);
        }
}
```

1 AF_INET is the socket type for the Internet communication domain. If AF_OSI were supported, it could be used to create a socket for OSI communications. The socket type SOCK_STREAM is specified for TCP or connection-oriented communication.

The t_open call specifies a special device file name instead of the socket address family, socket type, and protocol that the socket call requires.

Contrast the socket call in Section B.1.2 with the t_open call.

2 The serveraddr is of type sockaddr_in, which is dictated by the communication domain of the socket (AF_INET). The socket address for the Internet communication domain contains an Internet address and a 16-bit port number, which uniquely identifies an entity on the network. For the TCP/IP protocol suite, which includes UDP, this is the Internet address of the server and the port number on which it is listening.

Note that the information contained in the sockaddr_in structure is dependent on the address family (or the protocol).

3 AF_INET specifies the Internet communication domain. If AF_OSI were supported, it could be used to create a socket for OSI communications.

4 Obtaining information about the server depends on the protocol or the address family. To get the IP address of the server, you can use the `gethostbyname` routine.

5 SERVER_PORT is defined in the `<common.h>` header file. It is imperative that the same port number be used to connect to the XTI server program. Numbers from 0 through 1024 are reserved.

6 Bind the server address with the `t_bind` function to enable the client to start sending and receiving data.

7 Initiate a connection with the server using the `t_connect` function.

8 Send data with the `t_snd` function.

9 Receive data with the `t_rcv` function.

B.2 Connectionless Programs

This section contains sockets and XTI variations of the same server and client programs, written for connectionless modes of communication.

B.2.1 Socket Server Program

Example B–5 implements the server portion of the application in a manner similar to the socket server described in Section B.1.1. Instead of using a connection-oriented paradigm, this program uses a connectionless (datagram/UDP) paradigm for communicating with client programs. This program has the limitations described at the beginning of the appendix.

Example B–5: Connectionless Socket Server Program

```
/*
 *
 * This file contains the main socket server code
 * for a connectionless mode of communication.
 *
 * Usage:       socketserverDG
 *
 */
#include "server.h"

char                    *parse(char *);
struct transaction      *verifycustomer(char *, int, char *);

main(int argc, char *argv[])
{
        int                     sockfd;
        int                     newsockfd;
        struct sockaddr_in      serveraddr;
        int                     serveraddrlen = sizeof(serveraddr);
        struct sockaddr_in      clientaddr;
        int                     clientaddrlen = sizeof(clientaddr);
        struct hostent          *he;
        int                     pid;

        signal(SIGCHLD, SIG_IGN);

        /* Create a socket for the communications */
        if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0))       1
            < 0) {
                perror("socket_create");
                exit(1);
        }

        bzero((char *) &serveraddr,                         2
              sizeof(struct sockaddr_in));
        serveraddr.sin_family      = AF_INET;
        serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);     3
        serveraddr.sin_port        = htons(SERVER_PORT);    4

        if ( bind(sockfd,                                   5
                  (struct sockaddr *)&serveraddr,
                  sizeof(struct sockaddr_in)) < 0) {
                perror("socket_bind");
                exit(2);
        }

        transactions(sockfd);
```

Example B–5: Connectionless Socket Server Program (cont.)

```
}

transactions(int fd)                                         6
{
        int                   bytes;
        char                  *reply;
        int                   dcount;
        char                  datapipe[MAXBUFSIZE+1];
        struct sockaddr_in    serveraddr;
        int                   serveraddrlen = sizeof(serveraddr);

        bzero((char *) &serveraddr, sizeof(struct sockaddr_in));
        serveraddr.sin_family      = AF_INET;
        serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
        serveraddr.sin_port        = htons(SERVER_PORT);

        /* Look at the data buffer and parse commands.
         * Keep track of the collected data through
         * transaction_status.
         */
         while (1) {
                if ((dcount=recvfrom(fd, datapipe,           7
                                MAXBUFSIZE, 0,
                                (struct sockaddr *)&serveraddr,
                                &serveraddrlen)) < 0){
                        perror("transactions_receive");
                        break;
                }
                if (dcount == 0) {
                        return(0);
                }

                datapipe[dcount] = '\0';

                if ((reply=parse(datapipe)) != NULL) {
                        if (sendto(fd, reply, strlen(reply), 8
                            0,
                            (struct sockaddr *)&serveraddr,
                            serveraddrlen) < 0) {
                                perror("transactions_sendto");
                        }
                }
         }
}
```

1 Create a socket with the `socket` call.

AF_INET specifies the Internet communication domain. The socket type SOCK_DGRAM is specified for UDP or connectionless communication. This parameter indicates that the program is connectionless.

Contrast the `socket` call with the `t_open` call in the XTI server example (Section B.2.3).

2 The `serveraddr` is of type `sockaddr_in`, which is dictated by the communication domain of the socket (AF_INET). The socket address for the Internet communication domain contains an Internet address and a 16-bit port number, which uniquely identifies an entity on the network. For the TCP/IP protocol suite, which includes UDP, this is the Internet address of the server and the port number on which it is listening.

The information contained in the `sockaddr_in` structure is dependent on the address family, which is AF_INET in this example. If AF_OSI were used instead of AF_INET, then `sockaddr_osi` would be required for the `bind` call instead of `sockaddr_in`.

3 INADDRANY signifies any attached interface adapter on the system. All numbers must be converted to the network format using appropriate macros. See the following reference pages for more information: `htonl`(3), `htons`(3), `ntohl`(3), and `ntohs`(3).

4 SERVER_PORT is defined in the `<common.h>` header file. It has a data type of `short integer` which helps identify the server process from other application processes.

5 Bind the server's address to this socket with the `bind` call. The combination of the address and port number identify it uniquely on the network.

After the server process' address is bound, the server process is registered on the system and can be identified by the lower level kernel functions to which to direct requests.

6 Each incoming message packet is accepted and passed to the `parse` function, which tracks the information provided (such as the merchant's login ID, password, and customer's credit card number). This process is repeated until the `parse` function identifies a complete transaction and returns a response packet, to be sent to the client program.

Since this program uses a connectionless (datagram) protocol, it uses `sendto` and `recvfrom` to send and receive data, respectively.

7 Receive data with the `recvfrom` call.

8 Send data with the `sendto` call.

B.2.2 Socket Client Program

Example B–6 implements a socket client that can communicate with the socket server in Example B–5. Section B.2.1. It uses the socket interface in the connectionless, or datagram, mode.

Example B–6: Connectionless Socket Client Program

```
/*
 *
 * This file contains the main client socket code
 * for a connectionless mode of communication.
 *
 * usage: socketclientDG [serverhostname]
 *
 * If a host name is not specified, the local
 * host is assumed.
 *
 */
#include "client.h"

main(int argc, char *argv[])
{
        int                     sockfd;
        struct sockaddr_in      serveraddr;
        int                     serveraddrlen;
        struct hostent          *he;
        int                     n;
        char                    *serverhost = "localhost";
        struct hostent          *serverhostp;
        char                    buffer[1024];
        char                    inbuf[1024];


        if (argc>1) {
                serverhost = argv[1];
        }

        init();


        /* Create a socket for the communications */
        if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)    1
        {
                perror("socket_create");
                exit(1);
        }


        bzero((char *) &serveraddr,                           2
```

Example B–6: Connectionless Socket Client Program (cont.)

```
      sizeof(struct sockaddr_in));
serveraddr.sin_family      = AF_INET;

if ((serverhostp = gethostbyname(serverhost)) ==      3
    (struct hostent *)NULL) {
        fprintf(stderr,"gethostbyname on %s failed\n",
                serverhost);
        exit(1);
}
bcopy(serverhostp->h_addr,
      (char *)&(serveraddr.sin_addr.s_addr),
      serverhostp->h_length);

serveraddr.sin_port        = htons(SERVER_PORT);      4

/* Now connect to the server
if (connect(sockfd, (struct sockaddr *)&serveraddr,  5
            sizeof(serveraddr)) < 0) {
        perror ("connect");
        exit(2);
}
*/


while(1) {
        /* Merchant record */
        sprintf(buffer, "%%%%m%s##%%%%p%s##",
                merchantname, password);

        printf("\n\nSwipe card, enter amount: ");
        fflush(stdout);
        if (scanf("%s", inbuf) == EOF) {
                printf("bye...\n");
                exit(0);
        }
        soundbytes();

        sprintf(buffer, "%s%%%%a%s##%%%%n%s##",
                buffer, inbuf, swipecard());

        if (sendto(sockfd, buffer, strlen(buffer),0,    6
                  (struct sockaddr *)&serveraddr,
                  sizeof(serveraddr)) < 0) {
                perror("sendto");
                exit(1);
        }
```

Example B–6: Connectionless Socket Client Program (cont.)

```
        /* receive info */
        if ((n = recvfrom(sockfd, buffer, 1024, 0,   7
                 (struct sockaddr *)&serveraddr,
                 &serveraddrlen)) < 0) {
            perror("recvfrom");
            exit(1);
        }
        buffer[n] = '\0';

        if ((n=analyze(buffer))== 0) {
            printf("transaction failure, "
                "try again\n");
        } else if (n<0) {
            printf("login failed, try again\n");
            init();
        }
    }

}
```

1 Create a socket with the `socket` call.

AF_INET specifies the Internet communication domain. If AF_OSI were supported, it could be used to create a socket for OSI communications. The socket type SOCK_DGRAM is specified for UDP or connectionless communication.

Contrast the `socket` call with the `t_open` call in the XTI client example (Section B.2.4).

2 The `serveraddr` is of type `sockaddr_in`, which is dictated by the communication domain of the socket (AF_INET). The socket address for the Internet communication domain contains an Internet address and a 16-bit port number, which uniquely identifies an entity on the network. For the TCP/IP protocol suite, which includes UDP, this is the Internet address of the server and the port number on which it is listening.

Note that the information contained in the `sockaddr_in` structure is dependent on the address family (or the protocol).

3 Getting information about the server depends on the protocol or the address family. To get the IP address of the server, you can use the `gethostbyname` routine.

4 SERVER_PORT is defined in the `<common.h>` header file. It is a short integer, which helps identify the server process from other application processes.

5 Client issues a connect call to connect to the server. When the connect call is used with a connectionless protocol, it allows the client to store the server's address locally. This means that the client does not have to specify the server's address each time it sends a message.

6 Send data with the sendto call.

7 Receive data with the recvfrom call.

B.2.3 XTI Server Program

Example B–7 implements a server using the XTI library for network communication. It is an alternative design for a communication program that makes it transport independent. Compare this program with the socket server program in Example B–5. This program has the limitations described at the beginning of the appendix.

Example B–7: Connectionless XTI Server Program

```
/*
 *
 * This file contains the main XTI server code
 * for a connectionless mode of communication.
 *
 * Usage:       xtiserverDG
 *
 */
#include "server.h"

char                    *parse(char *);
struct transaction      *verifycustomer(char *, int, char *);

main(int argc, char *argv[])
{
        int                     xtifd;
        int                     newxtifd;
        struct sockaddr_in      serveraddr;
        struct hostent          *he;
        int                     pid;
        struct t_bind           *bindreqp;


        signal(SIGCHLD, SIG_IGN);


        /* Create a transport endpoint for the communications */
        if ((xtifd = t_open("/dev/streams/xtiso/udp+",          1
                            O_RDWR, NULL)) < 0) {
                xerror("xti_open", xtifd);
                exit(1);
```

Example B–7: Connectionless XTI Server Program (cont.)

```
}


bzero((char *) &serveraddr,                           2
      sizeof(struct sockaddr_in));
serveraddr.sin_family      = AF_INET;                 3
serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);       4
serveraddr.sin_port        = htons(SERVER_PORT)       5

/* allocate structures for the t_bind call */
if ((bindreqp=(struct t_bind *)t_alloc(xtifd,
                                        T_BIND,
                                        T_ALL))
        == NULL) {
        xerror("xti_alloc", xtifd);
        exit(3);
}


bindreqp->addr.buf      = (char *)&serveraddr;
bindreqp->addr.len      = sizeof(serveraddr);

/*
 * Specify how many pending connections can be
 * maintained, while we finish "accept" processing
 *
 */
bindreqp->qlen          = 8;                          6

if (t_bind(xtifd, bindreqp, (struct t_bind *)NULL)    7
    < 0) {
        xerror("xti_bind", xtifd);
        exit(4);
}

/*
 * Now the server is ready to accept connections
 * on this socket.  For each connection, fork a child
 * process in charge of the session, and then resume
 * accepting connections.
 *
 */
transactions(xtifd);

if (t_free((char *)bindreqp, T_BIND) < 0) {
     xerror("xti_free", xtifd);
     exit(3);
```

Example B–7: Connectionless XTI Server Program (cont.)

```
        }
}


transactions(int fd)                                         8
{
        int                     bytes;
        char                    *reply;
        int                     dcount;
        int                     flags;
        char                    datapipe[MAXBUFSIZE+1];
        struct t_unitdata       *unitdata;
        struct sockaddr_in      clientaddr;

        /* Allocate structures for t_rcvudata and t_sndudata call */
        if ((unitdata=(struct t_unitdata *)t_alloc(fd,
                                                   T_UNITDATA,
                                                   T_ALL))
                == NULL) {
                xerror("xti_alloc", fd);
                exit(3);
        }

        /*
         * Look at the data buffer and parse commands.
         * If more data required, go get it.
         *
         */
         while (1) {
                unitdata->udata.maxlen   = MAXBUFSIZE;
                unitdata->udata.buf      = datapipe;
                unitdata->addr.maxlen    = sizeof(clientaddr);
                unitdata->addr.buf       = (char *)&clientaddr;
                unitdata->opt.maxlen     = 0;

                if ((dcount=t_rcvudata(fd, &unitdata, &flags)) 9
                    < 0) {
                        /* if disconnected bid a goodbye */
                        if (t_errno == TLOOK) {
                                int tmp = t_look(fd);

                                if (tmp != T_DISCONNECT) {
                                        t_scope(tmp);
                                } else {
                                        exit(0);
                                }
                        }
```

Example B–7: Connectionless XTI Server Program (cont.)

```
                        xerror("transactions_receive", fd);
                        break;
                }
                if (unitdata->udata.len == 0) {

                        return(0);
                }

                datapipe[unitdata->udata.len] = '\0';

                if ((reply=parse(datapipe)) != NULL) {

                        /* sender's addr is in the unitdata */
                        unitdata->udata.len = strlen(reply);
                        unitdata->udata.buf = reply;

                        if (t_sndudata(fd, unitdata) < 0) { 10
                                xerror("xti_send", fd);
                                break;
                        }
                }
         }
         if (t_free((char *)unitdata, T_UNITDATA) < 0) {
              xerror("xti_free", fd);
              exit(3);
         }
}
```

1 The t_open call specifies a device special file name, which is /dev/streams/xtiso/udp+ in this example. This file name provides the necessary abstraction for the UDP transport protocol over IP. Unlike the socket interface, where you specify the address family (for example, AF_INET), this information is already represented in the choice of the device special file. The /dev/streams/xtiso/udp+ file implies both UDP transport and Internet Protocol. See the Chapter 5 for information about STREAMS devices. Contrast the t_open call with the socket call in Section B.2.1.

2 The serveraddr is of type sockaddr_in, which is dictated by the communication domain or address family of the socket (AF_INET). The socket address for the Internet communication domain contains an Internet address and a 16-bit port number, which uniquely identifies an application entity on the network. For TCP/IP and UDP/IP this is the Internet address of the server and the port number on which it is listening.

The information contained in the `sockaddr_in` structure is dependent on the address family (or the protocol).

3 AF_INET specifies the Internet communication domain or address family.

4 INADDRANY signifies any attached interface adapter on the system. All numbers must be converted to the network format using appropriate macros. See the following reference pages for more information: `htonl`(3), `htons`(3), `ntohl`(3), and `ntohs`(3).

5 SERVER_PORT is defined in the `<common.h>` header file. It is a short integer, which helps identify the server process from other application processes. Numbers from 0 to 124 are reserved.

6 Specify the number of pending connections the server can queue while it processes the last request.

7 Bind the server's address with the `t_bind` call. The combination of the address and port number identify it uniquely on the network. After the server process' address is bound, the server process is registered on the system and can be identified by the lower level kernel functions to which to direct any requests.

8 Each incoming message packet is accepted and passed to the `parse` function, which tracks the information provided, such as the merchant's login ID, password, and customer's credit card number. This process is repeated until the `parse` function identifies a complete transaction and returns a response packet, to be sent to the client program.

The client program can send information packets in any order (and in one or more packets), so the `parse` function is designed to remember state information sufficient to deal with this unstructured message stream.

Since this program uses a connectionless (datagram) protocol, it uses `t_sndudata` and `t_rcvudata` to send and receive data, respectively.

9 Receive data with the `t_rcvudata` function.

10 Send data with the `t_sndudata` function.

B.2.4 XTI Client Program

Example B–8 implements an XTI client that can communicate with the XTI server in Example B–7. It uses the XTI interface in the connectionless, or datagram, mode.

Example B–8: Connectionless XTI Client Program

```
/*
 *
 * This file contains the main XTI client code
 * for a connectionless mode of communication.
 *
 * usage: client [serverhostname]
 *
 */
#include "client.h"

main(int argc, char *argv[])
{
        int                     xtifd;
        struct sockaddr_in      serveraddr;
        struct hostent          *he;
        int                     n;
        char                    *serverhost = "localhost";
        struct hostent          *serverhostp;
        char                    buffer[MAXBUFSIZE+1];
        char                    inbuf[MAXBUFSIZE+1];
        struct t_unitdata       *unitdata;
        int                     flags = 0;


        if (argc>1) {
                serverhost = argv[1];
        }

        init();


        if ((xtifd = t_open("/dev/streams/xtiso/udp+",      1
                            O_RDWR, NULL)) < 0) {
                xerror("xti_open", xtifd);
                exit(1);
        }


        bzero((char *) &serveraddr,                         2
              sizeof(struct sockaddr_in));
        serveraddr.sin_family      = AF_INET;               3

        if ((serverhostp = gethostbyname(serverhost)) ==    4
            (struct hostent *)NULL) {
                fprintf(stderr,"gethostbyname on %s failed\n",
                        serverhost);
                exit(1);
        }
```

Example B–8: Connectionless XTI Client Program (cont.)

```
bcopy(serverhostp->h_addr,
      (char *)&(serveraddr.sin_addr.s_addr),
      serverhostp->h_length);
/*
 * SERVER_PORT is a short which identifies
 * the server process from other sources.
 *
 */
serveraddr.sin_port        = htons(SERVER_PORT);     5

if (t_bind(xtifd, (struct t_bind *)NULL,            6
           (struct t_bind *)NULL) < 0) {
        xerror("bind", xtifd);
        exit(2);
}

/* Allocate structures for t_rcvudata and t_sndudata call */
if ((unitdata=(struct t_unitdata *)t_alloc(xtifd,
                                           T_UNITDATA,
                                           T_ALL))
        == NULL) {
        xerror("xti_alloc", fd);
        exit(3);
}

while(1) {
        /* Merchant record */
        sprintf(buffer, "%%%%m%s##%%%%p%s##",
                merchantname, password);

        printf("\n\nSwipe card, enter amount: ");
        fflush(stdout);

        if (scanf("%s", inbuf) == EOF) {
                printf("bye...\n");
                exit(0);
        }

        soundbytes();

        sprintf(buffer, "%s%%%%a%s##%%%%n%s##",
                buffer, inbuf, swipecard());

        unitdata->addr.buf       = (char *)&serveraddr;
        unitdata->addr.len       = sizeof(serveraddr);
        unitdata->udata.buf      = buffer;
        unitdata->udata.len      = strlen(buffer);
```

Example B–8: Connectionless XTI Client Program (cont.)

```
                unitdata->opt.len        = 0;

                if (t_sndudata(xtifd, unitdata) < 0) {     7
                        xerror("t_snd", xtifd);
                        exit(1);
                }

                unitdata->udata.maxlen   = MAXBUFSIZE;
                unitdata->addr.maxlen    = sizeof(serveraddr);

                /* receive info */
                if ((t_rcvudata(xtifd, unitdata, &flags))  8
                    < 0) {
                        xerror("t_rcv", xtifd);
                        exit(1);
                }

                buffer[unitdata->udata.len] = ’\0’;

                if ((n=analyze(buffer))== 0) {
                        printf("transaction failure, "
                               "try again\n");
                } else if (n<0) {
                        printf("login failed, try again\n");
                        init();
                }
        }
         if (t_free((char *)unitdata, T_UNITDATA) < 0) {
              xerror("xti_free", fd);
              exit(3);
         }
}
```

1 The t_open call specifies a device special file name; for example /dev/streams/xtiso/udp+. This file name provides the necessary abstraction for the UDP transport protocol over IP. Unlike the socket interface, where you specify the address family (for example, AF_INET), this information is already represented in the choice of the device special file. The /dev/streams/xtiso/udp+ file implies both UDP transport and Internet Protocol. See the Chapter 5 for information about STREAMS devices.

Contrast the t_open call with the socket call in Section B.2.2.

2 The serveraddr is of type sockaddr_in, which is dictated by the communication domain of the socket (AF_INET). The socket address for

the Internet communication domain contains an Internet address and a 16-bit port number, which uniquely identifies an entity on the network. For the TCP/IP protocol suite, which includes UDP, this is the Internet address of the server and the port number on which it is listening.

The information contained in the `sockaddr_in` structure is dependent on the address family (or the protocol).

3 AF_INET specifies the Internet communication domain. If AF_OSI were supported it could be used to create a socket for OSI communications.

4 Getting information about the server depends on the protocol or the address family. To get the IP address of the server, you can use the `gethostbyname`(3) routine.

5 SERVER_PORT is defined in the `<common.h>` header file. It is a short integer, which helps identify the server process from other application processes.

6 Bind the server address with the `t_bind` function to enable the client to start sending and receiving data.

7 Send data with the `t_sndudata` function.

8 Receive data with the `t_rcvudata` function.

B.3 Common Code

The following header and database files are required for all or several of the client and server portions of this application:

- `<common.h>`
- `<server.h>`
- `serverauth.c`
- `serverdb.c`
- `xtierror.c`
- `<client.h>`
- `clientauth.c`
- `clientdb.c`

B.3.1 The common.h Header File

Example B–9 shows the `<common.h>` header file. It contains common header files and constants required by all sample programs.

Example B–9: The common.h Header File

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/errno.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>            1
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <fcntl.h>
#include <xti.h>


#define SEPARATOR       ’,’
#define PREAMBLE        "%%"
#define PREAMBLELEN     2      2
#define POSTAMBLE       "##"
#define POSTAMBLELEN    2


/* How to contact the server */
#define SERVER_PORT     1234   3
/* How to contact the client (for datagram only) */
#define CLIENT_PORT     1235

#define MAXBUFSIZE 4096
```

1 List of header files to include.

2 These statements define constants that allow more effective parsing of data exchanged between the server and client.

3 SERVER_PORT is a well known port that is arbitrarily assigned by the programmer so that clients can communicate with the server. SERVER_PORT is used to identify the service to which you want to connect. Port numbers 0 through 1024 are reserved for the system. Programmers can choose a number, as long as it does not conflict with any other applications. While debugging, this number is chosen randomly (and by trial and error). For a well-distributed application, some policy must be used to avoid conflicts with other applications.

B.3.2 The server.h Header File

Example B–10 shows the `<server.h>` header file. It contains the data structures for accessing the server’s database, as well as the data structures for analyzing and synthesizing messages to and from clients.

Example B–10: The server.h Header File

```
#include "common.h"

struct merchant {
        char *name;
        char *passwd;
};

struct customer {
        char                    *cardnum;
        char                    *name;
        int                     limit;
        int                     balance;
        struct transaction      *tlist;
        /* presumably other data */
};

struct transaction {
        struct transaction      *nextcust;
        struct transaction      *nextglob;
        struct customer         *whose;
        char                    *merchantname;
        int                     amount;
        char                    *verification;
};

extern struct transaction *alltransactions;
extern struct merchant  merchant[];
extern int              merchantcount;
extern struct customer  customer[];
extern int              customercount;

#define INVALID (struct transaction *)1

#define MERCHANTAUTHERROR       "%%A##"
#define USERAUTHERROR           "%%U##"
#define USERAMOUNTERROR         "%%V##"
#define TRANSMITERROR           "deadbeef"

/* define transaction_status flags */
#define NAME                    0x01
#define PASS                    0x02
#define AMOUNT                  0x04
#define NUMBER                  0x08

#define AUTHMASK                0x03
#define VERIMASK                0x0C
```

B.3.3 The serverauth.c File

Example B–11 shows the `serverauth.c` file.

Example B–11: The serverauth.c File

```
/*
 *
 * Authorization information (not related to the
 * networking interface)
 *
 */

#include "server.h"


/*
 * Currently a simple non-encrypted password method to search db
 *
 */
authorizemerchant(char *merch, char *password)
{
        struct merchant *mp;

        for(mp = merchant; (mp)->name != (char *)NULL; mp++) {
                if (!strcmp(merch, (mp)->name)) {
                        return (!strcmp(password, (mp)->passwd));
                }
        }
        return(0);
}


struct transaction *
verifycustomer(char *num, int amount, char *merchant)
{
        char buf[64];
        struct customer         *cp;
        struct transaction      *tp;

        for(cp = customer; (cp)->cardnum != NULL; cp++) {
                if (!strcmp(num, (cp)->cardnum)) {
                        if (amount <= (cp)->balance) {
                                (cp)->balance -= amount;
                                if ((tp = malloc(sizeof(
                                           struct transaction)))
                                    == NULL) {
                                        printf("Malloc error\n");
                                        return(NULL);
                                }
```

Example B–11: The serverauth.c File (cont.)

```
                                tp->merchantname = merchant;
                                tp->amount = amount;
                                sprintf(buf, "v%012d", time(0));
                                if ((tp->verification =
                                     malloc(strlen(buf)+1))
                                    == NULL) {
                                        printf("Malloc err\n");
                                        return(NULL);
                                }
                                strcpy(tp->verification, buf);
                                tp->nextcust = cp->tlist;
                                tp->whose = cp;
                                cp->tlist = tp;
                                tp->nextglob = alltransactions;
                                alltransactions = tp;
                                return(tp);
                        } else {
                                return(NULL);
                        }
                }
        }
        return(INVALID);

}

int                     transaction_status;
int                     authorized = 0;
int                     amount = 0;
char                    number[256];
char                    Merchant[256];
char                    password[256];

char *
parse(char *cp)
{
        char            *dp, *ep;
        unsigned char   type;
        int             doauth = 0;
        char            *buffer;

        dp = cp;
        if ((buffer=malloc(256)) == NULL) {
                return(TRANSMITERROR);
        }
        while (*dp) {
                /* terminate the string at the postamble */
                if (!(ep=strstr(dp, POSTAMBLE))) {
```

Example B–11: The serverauth.c File (cont.)

```
        return(TRANSMITERROR);
}
*ep = '\0';
ep = ep + POSTAMBLELEN;                1

/* search for preamble */
if (!(dp=strstr(dp, PREAMBLE))) {
        return(TRANSMITERROR);
}
dp += PREAMBLELEN;

/* Now get the token */
type = *dp++;

switch(type) {
        case 'm':
                strcpy(Merchant, dp);
                transaction_status |= NAME;
                break;
        case 'p':
                strcpy(password, dp);
                transaction_status |= PASS;
                break;
        case 'n':
                transaction_status |= NUMBER;
                strcpy(number, dp);
                break;
        case 'a':
                transaction_status |= AMOUNT;
                amount = atoi(dp);
                break;
        default:
                printf("Bad command\n");
                return(TRANSMITERROR);
}
if ((transaction_status & AUTHMASK) == AUTHMASK) {
        transaction_status &= ~AUTHMASK;
        authorized = authorizemerchant(
                      Merchant, password);
        if (!authorized) {
                printf("Merchant not"
                       " authorized\n");
                return(MERCHANTAUTHERROR);
        }
}
/* If both amount and number gathered,
 * do verification     */
```

Example B–11: The serverauth.c File (cont.)

```
                if ((authorized) &&
                    ((transaction_status&VERIMASK)
                     ==VERIMASK)) {
                        struct transaction *tp;
                        transaction_status &= ~VERIMASK;
                        /* send a verification back */
                        if ((tp=verifycustomer(number,
                                               amount,
                                               Merchant))
                            == NULL) {
                                return(USERAMOUNTERROR);
                        } else if (tp==INVALID) {
                                return(USERAUTHERROR);
                        } else {
                                sprintf(buffer,
                                   "%%%%%s##%%%%c%s##%%%%m%s##",
                                   tp->verification,
                                   tp->whose->name,
                                   tp->merchantname);
                                return(buffer);
                        }
                }
                dp = ep;
        }
        return(NULL);
}
```

[1] This function parses the incoming data, which includes the merchant authorization information, customer's credit card number, and the amount the customer is charging. Note that the function can not assume that all of the information is available in one message because the underlying TCP protocol is stream-oriented. This function can be simplified if a datagram type service is used or if a protocol that uses sequenced packets (SEQPACKET) is used. The function is designed to accept pieces of information in any order and in one or more message blocks.

B.3.4 The serverdb.c File

Example B–12 shows the `serverdb.c` file.

Example B–12: The serverdb.c File

```
/*
 *
 * Database of valid merchants and credit card customers with the
 * credit limits, etc.
 *
 *
 */
#include "server.h"

struct merchant merchant[] = {
        {"abc",                 "abc"},
        {"magic",               "magic"},
        {"gasco",               "gasco"},
        {"furnitureco",         "abc"},
        {"groceryco",           "groceryco"},
        {"bakeryco",            "bakeryco"},
        {"restaurantco",        "restaurantco"},
        {NULL,                  NULL}
};

int merchantcount = sizeof(merchant)/sizeof(struct merchant)-1;

struct customer customer[] = {
        { "4322546789701000", "John Smith",     1000,   800 },
        { "4322546789701001", "Bill Stone",     2000,   200 },
        { "4322546789701002", "Dave Adams",     1500,   500 },
        { "4322546789701003", "Ray Jones",      1200,   800 },
        { "4322546789701004", "Tony Zachry",    1000,   100 },
        { "4322546789701005", "Danny Einstein", 5000,   50  },
        { "4322546789701006", "Steve Simonyi",  10000,  5800},
        { "4322546789701007", "Mary Ming",      1100,   700 },
        { "4322546789701008", "Joan Walters",   800,    780 },
        { "4322546789701009", "Gail Newton",    1000,   900 },
        { "4322546789701010", "Jon Robertson",  1000,   1000},
        { "4322546789701011", "Ellen Bloop",    1300,   800 },
        { "4322546789701012", "Sue Svelter",    1400,   347 },
        { "4322546789701013", "Suzette Ring",   1200,   657 },
        { "4322546789701014", "Daniel Mattis",  1600,   239 },
        { "4322546789701015", "Robert Esconis", 1800,   768 },
        { "4322546789701016", "Lisa Stiles",    1100,   974 },
        { "4322546789701017", "Bill Brophy",    1050,   800 },
        { "4322546789701018", "Linda Smitten",  4000,   200 },
        { "4322546789701019", "John Norton",    1400,   900 },
        { "4322546789701020", "Danielle Smith", 2000,   640 },
        { "4322546789701021", "Amy Olds",       1300,   100 },
        { "4322546789701022", "Steve Smith",    2000,   832 },
        { "4322546789701023", "Robert Smart",   3000,   879 },
        { "4322546789701024", "Jon Harris",     500,    146 },
```

Example B–12: The serverdb.c File (cont.)

```
        { "4322546789701025", "Adam Gershner",  1600,   111 },
        { "4322546789701026", "Mary Papadimis", 2000,   382 },
        { "4322546789701027", "Linda Jones",    1300,   578 },
        { "4322546789701028", "Lucy Barret",    1400,   865 },
        { "4322546789701029", "Marie Gilligan", 1000,   904 },
        { "4322546789701030", "Kim Coyne",      3000,   403 },
        { "4322546789701031", "Mike Storm",     7500,   5183},
        { "4322546789701032", "Cliff Clayden",  750,    430 },
        { "4322546789701033", "John Turing",    4000,   800 },
        { "4322546789701034", "Jane Joyce",     10000,  8765},
        { "4322546789701035", "Jim Roberts",    4000,   3247},
        { "4322546789701036", "Stevw Stephano", 1750,   894 },
        {NULL, NULL}
};

struct transaction *
alltransactions   = NULL;
int customercount = sizeof(customer)/sizeof(struct customer)-1;
```

B.3.5 The xtierror.c File

Example B–13 shows the xtierror.c file. It is used to generate a descriptive message in case of an error. Note that for asynchronous errors or events the t_look function is used to get more information.

Example B–13: The xtierror.c File

```
#include <xti.h>
#include <stdio.h>

void t_scope();

void
xerror(char *marker, int fd)
{
        fprintf(stderr, "%s error [%d]\n", marker, t_errno);
        t_error("Transport Error");
        if (t_errno == TLOOK) {
                t_scope(t_look(fd));
        }

}

void
t_scope(int tlook)
{
        char *tmperr;

        switch(tlook) {
                case T_LISTEN:
                        tmperr = "connection indication";
                        break;
                case T_CONNECT:
                        tmperr = "connect confirmation";
                        break;
                case T_DATA:
                        tmperr = "normal data received";
                        break;
                case T_EXDATA:
                        tmperr = "expedited data";
                        break;
                case T_DISCONNECT:
                        tmperr = "disconnect received";
                        break;
                case T_UDERR:
                        tmperr = "datagram error";
                        break;
                case T_ORDREL:
                        tmperr = "orderly release indication";
```

Example B–13: The xtierror.c File (cont.)

```
                        break;
                case T_GODATA:
                        tmperr = "flow control restriction lifted";
                        break;
                case T_GOEXDATA:
                        tmperr = "flow control restriction "
                                 "on expedited data lifted";
                        break;
                default:
                        tmperr = "unknown event";
        }
                        fprintf(stderr,
                                "Asynchronous event: %s\n",
                                tmperr);
}
```

B.3.6 The client.h Header File

Example B–14 shows the client.h header file.

Example B–14: The client.h File

```
#include "common.h"

extern char     merchantname[];
extern char     password[];
```

B.3.7 The clientauth.c File

Example B–15 shows the clientauth.c file. It contains the code that obtains the merchant's authorization, as well as the logic to analyze the message sent from the server. The resulting message is interpreted to see if the authorization was granted or rejected by the server.

Example B–15: The clientauth.c File

```
#include "client.h"

init()
{
        printf("\nlogin: "); fflush(stdout);
        scanf("%s", merchantname);

        printf("Password: "); fflush(stdout);
```

Example B–15: The clientauth.c File (cont.)

```
        scanf("%s", password);

        srandom(time(0));
}

/* simulate some network activity via sound */
soundbytes()
{
        int i;

        for(i=0;i<11;i++) {
                printf();
                fflush(stdout);
                usleep(27000*(random()%10+1));
        }
}

analyze(char *cp)
{

        char *dp, *ep;
        unsigned char type;
        char customer[128];
        char verification[128];

        customer[0] = verification[0] = ’\0’;

        dp = cp;

        while ((dp!=NULL) && (*dp)) {
                /* terminate the string at the postamble */
                if (!(ep=strstr(dp, POSTAMBLE))) {
                        return(0);
                }
                *ep = ’\0’;
                ep = ep + POSTAMBLELEN;

                /* search for preamble */
                if (!(dp=strstr(dp, PREAMBLE))) {
                        return(0);
                }
                dp += PREAMBLELEN;

                /* Now get the token */
                type = *dp++;

                switch(type) {
```

Example B–15: The clientauth.c File (cont.)

```
                        case ’m’:
                                if (strcmp(merchantname, dp)) {
                                        return(0);
                                }
                                break;
                        case ’c’:
                                strcpy(customer, dp);
                                break;
                        case ’U’:
                                printf("Authorization denied\n");
                                return(1);
                        case ’V’:
                                printf("Amount exceeded\n");
                                return(1);
                        case ’A’:
                                return(-1);
                        case ’v’:
                                strcpy(verification, dp);
                                break;
                        default:
                                return(0);
                }
                dp = ep;
        }
        if (*customer && *verification) {
                printf("%s, verification ID: %s\n",
                       customer, verification);
                return(1);
        }
        return(0);
}
```

B.3.8 The clientdb.c File

Example B–16 shows the `clientdb.c` file. It contains a database of customer credit card numbers used to simulate the card swapping action. In a real world application, a magnetic reader reads the numbers through an appropriate interface. Also, the number cache is not required for a real world application.

Example B–16: The clientdb.c File

```
/*
 *
 * Database of customer credit card numbers to simulate
 * the card swapping action. In practice the numbers
 * will be read by magnetic readers through an
 * appropriate interface.
 */

#include <time.h>

char    merchantname[256];
char    password[256];


char *numbercache[] = {
        "4322546789701000",
        "4322546789701001",
        "4322546789701002",
        "4222546789701002",             /* fake id */
        "4322546789701003",
        "4322546789701004",
        "4322546789701005",
        "4322546789701006",
        "4322546789701007",
        "4322546789701008",
        "4322546789701009",
        "4322546789701010",
        "4322546789701011",
        "4322546789701012",
        "4322546789701013",
        "4322546789701014",
        "4322546789701015",
        "4322546789701016",
        "4322546789701017",
        "4322546789701018",
        "4222546789701018",             /* fake id */
        "4322546789701019",
        "4322546789701020",
        "4322546789701021",
        "4322546789701022",
        "4322546789701023",
        "4322546789701024",
        "4322546789701025",
        "2322546789701025",             /* fake id */
        "4322546789701026",
        "4322546789701027",
        "4322546789701028",
        "4322546789701029",
```

Example B–16: The clientdb.c File (cont.)

```
        "4322546789701030",
        "4322546789701031",
        "4322546789701032",
        "4322546789701033",
        "4322546789701034",
        "4322546789701035",
        "4322546789701036",
};

#define CACHEENTRIES (sizeof(numbercache)/sizeof(char *))


char *
swipecard()
{
        return(numbercache[random()%CACHEENTRIES]);
}
```

C

IPv4 and IPv6 Sockets Programming Examples

This appendix contains annotated files for a sample client/server program. Clients send a request to the server and print the server's response. Servers listen for client requests, print the request, and send a response to the client.

Although the program is not a real world application, it is structured demonstrate the sequence and use of the various socket calls.

The information is organized as follows:

- AF_INET sockets programs
  - Client
  - Server
- AF_INET6 sockets programs
  - Client
  - Server

You can find these programs and a protocol-independent client in the/usr/examples/ipv6/network_programming directory. In addition, this section also contains sample output from these programs.

C.1 Programs Using AF_INET Sockets

This section contains a client and server program that use AF_INET sockets.

C.1.1 Client Program Using AF_INET Sockets

Example C–1 shows a sample client program that you can build, compile, and run on your system. The program sends a request to and receives a response from the system specified on the command line.

Example C–1: Client Stub Routine

```
/*
 * *****************************************************************
 * *                                                               *
 * *    Copyright (c) Compaq Computer Corporation, 2000            *
 * *                                                               *
 * *   The software contained on this media  is  proprietary  to   *
 * *   and  embodies  the  confidential  technology  of  Compaq    *
```

Example C–1: Client Stub Routine (cont.)

```
 * *   Computer  Corporation.  Possession, use,  duplication  or   *
 * *   dissemination of the software and media is authorized only  *
 * *   pursuant to a valid written license from Compaq Computer    *
 * *   Corporation.                                                *
 * *                                                               *
 * *   RESTRICTED RIGHTS LEGEND   Use, duplication, or disclosure  *
 * *   by the U.S. Government is subject to restrictions  as  set  *
 * *   forth in Subparagraph (c)(1)(ii)  of  DFARS  252.227-7013,  *
 * *   or  in  FAR 52.227-19, as applicable.                       *
 * *                                                               *
 * *****************************************************************
 */
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/errno.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <arpa/inet.h>

#define SERVER_PORT 7639
#define CLIENT_PORT 7739

#define MAXBUFSIZE 4096

int main (
    int argc,
    char **argv )
{
    int             s;
    char            databuf[MAXBUFSIZE];
    int             dcount;
    struct sockaddr_in     serveraddr; 1
    struct sockaddr_in     clientaddr;
    int             serveraddrlen;
    const char      *ap;
    const char      *request = "this is the client's request";
    struct hostent  *hp;
    char            *server;

    if (argc < 2) {
        printf("Usage: client <server>\n");
        exit(1);
    }
    server = argv[1];

    bzero((char *) &serveraddr, sizeof(struct sockaddr_in)); 2
    serveraddr.sin_family = AF_INET;
    if ((hp = gethostbyname(server)) == NULL) { 3
        printf("unknown host: %s\n", server);
        exit(2);
    }
    serveraddr.sin_port = htons(SERVER_PORT);

    while (hp->h_addr_list[0] != NULL) {
        if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0) { 4
            perror("socket");
            exit(3);
```

Example C–1: Client Stub Routine (cont.)

```
    }
    memcpy(&serveraddr.sin_addr.s_addr, hp->h_addr_list[0],
                     hp->h_length);

    if (connect(s, (struct sockaddr *)&serveraddr,
                      sizeof(serveraddr)) < 0) { 5
        perror("connect");
        close(s);
        hp->h_addr_list++;
        continue;
    }
    break;
}
if (send(s, request, strlen(request), 0) < 0) { 6
    perror("send");
    exit(5);
}
dcount = recv(s, databuf, sizeof(databuf), 0); 7
if (dcount < 0) {
    perror("recv");
    exit(6);
}
databuf[dcount] = '\0';

hp = gethostbyaddr((char *)&serveraddr.sin_addr.s_addr, 8
    sizeof(serveraddr.sin_addr.s_addr), AF_INET);
ap = inet_ntoa(serveraddr.sin_addr); 9
printf("Response received from");
```

Example C–1: Client Stub Routine (cont.)

```
    if (hp != NULL)
        printf(" %s", hp->h_name);
    if (ap != NULL)
        printf(" (%s)", ap);
    printf(": %s\n", databuf);

    close(s);
}
```

1. Declares `sockaddr_in` structures. The use of this type of structure is dictated by the communication domain of the socket (AF_INET), which implies communication using the IPv4 protocol.
2. Clears the server address and sets up server variables. The socket address for IPv4 communications is a 32–bit Internet address and a 16–bit port number. This will be the Internet address of the server and the port number on which it is listening.
3. Obtain the server's IPv4 address. A call to `gethostbyname` returns IPv4 address only.
4. Creates an AF_INET socket with a `socket` call. The socket type SOCK_STREAM is specified for TCP or connection-oriented communication.
5. Connects to the server using the address in the `sockaddr_in` structure named `serveraddr`.
6. Sends a request to the server.
7. Receives a response from the server.
8. Retrieves the server name using the address in the `sockaddr_in` structure named `serveraddr`. A call to `gethostbyaddr`expects an IPv4 address as input.
9. Convert the server's 32–bit IPv4 address to a dot-formatted Internet address text string. A call to `inet_ntoa` expects an IPv4 address as input.

C.1.2 Server Program Using AF_INET Sockets

Example C–2 shows a sample server program that you can build, compile, and run on your system. The program receives requests from and sends responses to client programs on other systems.

Example C–2: Server Stub Routine

```
/*
 * *****************************************************************
 * *                                                               *
 * *    Copyright (c) Compaq Computer  Corporation, 2000           *
 * *                                                               *
 * *   The software contained on this media  is  proprietary  to   *
 * *   and  embodies  the  confidential  technology  of  Compaq    *
 * *   Computer  Corporation.  Possession, use,  duplication  or   *
 * *   dissemination of the software and media is authorized only  *
 * *   pursuant to a valid written license from Compaq Computer    *
 * *   Corporation.                                                *
 * *                                                               *
 * *   RESTRICTED RIGHTS LEGEND   Use, duplication, or disclosure  *
 * *   by the U.S. Government is subject to restrictions  as  set  *
 * *   forth in Subparagraph (c)(1)(ii)  of  DFARS  252.227-7013,  *
 * *   or  in  FAR 52.227-19, as applicable.                       *
 * *                                                               *
 * *****************************************************************
 */

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/errno.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <arpa/inet.h>

#define SERVER_PORT 7639
#define CLIENT_PORT 7739

#define MAXBUFSIZE 4096

int main (
    int argc,
    char **argv )
{
    int             s;
    char            databuf[MAXBUFSIZE];
    int             dcount;
    struct sockaddr_in    serveraddr; 1
    struct sockaddr_in    clientaddr;
    int                   clientaddrlen;
    struct hostent        *hp;
    const char            *ap;
    const char            *response = "this is the server's response";
    u_short               port;

    if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0) { 2
        perror("socket");
        exit(1);
    }

    bzero((char *) &serveraddr, sizeof(struct sockaddr_in)); 3
    serveraddr.sin_family      = AF_INET;
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY); 4
    serveraddr.sin_port        = htons(SERVER_PORT);

    if (bind(s, (struct sockaddr *)&serveraddr, sizeof(serveraddr)) < 0) { 5
```

Example C–2: Server Stub Routine (cont.)

```
        perror("bind");
        exit(2);
    }
    if (listen(s, SOMAXCONN) < 0) { 6
        perror("Listen");
        close(s);
        exit(3);

    while (1) {
        int new_s;
        clientaddrlen = sizeof(clientaddr);
        new_s = accept(s, (struct sockaddr *)&clientaddr, &clientaddrlen); 7

        dcount = recv(new_s, databuf, sizeof(databuf), 0); 8
        if (dcount <= 0) {
            perror("recv");
            close(new_s);
            continue;
        }
        databuf[dcount] = ’\0’;
        hp = gethostbyaddr((char *)&clientaddr.sin_addr.s_addr, 9
            sizeof(clientaddr.sin_addr.s_addr), AF_INET);
        ap = inet_ntoa(clientaddr.sin_addr); 10
        port = ntohs(clientaddr.sin_port);
        printf("Request received from");
        if (hp != NULL)
            printf(" %s", hp->h_name);
        if (ap != NULL)
            printf(" (%s)", ap);
        printf(" port %d \"%s\"\n", port, databuf);

        if (send(new_s, response, strlen(response), 0) < 0) { 11
            perror("send");
            close(new_s);
            continue;
        }
        close(new_s);
    }
    close(s);
}
```

1 Declares `sockaddr_in` structures. The use of this type of structure is dictated by the communication domain of the socket (AF_INET), which implies communication using the IPv4 protocol.

2 Creates an AF_INET socket. The socket type SOCK_STREAM is specified for TCP or connection-oriented communication.

3 Clears the server address and sets up server variables. The socket address for IPv4 communications is a 32–bit Internet address and a 16–bit port number. This will be the Internet address of the server and the port number on which it is listening.

4 Sets the server address to the IPv4 wildcard address INADDR_ANY. This signifies any attached network interface on the system.

5 Binds the server's address to the AF_INET socket.

6 Listens on the socket for a connection. The server will queue up to SOMAXCONN pending connections while it finishes processing the previous `accept` call. See `sys_attrs_socket`(5) for more information on the socket subsystem kernel attributes.

7 Accepts a connection on this socket. The `accept` call places the client's address in the `sockaddr_in` structure named `clientaddr`.

8 Receives data from the client.

9 Retrieves the client name using the address in the `sockaddr_in` structure named `clientaddr`. A call to `gethostbyaddr`expects an IPv4 address as input.

10 Converts the client's 32–bit IPv4 address to a dot-formatted Internet address text string. A call to `inet_ntoa` expects an IPv4 address as input.

11 Sends a response to the client.

C.2 Programs Using AF_INET6 Sockets

This section contains a client and server program that use AF_INET6 sockets.

C.2.1 Client Program Using AF_INET6 Sockets

Example C–3 shows a sample client program that you can build, compile, and run on your system. The program sends a request to and receives a response from the system specified on the command line. All addresses are in IPv6 address format.

Example C–3: Client Stub Routine

```
/*
 * ******************************************************************
 * *                                                                *
 * *    Copyright (c) Compaq Computer Corporation, 2000             *
 * *                                                                *
 * *   The software contained on this media  is  proprietary  to    *
 * *   and  embodies  the  confidential  technology  of  Compaq     *
 * *   Computer Corporation.  Possession, use,  duplication  or     *
 * *   dissemination of the software and media is authorized only  *
 * *   pursuant to a valid written license from Compaq Computer     *
 * *   Corporation.                                                 *
 * *                                                                *
 * *   RESTRICTED RIGHTS LEGEND   Use, duplication, or disclosure   *
 * *   by the U.S. Government is subject to restrictions  as  set   *
 * *   forth in Subparagraph (c)(1)(ii)  of  DFARS  252.227-7013,   *
 * *   or  in  FAR 52.227-19, as applicable.                        *
 * *                                                                *
 * ******************************************************************
 */
```

Example C–3: Client Stub Routine (cont.)

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/errno.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <arpa/inet.h>


#define SERVER_PORT 7639
#define CLIENT_PORT 7739

#define MAXBUFSIZE 4096

int main (
    int argc,
    char **argv )
{
    int             s;
    char            databuf[MAXBUFSIZE];
    int             dcount;
    struct addrinfo *server_info;[1]
    struct addrinfo *cur_info;
    struct addrinfo hints;
    struct sockaddr_in6    serveraddr;[2]
    char            addrbuf[INET6_ADDRSTRLEN];
    char            node[MAXDNAME];
    char            service[MAXDNAME];
    int             ni;
    int             err;
    int             serveraddrlen;
    const char      *request = "this is the client’s request";
    char            *server;

    if (argc < 2) {
        printf("Usage: client <server>\n");
        exit(1);
    }
    server = argv[1];
    bzero((char *) &hints, sizeof(hints));[3]
    hints.ai_family = AF_INET6;
    hints.ai_protocol = IPPROTO_TCP;
    hints.ai_flags = AI_ADDRCONFIG | AI_V4MAPPED;

    sprintf(service, "%d", SERVER_PORT);

    err = getaddrinfo(server, service, &hints, &server_info); [4]
    if (err != 0) {
        printf("%s\n", gai_strerror(err));
            if (err == EAI_SYSTEM)
                perror("getaddrinfo");
            exit(2);
    }
    cur_info = server_info;

    while (cur_info != NULL) {
        if ((s = socket(cur_info->ai_family, cur_info->ai_socktype, 0)) < 0) {[5]
            perror("socket");
```

Example C–3: Client Stub Routine (cont.)

```
            freeaddrinfo(server_info);
            exit(3);
        }
        if (connect(s, cur_info->ai_addr, cur_info->ai_addrlen) <0 { 6
            close(s);
            cur_info = cur_info->ai_next;
            continue;
        }
        break;
    }
    freeaddrinfo(server_info);7

    if (send(s, request, strlen(request), 0) < 0) {8
        perror("send");
        exit(5);
    }
    dcount = recv(s, databuf, sizeof(databuf), 0);9
    if (dcount < 0) {
        perror("recv");
        exit(6);
    }
    databuf[dcount] = '\0';
    serveraddrlen = sizeof(serveraddr);
    if (getpeername(s, (struct sockaddr*) &serveraddr, &serveraddrlen) < 0) {10
        perror("getpeername");
        exit(7);
    }
    printf("Response received from");
    ni = getnameinfo((struct sockaddr*)&serveraddr, serveraddrlen,11
                        node, sizeof(node), NULL, 0, NI_NAMEREQD);
    if (ni == 0)
        printf(" %s", node);
    ni = getnameinfo((struct sockaddr*)&serveraddr, serveraddrlen,12
                        addrbuf, sizeof(addrbuf), NULL, 0, NI_NUMERICHOST);
    if (ni == 0)
        printf(" (%s)", addrbuf);

    printf(": %s\n", databuf);

    close(s);
}
```

1 Declares `addrinfo` structures and the `hints` structure.

2 Declares the `sockaddr_in6` structure. The use of this type of structure is dictated by the communication domain of the socket (AF_INET6), which implies communication using the IPv6 protocol. If you wanted to write a protocol-independent program, you would declare a `sockaddr_storage` structure.

3 Declares the address string buffer, node name string buffer, service name string buffer, error number variable, and server address length variable.

4 Clears the hints structure and sets up hints variables. Thehints structure contains values that direct the getaddrinfo processing. In this case, AF_INET6 returns IPv6 addresses. The AI_ADDRCONFIG and AI_V4MAPPED values return AAAA records if an IPv6 address is configured, and if none are found, return A records if an IPv4 address is configured. See getaddrinfo(3) for more information on hints structure values.

5 Obtains the server address. A call to getaddrinfo returns IPv6–formatted addresses in one or more structures of type addrinfo.

6 Creates an AF_INET6 socket. The socket type is specified in the addrinfo structure.

7 Connects to the server using the address in the addrinfo structure named cur_info.

8 Frees all addrinfo structures.

9 Sends a request to the server.

10 Receives a response from the server.

11 Obtains the address of the peer socket at the other end of the connection and stores the address in a sockaddr_in6 structure named serverinfo.

12 Obtains the server's name with a call to getnameinfo using the address in the sockaddr_in6 structure named serveraddr. The NI_NAMEREQD flag directs the routine to return a host name for the given address.

13 Obtains the server's numeric address value with a call to getnameinfo using the address in the sockaddr_in6 structure named serveraddr. The NI_NUMERICHOST flag directs the routine to return an address value for the given address.

C.2.2 Server Program Using AF_INET6 Sockets

Example C–4 shows a sample server program that you can build, compile, and run on your system. The program receives requests from and sends responses to client programs on other systems.

Example C–4: Server Stub Routine

```
/*
 * *****************************************************************
 * *                                                               *
 * *    Copyright (c) Compaq Computer Corporation, 2000            *
 * *                                                               *
 * *   The software contained on this media  is  proprietary  to   *
 * *   and  embodies  the  confidential  technology  of  Compaq    *
 * *   Computer Corporation.  Possession, use,  duplication  or    *
 * *   dissemination of the software and media is authorized only  *
```

Example C–4: Server Stub Routine (cont.)

```
 * *   pursuant to a valid written license from Compaq Computer    *
 * *   Corporation.                                                *
 * *                                                               *
 * *   RESTRICTED RIGHTS LEGEND   Use, duplication, or disclosure  *
 * *   by the U.S. Government is subject to restrictions  as  set  *
 * *   forth in Subparagraph (c)(1)(ii)  of  DFARS  252.227-7013,  *
 * *   or  in  FAR 52.227-19, as applicable.                       *
 * *                                                               *
 * *****************************************************************
 */

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/errno.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <arpa/inet.h>

#define SERVER_PORT 7639
#define CLIENT_PORT 7739

#define MAXBUFSIZE 4096

int main (
    int argc,
    char **argv )
{
    int             s;
    char            databuf[MAXBUFSIZE];
    int             dcount;
    struct sockaddr_in6       serveraddr;[1]
    struct sockaddr_storage   clientaddr;[2]
    char                      addrbuf[INET6_ADDRSTRLEN];
    char                      node[MAXDNAME];
    char                      port[MAXDNAME];
    int                       err;
    int                       ni;
    int                       clientaddrlen;
    const char                *response = "this is the server's response";

    if ((s = socket(AF_INET6, SOCK_STREAM, 0)) < 0) {[3]
        perror("socket");
        exit(1);
    }

    bzero((char *) &serveraddr, sizeof(struct sockaddr_in6));[4]
    serveraddr.sin6_family      = AF_INET6;
    serveraddr.sin6_addr        = in6addr_any;
    serveraddr.sin6_port        = htons(SERVER_PORT);

    if (bind(s, (struct sockaddr *)&serveraddr, sizeof(serveraddr)) < 0) {[5]
        perror("bind");
        exit(2);
    }
    if (listen(s, SOMAXCONN) < 0) { [6]
        perror("listen");
        close(s);
```

Example C–4: Server Stub Routine (cont.)

```
        exit(3);
    }
    while (1) {
        int new_s;
        clientaddrlen = sizeof(clientaddr);
        bzero((char *)&clientaddr, clientaddrlen); 7
        new_s = accept(s, (struct sockaddr*)&clientaddr, &clientaddrlen); 8

        if (new_s < 0) {
            perror("accept");
            continue;
        }
        dcount = recv(new_s, databuf, sizeof(databuf), 0); 9
        if (dcount < 0) {
            perror("recv");
            close(new_s);
            continue;
        }
        databuf[dcount] = ’\0’;

        printf("Request received from");
        ni = getnameinfo((struct sockaddr *)&clientaddr, 10
            clientaddrlen, node, sizeof(node), NULL, 0, NI_NAMEREQD);
        if (ni == 0)
            printf(" %s", node);
        ni = getnameinfo((struct sockaddr *)&clientaddr, 11
            clientaddrlen, addrbuf, sizeof(addrbuf), port, sizeof(port),
            NI_NUMERICHOST|NI_NUMERICSERV);
        if (ni == 0)
            printf(" (%s) port %d", addrbuf, port);
        printf(" \"%s\"\n", port, databuf);
        if (send(new_s, response, strlen(response), 0) < 0) { 12
```

Example C–4: Server Stub Routine (cont.)

```
          perror("send");
          close(new_s);
          continue;
       }
       close(new_s);
    }
    close(s);
}
```

1 Declares the sockaddr_in6 structure named serveraddr. The use of this type of structure is dictated by the communication domain of the socket (AF_INET6), which implies communication using the IPv6 protocol.

2 Declares a sockaddr_storage structure named clientaddr. The use of this type of structure enables your program to be protocol independent.

3 Creates an AF_INET6 socket. The socket type SOCK_STREAM is specified for TCP or connection-oriented communication.

4 Clears the server address and sets up the server variables.

5 Binds the server's address to the AF_INET socket.

6 Listens on the socket for a connection. The server will queue up to SOMAXCONN pending connections while it finishes processing the previous accept call. See sys_attrs_socket(5) for more information on the socket subsystem kernel attributes.

7 Clears the client address.

8 Accepts a connection on this socket. The accept call places the client's address in the sockaddr_storage structure named clientaddr.

9 Receives data from the client.

10 Obtains the client's name with a call to getnameinfo using the address in the sockaddr_storage structure named clientaddr. The NI_NAMEREQD flag directs the routine to return a host name for the given address.

11 Obtains the client's numeric address value and port nnumber with a call to getnameinfo using the address in the sockaddr_storage structure named clientaddr. The NI_NUMERICHOST and NI_NUMERICSERV flags direct the routine to return a values for the given address and the port number.

12 Sends a response to the client.

C.3 Sample Program Output

This section contains sample output from the server and client programs. The server program makes and receives all requests on an AF_INET6 socket using `sockaddr_in6`. For requests received over IPv4, `sockaddr_in6` contains an IPv4-mapped IPv6 address.

The following example shows the client program running on node `hostb6` and sending a request to node `hosta6`. The program uses an AF_INET6 socket. The `hosta6` node has the IPv6 address 3ffe:1200::a00:2bff:fe97:7be0 in the Domain Name System (DNS).

```
user2@hostb6> ./client hosta6
Response received from hosta6.ipv6.corp.example (3ffe:1200::a00:2bff:fe97:7be0):
 this is the server's response
```

On the server node, the following example shows the server program invocation and the request received from the client node `hostb6`:

```
user1@hosta6> ./server
Request received from hostb6.ipv6.corp.example (3ffe:1200::a00:2bff:fe2d:02b2
 port 7739 "this is the client's request"
```

The following example shows the client program running on node `hostb` and sending a request to node `hosta`. The program uses an AF_INET6 socket. The `hosta` node has the IPv4 address 10.10.10.13 in the DNS.

```
user2@hostb> ./client hosta
Response received from hosta.corp.example (::ffff:10.10.10.13): this is the
 server's response
```

On the server node, the following example shows the server program invocation and the request received from the client node `hostb`:

```
user1@hosta6> ./server
Request received from hostb.corp.example (::ffff:10.10.10.251) port 7739
 "this is the client's request"
```

The following example shows the client program running on node `hostc` and sending a request to node `hosta`. The program was built and run on an IPv4-only system using an AF_INET socket.

```
user3@hostc> ./client hosta
Response received from hosta.corp.example (10.10.10.13): this is the
 server's response
```

On the server node, the following example shows the server program invocation and the request received from the client node `hostc`:

```
user1@hosta6> ./server
Request received from hostc.corp.example (::ffff:10.10.10.63) port 7739
 "this is the client's request"
```

D

TCP Specific Programming Information

This appendix contains information about performance aspects of the Transport Control Protocol (TCP).

It discusses how programs can influence TCP throughput by controlling the following via socket options:

- TCP window size
- TCP error recovery
- TCP round-trip time
- TCP reliability

D.1 TCP Throughput and Window Size

TCP throughput depends on the transfer rate, which is the rate at which the network can accept packets, and the round-trip time, which is the delay between the time a TCP segment is sent and the time an acknowledgement arrives for that segment. These factors determine the amount of data that must be buffered (the window) prior to receiving acknowledgment to obtain maximum throughput on a TCP connection.

If the transfer rate or the round-trip time or both is high, the default window size used by TCP may be insufficient to keep the pipe fully loaded. Under these circumstances, TCP throughput can be limited because the sender is required to stall until acknowledgements for prior data are received.

The receive socket buffer size determines the maximum receive window for a TCP connection. The transfer rate from a sender can also be limited by the send socket buffer size. The default value is 61440 bytes for TCP send and receive buffers.

D.1.1 Programming the TCP Socket Buffer Sizes

An application can override the default TCP send and receive socket buffer sizes by using the `setsockopt` system call and specifying the SO_SNDBUF and SO_RCVBUF options, prior to establishing the connection. The largest size that can be specified with the SO_SNDBUF and SO_RCVBUF options is limited by the kernel variable `sb_max`. See Section D.1.2.1 for information about increasing this value.

For maximum throughput, send and receive socket buffers on both ends of the connection should be of equal size.

When writing programs that use the `setsockopt` system call to change a TCP socket buffer size (SO_SNDBUF, SO_RCVBUF), note that the actual socket buffer size used for a TCP connection can be larger than the specified value. This situation occurs when the specified socket buffer size is not a multiple of the TCP Maximum Segment Size (MSS) to be used for the connection.

TCP determines the actual size, and the specified size is rounded up to the nearest multiple of the negotiated MSS. For local network connections, the MSS is generally determined by the network interface type and its maximum transmission unit (MTU).

D.1.2 TCP Window Scale Option

Tru64 UNIX implements the TCP window scale option, as defined in RFC 1323: *TCP Extensions for High Performance*. The TCP window scale option, which allows larger windows to be used, was designed to increase throughput of TCP over high bandwidth, long delay networks. This option may also increase throughput of TCP in local Gigabit Ethernet and FDDI networks.

The window field in the TCP header is 16 bits. Therefore, the largest window that can be used without the window scale option is 2**16 (64KB). When the window scale option is used between cooperating systems, windows up to (2**30)-1 bytes are allowed. The option, transmitted between TCP peers at the time a connection is established, defines a scale factor which is applied to the window size value in each TCP header to obtain the actual window size.

The maximum receive window, and therefore the scale factor offered by TCP during connection establishment, is determined by the maximum receive socket buffer space.

If the receive socket buffer size is greater than 65535 bytes, during connection establishment, TCP will specify the Window Scale option with a scale factor based on the size of the receive socket buffer. Both systems involved in the TCP connection must send the Window Scale option in their SYN segments for window scaling to occur in either direction on the connection. As stated previously, for maximum throughput, send and receive buffers on both ends of the connection should be of equal size.

D.1.2.1 Increasing the System Socket Buffer Size Limit

The `sb_max` kernel attribute for the Socket kernel subsystem limits the amount of socket buffer space that can be allocated for each send and receive buffer. The current default is 1048576 bytes (1MB) but optionally you can increase it.

For local Gigabit Ethernet connections, the current value is sufficient. For long delay, high bandwidth paths, values greater than 1MB may be required.

To change the `sb_max` kernel attribute in the kernel currently in memory, use either the `dxkerneltuner` utility or the `sysconfig -r` command. See `dxkerneltuner`(8) or `sysconfig`(8), respectively, for more information.

D.2 TCP Performance and Error Recovery

TCP relies on acknowledgements to determine if packets arrive at their destination. In high-speed connections (for example, Gigabit Ethernet) that use large windows, the default mechanism can seriously affect throughput.

By default, if a packet is lost, TCP retransmits that packet and all packets after it. An application can override the default by using the `setsockopt` system call specifying the TCP_SACKENA option , prior to establishing the connection. After the option is agreed upon, the data receiver can inform the sender about all segments that have arrived successfully. In this way, the sender need retransmit only those segments that have actually been lost. This option is useful in cases where multiple segments are dropped.

D.3 TCP Performance and Round-Trip Measurement

TCP bases its round-trip time measurements on a only one packet per window. In high-speed connections (for example, Gigabit Ethernet) that use large window, it is possible for the round-trip time estimates to be seriously flawed, resulting in many retransmissions.

By default, TCP does not send time stamps in the TCP header. An application can override the default by using the `setsockopt` system call specifying the TCP_TSOPTENA option, prior to establishing the connection. After the option is selected, the sender places a timestamp in each data segment. The receiver, if configured to accept them, sends these timestamps back in ACK segments. This provides the sender with a reliable mechanism with which to measure round-trip time.

D.4 TCP Reliability and Sequence Numbers

TCP relies on sequence numbers to determine the correct sequencing of packets and to determine if duplicate packets have been received. In high-speed connections (for example, Gigabit Ethernet), it is possible for the sequence numbers to wrap around. This means that two packets could have the same sequence number yet contain different information; they are not duplicate but TCP will assume that they are.

By default, TCP does not provide a mechanism for rejecting old duplicate packets. An application can override the default by using the `setsockopt` system call specifying the TCP_PAWS option, after specifying the TCP_TSOPTENA option, and prior to establishing the connection. When the PAWS (Protect Against Wrapped Sequence numbers) option is enabled, the receiver rejects any old duplicate segments that are received. This option is used on synchronized TCP connections only.

E

Information for Token Ring Driver Developers

This appendix contains the following information for developers of Token Ring drivers for the Tru64 UNIX operating system:

- Enabling source routing
- Using canonical addresses
- Avoiding unaligned access
- Setting fields in the `softc` structure of the driver

E.1 Enabling Source Routing

Source routing is a bridging mechanism that systems on a Token Ring local area network (LAN) use to send messages to a system on another interconnected Token Ring LAN. Under this mechanism, the system that is the source of a message uses a route discover process to determine the optimum route over Token Ring LANs and bridges to a destination system.

To use the Token Ring source routing module you must add the TRSRCF option to your kernel configuration file. Use the `doconfig –c` command to add the TRSRCF option, as follows:

1. Enter the `doconfig –c` *HOSTNAME* command from the superuser prompt (#). *HOSTNAME* is the name of your system in uppercase letters; for example, for a system called `host1` you would enter:

   ```
   # doconfig –c HOST1
   ```

2. Add TRSRCF to the options section of the kernel configuration file.

   Enter `y` when the system asks whether you want to edit the kernel configuration file. The `doconfig` command allows you to edit the configuration file with the `ed` editor. For information about using the `ed` editor, see `ed`(1).

   The following `ed` editing session shows how to add the TRSRCF option to the kernel configuration file for `host1`. The number of the line after which you append the new line can differ between kernel configuration files:

```
*** KERNEL CONFIGURATION AND BUILD PROCEDURE ***

Saving /sys/conf/HOST1 as /sys/conf/HOST1.bck

Do you want to edit the configuration file? (y/n) [n]: y

Using ed to edit the configuration file.  Press return when ready,
or type 'quit' to skip the editing session:
2153

48a
options         TRSRCF
.
1,$w
2185
q


*** PERFORMING KERNEL BUILD ***
```

3. After the new kernel is built, you must move it from the directory where `doconfig` places it to the root directory (\/) and reboot your system.

For detailed information on reconfiguring your kernel or the `doconfig` command see the *System Administration* manual.

The Token Ring source routing functionality is initialized if the `trn_units` variable is greater than or equal to 1. The `trn_units` variable indicates the number of Token Ring adapters initialized on the system.

The driver should declare `trn_units` as follows:

```
extern int trn_units;
```

At the end of its `attach` routine, the driver should increment the `trn_units` variable as follows:

```
trn_units++;
```

For information on source routing management see the *Network Administration: Connections* manual.

E.2 Using Canonical Addresses

The Token Ring driver requires that the destination address (DA) and source address (SA) in the Media Access Control (MAC) header be in the canonical form while presenting it to the layers above the driver.

The canonical form is also known as the Least Significant Bit (LSB) format. It differs from the noncanonical form, known as the Most Significant Bit (MSB) format, in that it transmits the LSB first. The noncanonical form transmits the MSB first. The two formats also differ in that the bit order within each octet is reversed.

For example, the following address is in noncanonical form:

```
10:00:d4:f0:22:c4
```

The same address in canonical form is as follows:

```
08-00-2b-0f-44-23
```

If the hardware does not present the driver with a canonical address in the MAC header, you should convert the address to canonical form before passing it up to the higher layers. The `haddr_convert` kernel routine is available for converting canonical addresses to noncanonical, and vice versa. It has the following format:

```
haddr_convert( addr)
unsigned char *addr
```

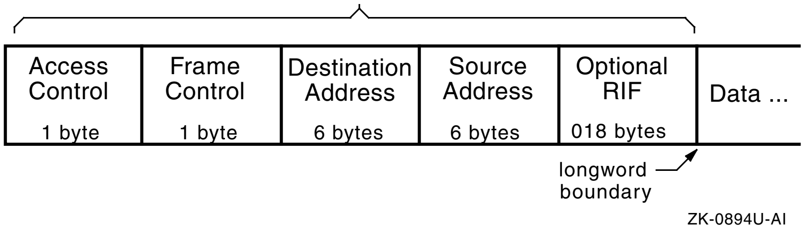
The *addr* variable is a pointer to the 6 bytes of the address that require conversion from either noncanonical to canonical or canonical to noncanonical form. The converted address is returned in the same buffer.

E.3 Avoiding Unaligned Access

The frame that the driver receives consists of the Media Access Control (MAC) header, which includes the Routing Information Field (RIF) and data. Because the length of the RIF can vary between 0 and 18 bytes, the data after the RIF may not be aligned to a longword boundary. To avoid degraded performance, you should pad the RIF field so that data always starts on a longword boundary.

Figure E–1 illustrates the relationship between the components of the MAC header and the data in a typical frame.

Figure E–1: Typical Frame



E.4 Setting Fields in the softc Structure of the Driver

The `softc` structure contains driver-specific information.

You must set the following field of the softc structure in the attach routine of the driver:

```
sc->isac.ac_arphrd=ARPHRD_802;
```

Here, sc is a pointer to the softc structure, and ARPHRD_802 is the value of the hardware type used in an Address Resolution Protocol (ARP) packet sent from this interface. A value of 6 for ARPHRD_802 indicates an IEEE 802 network.

F
The Data Link Interface

The data link interface (DLI) is a programming interface that allows programs to use the data link facility directly to communicate with data link programs running on a remote system.

See Section F.5 for client and server DLI programming examples.

F.1 Prerequisites for DLI Programming

DLI programming requires both a thorough knowledge of the C programming language and experience writing system programs. If you intend to use the Ethernet substructure, you should be familiar with the Ethernet protocol. If you intend to use the 802 substructure, you should be familiar with the 802.2, 802.3, and FDDI protocols.

You should be also be familiar with the following concepts before attempting to write programs to the DLI interface:

- Datagram sockets

  Your application uses sockets to send and receive Ethernet, 802.3 and FDDI frames. DLI uses datagram sockets only.

  For more information about using sockets, see Chapter 4.

- Logical Link Control (LLC)

  LLC is a sublayer of DLI that provides a set of services determined by a value in the 802.2 frame format.

- Physical and multicast addressing

  You can send and receive messages over the network using physical or multicast addresses. You can use physical addresses to send messages to a single destination system. Multicast addresses are not associated with any specific system; instead, a packet sent to a multicast address is received by all systems with the multicast address enabled.

  For more information about multicast addressing, see Section 4.7.

- Standard frame formats

  The Ethernet frame format is a proprietary standard that belongs to Compaq Computer Corporation, Intel Corporation, and Xerox Corporation. The IEEE 802.3 frame format is a standard for multivendor networking. The FDDI and IEEE 802.3 frame formats are very similar.

Both contain the LLC (or 802.2) frame within them. See Section F.3.1 for more information.

Note that running DLI applications on this operating system requires superuser or root privileges.

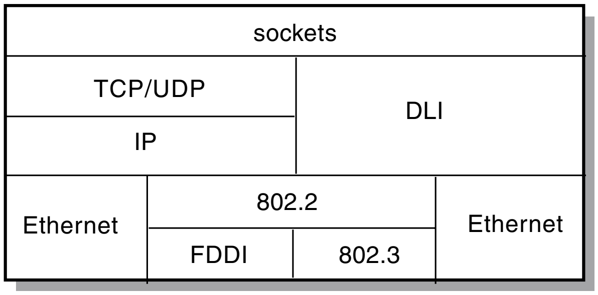
F.2 DLI Overview

DLI programs transfer data over networks using the standard Ethernet frame format, the Open Systems Interconnect (OSI) 802.3 frame format, or the FDDI frame format. Your operating system can run Internet, DECnet, and DLI programs concurrently.

The operating system supports both Ethernet and 802.2 data link services. DLI and IP both run over Ethernet and 802.2. FDDI and 802.3 use the 802.2 Logical Link Control (LLC) as their data link sublayer. TCP and UDP run over IP, providing data delivery and message routing services to the programs that use them. Because DLI provides direct access to the data link layer it does not provide the higher-level services that TCP and UDP do.

Figure F–1 illustrates in greater detail the relationships between DLI and IP, DLI and Ethernet, and DLI and 802.2.

Figure F–1: DLI and the Network Programming Environment



Sockets are the user application interface and facilitate access to TCP, UDP, and DLI. See Chapter 4 for information about opening sockets in the DLI communication domain (AF_DLI).

F.2.1 DLI Services

DLI provides the following services at the data link layer:

- Datagram service
- Logical Link Control (LLC) layer
    - ISO 802.2 Class I, Type I service

- Multicast address mode
- Medium Access Control (MAC) layer
  - Ethernet frames
  - 802.3 frames
  - FDDI frames

F.2.2 Hardware Support

DLI requires no knowledge of the underlying hardware. It uses Ethernet or FDDI device drivers, which each use the `probe` routine to determine what devices a particular system has configured. For a complete list of the supported network devices, see the Tru64 UNIX Software Product Description.

To determine which network devices are configured on your system, use the `/usr/sbin/netstat –i` command, as follows:

```
% /usr/sbin/netstat –i

Name  Mtu   Network     Address      Ipkts Ierrs    Opkts Oerrs  Coll
ln0   1500  <Link>                     746     0      234     0    18
ln0   1500  orange-net  host1          746     0      234     0    18
sl0*  296   <Link>                       0     0        0     0     0
sl1*  296   <Link>                       0     0        0     0     0
lo0   1536  <Link>                      74     0       74     0     0
lo0   1536  loop        localhost       74     0       74     0     0
```

The output displayed on your screen contains information about the interfaces or devices that your system has configured. In this example, an Ethernet hardware device (`ln`) is configured, as are two Serial Line Interface Protocol devices (`sl0` and `sl1`). The asterisk (*) following the `sl0` and `sl1` indicates that the support for the interfaces has not been turned on yet.

F.2.3 Using DLI to Access the Local Area Network

A data link on a single local area network (LAN) controller supports multiple concurrent users. Each station represents an available port on the network channel.

Because multiple users simultaneously access the network channel, your program must use addressing mechanisms that ensure delivery of messages to the correct recipient. Any message you transmit on the network must include an Ethernet or FDDI address that identifies the destination system. The message must also include an additional identifier that directs the message to the correct user on the destination system; this identifier varies according to the frame format you choose to use. DLI builds frames according to the Ethernet, IEEE 802.3, or FDDI standards.

F.2.4 Including Higher-Level Services

DLI provides only datagram services. Because DLI is a direct interface to the data link layer, it does not offer higher-level services normally provided by Internet and DECnet. Therefore, your application should provide the following kinds of services:

- Packet routing and guaranteed delivery
- Flow control
- Error recovery
- Data segmentation

F.3 DLI Socket Address Data Structure

This section describes the Ethernet, 802.3, and FDDI standard frame formats, and the function of the DLI socket address data structure (`sockaddr_dl`). It explains how you use `sockaddr_dl` to specify the domain address, the network device, and the Ethernet, 802.3, or FDDI substructure.

F.3.1 Standard Frame Formats

The following diagrams illustrate the differences and similarities between the Ethernet, 802.3, and FDDI frames.

Figure F–2 illustrates the Ethernet frame format.

Figure F–2: The Ethernet Frame Format

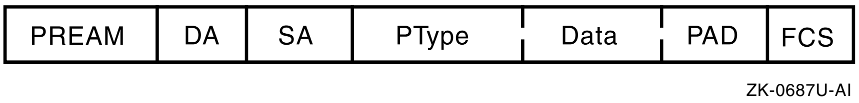


Figure F–3 illustrates the 802.3 frame format. Note that the 802.3 frame format contains the 802.2 structure, which is illustrated in Figure F–5.

Figure F–3: The 802.3 Frame Format

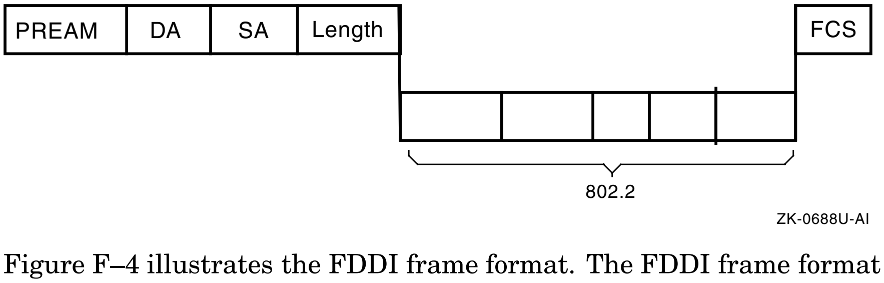


Figure F–4 illustrates the FDDI frame format. The FDDI frame format also contains within it the 802.2 structure illustrated in Figure F–5.

Figure F–4: The FDDI Frame Format

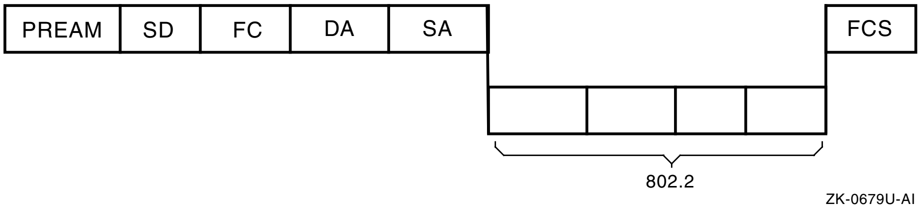
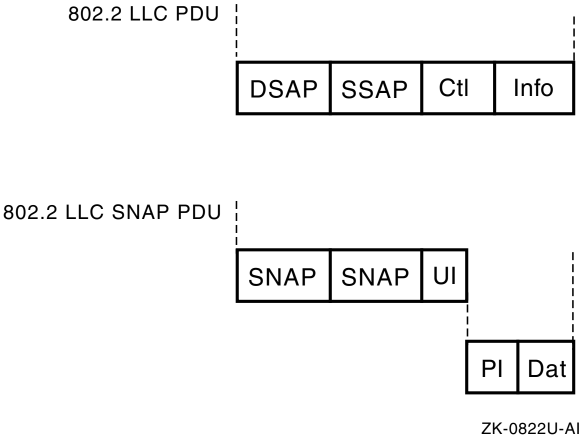


Figure F–5 illustrates the 802.2 LLC PDU and the 802.2 LLC SNAP PDU. One of these two structures is contained within the 802.3 and FDDI frame formats.

Figure F–5: The 802.2 Structures



Typically, 802 applications use the 802.2 LLC PDU format; however, an application developer may choose to use the 802.2 LLC SNAP PDU format for the following reasons:

- Using the SNAP_SAP is a convenient way to map Ethernet protocol types on to 802.2 protocols. This is useful for applications that operate over both Ethernet and 802.2, or are migrating from Ethernet to 802.2.
- The I/O control flags (DLI_NORMAL, DLI_EXCLUSIVE and DLI_DEFAULT) are valid only for Ethernet and 802.2 SNAP frames. These flags are meaningless when the non-SNAP 802.2 LLC PDU is used.
- Using the SNAP_SAP allows a greater number of applications to run over 802.2 because the SNAP SAP has a five byte protocol ID associated with it. The normal 802.2 LLC PDU, on the other hand, is multiplexed on the 7 most significant bits of the DSAP.

F.3.2 How the sockaddr_dl Structure Works

DLI provides a socket address data structure through which you can configure the set of services required for communication at the data link layer. The data structure `sockaddr_dl` is used to convey information to DLI when an application binds to the network, or when it transmits a packet to the network. DLI also uses it to convey information to the application when it receives a packet from the network. This includes network device information, the packet format to be used, and addressing information.

The following example shows the DLI socket address structure, which is defined in the header file `<dli/dli_var.h>`:

```
#define DLI_ETHERNET    0
#define DLI_802         2
.
.
.

struct sockaddr_dl {
    u_char dli_len;                 /* length of sockaddr */
    u_char dli_family;              /* address family (AF_DLI) */
    struct dli_devid dli_device;    /* id of comm device to use */
    u_char dli_substructype;        /* id to interpret following */
                                    /* structure */
    union {
        struct sockaddr_edl dli_eaddr;       /* Ethernet */
        struct sockaddr_802 dli_802addr;     /* OSI 802 support */
        caddr_t dli_aligner1;                /* this needs to have */
                                             /* longword alignment */
    } choose_addr;
};
```

Any single application can send and receive both Ethernet and 802 substructures. The Ethernet substructure enables applications to communicate across an Ethernet. The 802 substructure enables applications to use 802.2, 802.3, and FDDI protocols to communicate with each other.

You can use system calls to specify values within the socket address structure by using either the Ethernet or 802 substructures.

The fields within the substructures are updated as a function of the system call. For example, the `bind` system call is used to specify the domain, network device, and most of the substructure. When using the `sendto` system call to transmit data, the domain, network device, and part of the substructure must be specified. When using the `recvfrom` system call to receive data, DLI fills in the entire `sockaddr` structure.

The `dli_econn` and `dli_802_3_conn` user-written subroutines open a socket and bind the associated domain, network device name, protocol type, and other substructure information to the socket. See Section F.5 for examples of the `dli_econn` and `dli_802_3_conn` user-written subroutines.

The following sections describe the functions that the Ethernet and 802.2 substructures provide within the DLI `sockaddr_dl` data structure.

F.3.3 Ethernet Substructure

The following example shows the DLI Ethernet socket address substructure:

```
#define DLI_EADDRSIZE   6
.
.
.

struct sockaddr_edl {
    u_char dli_ioctlflg;                /* i/o control flags */
    u_char dli_options;                 /* Ethernet options */
    u_short dli_protype;                /* Ethernet protocol type */
    u_char dli_target[DLI_EADDRSIZE];   /* Ethernet address of */
                                        /* destination system */
    u_char dli_dest[DLI_EADDRSIZE];     /* Ethernet address used to */
                                        /* address the local system; */
};                                      /* DLI places the destination */
                                        /* address of an incoming */
                                        /* packet here to be used in */
                                        /* the recvfrom call. This */
                                        /* address can be the sys- */
                                        /* tem's address or a multi */
                                        /* cast address. */
```

The Ethernet substructure specifies the following:

- An I/O control flag for the protocol type (`dli_ioctlflg`)
- Whether Ethernet is padded (`dli_options`)

  The PAD is a 2-byte length field in little-endian after the MAC/LLC header. The following entry in the `<dli/dli_var.h>` header file is the bit that must be set in the `dli_options` field to turn on padding:

  ```
  #define DLI_ETHERPAD     0x01    /* Protocol is padded */
  ```
- The DLI protocol type (`dli_prototype`)
- The Ethernet address of the destination system (`dli_target`)
- The Ethernet address used to address the local system (`dli_dest`)

This information is used to create the Ethernet frame format.

F.3.3.1 How Ethernet Frames Work

All Ethernet frames contain a 16-bit identification number called an Ethernet protocol type (PType). When a message arrives at the controller, the protocol type is used to identify which port receives the frame. DLI applications that communicate across the Ethernet must always enable the same Ethernet protocol type. In addition to using protocol types to select a user for an incoming packet, you can configure DLI to select a user as a function of both the protocol type and the physical address of the remote system. This allows several applications in the same system to use the same type, which can make input/output simpler for the application.

F.3.3.2 Defining Ethernet Substructure Values

The user specifies the values for the following fields in the Ethernet socket address substructure. The other fields are filled in either by system calls or DLI:

- Destination address (dli_target[DLI_EADDRSIZE])

  You can use the dli_target field to specify the destination address.
- Protocol type (dli_protype)

  You can use the dli_prototype field to specify the protocol to be used for data transmission.
- I/O Control Flag (dli_ioctlflg)

The following sections define the values for user-definable members in the Ethernet substructure.

Destination Node Physical Address

The destination system physical address (DA in Figure F–2) is a 48-bit unique value assigned by the manufacturer to a station on the Ethernet. For example, 08-00-2b-XX-XX-XX is the form a valid Ethernet address takes, with the Xs being replaced by hexadecimal digits. DA is the address of the remote system with respect to the local system.

If you do not specify the DA value with the bind call, you must specify it when sending data by using the sendto call. In addition, you should use the recvfrom call to determine the source of a data message. You can use either the physical address or a multicast address to send messages in the sendto system call.

Protocol Type

The protocol type (PType in Figure F–2) is a 16-bit value in the Ethernet frame following the source address. The Ethernet driver passes the protocol type to DLI for use in determining the recipient of the data in the frame. With the exception of reserved values, you can use any Ethernet protocol type if it is assigned to you by the manufacturer and not used elsewhere in your system.

The following hexadecimal values are reserved for use by the system:

- 0X 0200 — PUP Protocol
- 0X 0800 — Internet Protocol
- 0X 0806 — Address Resolution Protocol
- 0X 6004 — Local Area Transport
- 0X 6003 — Phase IV DECnet

- 0X 6002 — MOP CCR Protocol
- 0X 6001 — MOP Downline Load Protocol
- 0X 9000 — MOP Loopback Protocol
- 0X 1000 to 0X 100f — Internet Trailer Protocol (used by VAX only)

I/O Control Flag

The I/O control flag, defined in the header file `<dli/dli_var.h>`, is a value that DLI uses to determine how your program reserves a protocol type. It is used by DLI to determine whether to select a user as a function of the protocol type alone or as a function of the combination of the protocol type and the target audience. The following list defines the possible I/O control flags and describes the conditions for their use:

- `NORMAL`

  Allows your program to exchange messages with one destination system, using only the specified protocol type. When using the `NORMAL` flag, you must specify the destination system physical address in the `bind` call and you can use any of the data transfer calls to send and receive data. DLI forwards to the user all messages containing the specified protocol type from the specified target.

- `EXCLUSIVE`

  Gives your program exclusive use of the specified protocol type and allows the program to exchange data with any other system using this protocol type. In other words, the program receives all messages with the specified protocol type. When you use the `EXCLUSIVE` flag, do not specify the target address with the `bind` call. You must use the `sendto` and `recvfrom` calls to exchange data with other systems, and you must specify the target address with the `sendto` call. In the address structure (returned with `recvfrom`), DLI fills in the target address with the source address in the Ethernet frame. It also fills in the destination address with the destination address in the Ethernet frame.

- `DEFAULT`

  Allows your program to receive messages that contain the specified protocol type and that are meant for no other program on the system. If no other program is bound exclusively to the protocol type or the protocol type/address pair in the message, the socket bound to the protocol type gets the message by default. This mode of operation is recommended for use in programs that listen for messages but do not necessarily send them. When you use the `DEFAULT` flag, do not specify the target address with the `bind` call. Use the `recvfrom` call to receive data from other systems. If you are using the `DEFAULT` flag, DLI fills in the target address with the source address in the Ethernet frame. It also fills in the destination address with the destination address in the Ethernet frame.

F.3.4 802.2 Substructure

The 802.2 substructure enables applications to communicate with each other using the 802.2, 802.3, and FDDI protocols. It uses two basic modes of operation: Class I, Type 1 service, and the services supplied by your application using the 802.2 protocol.

The following example shows the DLI 802.3 socket address substructure:

```
struct sockaddr_802 {               /* 802.3 sockaddr struct */
    u_char ioctl;                   /* filter on incoming packets */
                                    /* addressed to the SNAP SAP */
    u_char svc;                     /* service class for this portal */
    struct osi_802hdr eh_802;       /* OSI 802 header format */
};
```

The 802.2 substructure subsumes both the 802.3 and FDDI frame formats. You can specify values for the following fields:

- Destination system physical address (DA in Figure F–3 and Figure F–4)
- Service class
- Destination service access points (DSAP in Figure F–5)
  - Individual
  - Group
- Source service access point (SSAP in Figure F–5)

  The protocol identifier and I/O control field may be required, depending on the type of SSAP you enable.
- Control field

F.3.4.1 Defining 802 Substructure Values

The following sections define the possible values for all members in the 802 substructure.

Destination Node Physical Address

The destination system physical address (DA) is a 48-bit unique value assigned by the manufacturer to a station on an Ethernet or FDDI network. For example, 08-00-2b-XX-XX-XX is a valid Ethernet or FDDI address, with the Xs being replaced by hexadecimal digits. This is the address of the remote system with which the application attempts to exchange packets. It must be specified in the bind call, except when the I/O control field is either EXCLUSIVE or DEFAULT and the service access point (SAP) is a SNAP_SAP type. The SAP must be specified in the sendto call.

Service Class

The service class is a value in the 802.2 substructure that determines the capabilities and features provided by the Logical Link Control (LLC) sublayer of the data link layer. The possible service classes are:

- `TYPE1`

  This value causes DLI to interpret all header information and provide Class I, Type 1 service.

  > **Note**
  >
  > When Type 1 service is used, the DLI software handles the `XID` and `TEST` packets. This is transparent to the application.

  DLI uses the source and destination service access points to determine who should receive the message; it interprets the control field on behalf of the user. Whether DLI passes the data field to the user depends on the value of the control field.

- `USER`

  This value provides few services. The user must, therefore, implement most of the 802.2 protocol. In other words, the application must handle the `XID` and `TEST` packets. DLI uses the source and destination service access points, but it passes the control field with the data to the user. The user must interpret the control field. This mode must be selected if the application needs to implement Class II, Type 2 service.

Destination Service Access Point

The destination service access point (DSAP) is a field in the 802.2 frame that identifies the application for which the message is intended. You can use individual or group DSAPs to identify one user or a group of users. You can use group DSAPs only when the service class is set to `USER`. The possible values for this field are:

- Individual DSAPs

  `NULL_SAP` — A DSAP consisting of all zeros. You can send `TEST` and `XID` commands and responses, but no data, to a `NULL_SAP`. (`TEST` and `XID` are explained later in this section.) The data link layer uses the `NULL_SAP` to talk to another data link layer, primarily for testing.

  User-defined DSAP — Identifies one user for whom the message is intended. The user-defined individual DSAP must be an even number greater than or equal to 2 and less than or equal to 254.

  `SNAP_SAP` — The 802.3 Subnetwork Access Protocol.

- Group DSAP (user defined)

Identifies more than one user for whom the message is intended. You can send data to a maximum of 127 group DSAPs on one socket. The user defined group DSAP must be an odd number greater than or equal to 3 and less than or equal to 255. Note that the 255 number is the global SAP and must be enabled like any other group SAP. You can use group SAPs only when the service class is set to `USER`.

Source Service Access Point

The source service access point (SSAP) is a field in the 802.2 frame that identifies the address of the application that sent the message. You can enable only one SSAP on a socket. The SSAP must be an even number greater than or equal to 2 and less than or equal to 254.

Note

When using the `SNAP_SAP`, both the DSAP and SSAP must be set to `SNAP_SAP`. In addition, you must specify the protocol identifier and control field. The protocol identifier is five bytes. The control field is one byte. Enabling the `SNAP_SAP` is allowed only when the service class is `TYPE1`.

Note also that IEEE 802.2 standard reserves for its own definition all SAP addresses with the second least significant bit set to 1. It is suggested that you use these SAP values for their intended purposes, as defined in the IEEE 802.2 standard.

Control Field

The control field specifies the packet type. The following values are defined for Class I, Type 1 service, and can also be used in the user-supplied mode to provide Class II, Type 2 service.

Note

An application using this user mode is responsible for providing the correct services. For other operations supported by CLASS II service, see the *IEEE Standards for Local Area Networks: Logical Link Control*, published by the Institute of Electrical and Electronics Engineers, Inc.

- Exchange Identification

  The value `XID` identifies the exchange identification command or response. An 8-bit format identifier and a 16-bit parameter follow the `XID` control field. The 16-bit parameter identifies the supported LLC services and the receive window size. The LLC is the top sublayer in the data link layer of the IEEE/Std 802 Local Area Network Protocol. The following values of XID are defined in the DLI header file <dli/dli_var.h>:

- XID_PCMD

  Exchange identification command with the poll bit set. The exchange identification command conveys the types of LLC services supported and the receive window size to the destination LLC. This command causes the destination LLC to reply with the XID response Protocol Data Unit (PDU) at the earliest opportunity. The poll bit is set to 1, soliciting a response PDU.

- XID_NPCMD

  Exchange identification command with no poll bit set. This command is identical to the previous command, except that you clear the poll bit. No response is expected.

- XID_PRSP

  Exchange identification response with the poll bit set. The Data Link layer uses the exchange identification response to reply to an XID command at the earliest opportunity. The XID response PDU identifies the responding LLC and includes an information field like that defined for the XID command PDU, regardless of what information is present in the information field of the received XID command PDU. The final bit is set to 1, indicating that this response is sent by the LLC as a reply to a soliciting command PDU.

- XID_NPRSP

  Exchange identification response with no poll bit set. This response is identical to the previous one, except that the final bit is cleared.

- LLC Protocol Data Unit Test

  The value TEST identifies the LLC PDU command or response test. The TEST control field can be followed by a data field. The following values of TEST are defined in the DLI header file <dli/dli_var.h>:

  - TEST_PCMD

    TEST command with the poll bit set. The TEST command tests the LLC-to-LLC transmission path by causing the destination LLC to respond with the TEST response PDU at the earliest opportunity. An information field is optional with this control field value. If used, the receiving LLC returns the information rather than passing it to the user. The poll bit is set to 1, soliciting a response PDU.

  - TEST_NPCMD

    TEST command with no poll bit set. This command is identical to the previous command, except that the poll bit is cleared.

  - TEST_PRSP

    TEST response with the poll bit set. The TEST response PDU is a reply to the TEST command PDU. An information field, if present in the TEST command PDU, is returned in the corresponding TEST response PDU. The final bit is set to 1, indicating that this response is sent by the LLC as a reply to a soliciting command PDU.

  - TEST_NPRSP

    TEST response with no poll bit set. This response is identical to the previous one, except that the final bit is cleared.

- Unnumbered Information Command

  The unnumbered information command with no poll set (UI_NPCMD) sends information to one or more LLCs. The UI_NPCMD command does not have an LLC response PDU. This is usually passed up to the application. Class I, Type 1 applications generally send and receive data using this command.

F.4 Writing DLI Programs

This section explains how to use system calls to write DLI programs and describes procedures for specifying values within the Ethernet and 802 substructures.

Section F.5 contains DLI programming examples of the procedures described in this section.

For additional information about how to use sockets and system calls to write application programs, see Chapter 4.

F.4.1 Supplying Data Link Services

Because DLI provides only a datagram service, a DLI application should provide the services that the higher levels of network software normally provide:

- Flow control — DLI programs running on different systems must synchronize data transfer or they will lose data.
- Error recovery — DLI reports errors, but your application must recover from them.
- Data segmentation — Your application must segment data during transmission. (See Section F.4.7 for information about the buffer size for Ethernet, 802.3, and FDDI packets.)

F.4.2 Using Tru64 UNIX System Calls

Your DLI program uses the socket interface with input arguments, structures, and substructures specific to DLI. For example, when issuing the `socket` system call, your program uses the address format `AF_DLI` and the protocol `DLPROTO_DLI`.

The beginning of any DLI program must include the header file `<dli/dli_var.h>`. Then it should follow the calling sequence shown in Table F–1.

Table F–1: Calling Sequence for DLI Programs

Function	System Call
Create a socket.	`socket`
Bind the socket to a device by specifying the address family, the frame format type, and the device over which the program will send the data using the `sockaddr_dl` structure.	`bind`
Set socket options. This call is optional.	`setsockopt`
Transfer data.	`send`, `recv`, `read`, `write`, `sendto`, and `recvfrom`
Deactivate the socket descriptor.	`close`

See Chapter 4 and the reference page for each system call for more information.

The following sections describe DLI functions, input arguments, and structures.

F.4.3 Creating a Socket

Your DLI application must create a socket by using the `socket` system call with the following input arguments:

Address family:	`AF_DLI`
Socket type:	`SOCK_DGRAM`
Protocol:	`DLPROTO_DLI`

The value `AF_DLI` specifies the DLI address family. `SOCK_DGRAM` creates a datagram socket, which is the only type of socket that DLI allows. DLI does not supply the services necessary for connecting to other programs and for using other socket types. The value DLPROTO_DLI specifies the DLI protocol module.

The following example shows how the socket call is used to open a socket to DLI:

```
int so;
 .
 .
 .
if ( (so = socket(AF_DLI,SOCK_DGRAM,DLPROTO_DLI))<0)
  {
      perror("cannot open DLI socket");
      return (-1);
  }
```

F.4.4 Setting Socket Options

Use the setsockopt call to set the following socket options within the sockaddr_dl structure:

Option	Description
DLI_ENAGSAP	Enables a group service access point (GSAP)
DLI_DISGSAP	Disables a group service access point (GSAP)
DLI_SET802CTL	Sets the 802 control field
DLI_MULTICAST	Enables the reception of all messages addressed to a multicast address

The following code examples show how to use the setsockopt call to set the socket options.

The following example shows how the setsockopt call is used to enable the GSAP option:

```
/* enable GSAPs supplied by user */
 j = 3;
 i = 0;
 while (j < argc ) {
     sscanf(argv[j++], "%x", &k);
     out_opt[i++] = k;
 }
 optlen = i;
 if (setsockopt(sock,DLPROTO_DLI,DLI_ENAGSAP,&out_opt[0],optlen) < 0){
     perror("dli_setsockopt: Can’t enable gsap");
     exit(1);
 }
```

The following example shows how the setsockopt call is used to disable the GSAP option:

```
/* disable all but the last 4 or all GSAPs, */
/* whichever is smallest */
if ( optlen > 4 )
    optlen -= 4;
if (setsockopt(sock,DLPROTO_DLI,DLI_DISGSAP,&out_opt[0],optlen) < 0){
    perror("dli_setsockopt: Can't disable gsap");
}
```

The following example shows how the `setsockopt` call is used to set the 802 control field:

```
/* set 802 control field */
out_opt[0] = TEST_PCMD;
optlen = 1;
if (setsockopt(sock,DLPROTO_DLI,DLI_SET802CTL,
            &out_opt[0],optlen)<0){
    perror("dli_setsockopt: Can't set 802 control");
    exit(1);
}
```

The following example shows how the `setsockopt` call is used to enable two multicast addresses:

```
/* enable two multicast addresses */
bcopy(mcast0, out_opt, sizeof(mcast0));
bcopy(mcast1, out_opt+sizeof(mcast0), sizeof(mcast1));
 if ( setsockopt(sock, DLPROTO_DLI, DLI_MULTICAST, &out_opt[0],
               (sizeof(mcast0) + sizeof(mcast1))) < 0 ) {
    perror("dli_setsockopt: can't enable multicast");
}
```

See Section F.5 for more detailed code examples.

F.4.5 Binding the Socket

After you create the socket, your application must bind the socket to a network device. At this point, you specify the type of format for the message. You assign a name to the socket, where the variable *name* is a pointer to a structure of the type `sockaddr_dl`. Then, you must fill in the `sockaddr_dl` data structure and include the appropriate substructure (Ethernet or 802).

To bind the socket, use the `bind` system call. See `bind`(2) for more information.

F.4.6 Filling in the sockaddr_dl Structure

Fill in the `sockaddr_dl` structure with the following information:

- Address family
- I/O device ID
- Substructure type

F.4.6.1 Specifying the Address Family

To specify the address family, use the value AF_DLI in the socket call.

F.4.6.2 Specifying the I/O Device ID

The I/O device is the controller over which your program sends and receives data to and from the target system. The I/O device ID consists of the device name, *dli_devname*, and the device number, *dli_devnumber*. Definitions for each variable follow:

- *dli_devname*

  The netstat –i command lists the devices that are available on your system.

- *dli_devnumber*

  The device number is set up in the system configuration file.

F.4.6.3 Specifying the Substructure Type

The substructure specifies the type of frame format that the program will use. Definitions for each variable follow:

- *dli_eaddr*

  Ethernet frame format (DLI_ETHERNET)

- *dli_802addr*

  802.3 frame format (DLI_802)

A program can send and receive Ethernet, 802.3, and FDDI frames, as long as it has a socket associated with each type. For example, your DLI program might communicate with one system using the Ethernet frames and another system using 802.3 or FDDI frames. Your choice of frame formats depends on the frame types used by the target program; however, only one type of frame per socket is allowed.

Your program specifies the packet header for sending your message by filling in the substructure of your choice. Example F–1 shows how to fill the sockaddr_dl structure for the Ethernet protocol. Example F–2 shows how to fill the sockaddr_dl structure for the 802 protocol:

Example F–1: Filling the sockaddr_dl Structure for Ethernet

```
/*
 * Fill out the sockaddr_dl structure for the bind call
 */
bzero(&out_bind, sizeof(out_bind));
out_bind.dli_family = AF_DLI;
out_bind.dli_substructype = DLI_ETHERNET;
bcopy(devname, out_bind.dli_device.dli_devname, i);
out_bind.dli_device.dli_devnumber = devunit;
```

Example F–1: Filling the sockaddr_dl Structure for Ethernet (cont.)

```
 out_bind.choose_addr.dli_eaddr.dli_ioctlflg = ioctl;
 out_bind.choose_addr.dli_eaddr.dli_protype = ptype;
 if ( taddr )
      bcopy(taddr, out_bind.choose_addr.dli_eaddr.dli_target,
            DLI_EADDRSIZE);

  if ( bind(sock, &out_bind, sizeof(out_bind)) < 0 )
 {
      perror("dli_eth, can't bind DLI socket");
      return(-1);
 }

  return(sock);
}
```

Example F–2: Filling the sockaddr_dl Structure for 802.2

```
 /*
  * Fill out sockaddr_dl structure for the bind call.
  * Note that we need to determine whether the
  * control field is 8 bits (unnumbered format) or
  * 16 bits (informational/supervisory format).  We do this
  * by checking the low order 2 bits, which are both 1 only
  * for unnumbered control fields.
  */
 bzero(&out_bind, sizeof(out_bind));
 out_bind.dli_family = AF_DLI;
 out_bind.dli_substructype = DLI_802;
 bcopy(devname, out_bind.dli_device.dli_devname, i);
 out_bind.dli_device.dli_devnumber = devunit;
 out_bind.choose_addr.dli_802addr.ioctl = ioctl;
 out_bind.choose_addr.dli_802addr.svc = svc;
 if(ctl & 3)
     out_bind.choose_addr.dli_802addr.eh_802.ctl.U_fmt=(u_char)ctl;
 else
     out_bind.choose_addr.dli_802addr.eh_802.ctl.I_S_fmt = ctl;
 out_bind.choose_addr.dli_802addr.eh_802.ssap = sap;
 out_bind.choose_addr.dli_802addr.eh_802.dsap = dsap;
 if ( ptype )
     bcopy(ptype,out_bind.choose_addr.dli_802addr.eh_802.osi_pi,5);
 if ( taddr )
      bcopy(taddr, out_bind.choose_addr.dli_802addr.eh_802.dst,
            DLI_EADDRSIZE);
 if ( bind(sock, &out_bind, sizeof(out_bind)) < 0 )
 {
      perror("dli_802, can't bind DLI socket");
      return(-1);
 }

  return(sock);
}
```

F.4.7 Calculating the Buffer Size

The buffer size must be no larger than the controllers on the communicating systems can handle, or you will lose data. The maximum buffer size for Ethernet packets is 1500 bytes.

The maximum buffer size for 802.3 packets is calculated as follows:

```
    bytes = 1500 - [ 2 + (control field == UI? 1:2) +
                    (Source SAP == SNAP SAP ? 5:0)]
```

The number of bytes in the control field and in the `Source SAP` are specified in the `bind` call.

The maximum buffer size for FDDI packets 4352 bytes.

F.4.8 Transferring Data

A DLI program can use the `write`, `send`, or `sendto` calls to send data and the `read`, `recv`, or `recvfrom` calls to receive data. The X's in Table F–2 indicate the conditions under which you can use the system calls as a function of the I/O control flag set up during the `bind` call.

Note

You must set the target address in the `bind` call when using the Normal control flag. You do not need to set the target address in the `bind` call when using the Exclusive or Default control flags. However, if you do not set the target address then you must use the `sendto` and `recvfrom` system calls.

Table F–2: Data Transfer System Calls Used with DLI

System Calls	Normal Control	Exclusive Control	Default Control
`write`	X		
`send`	X		
`sendto`	X	X	X
`read`	X		
`recv`	X		
`recvfrom`	X	X	X

When you set the control flag to `NORMAL`, set the target address in the `bind` call. Then use any of the following calls to transfer data: `write`, `send`, `sendto`, `read`, `recv`, `recvfrom`.

When you set the control flag to EXCLUSIVE, make the value of the target address in the bind call zero. Then, set the target address in the sendto call. Use only the sendto and recvfrom calls to transfer data.

When you set the control flag to DEFAULT, make the value of the target address in the bind call zero. Then use the sendto call to send data and set the target address in that call. Use the recvfrom call to determine the source address of any data.

F.4.9 Deactivating the Socket

When you have finished sending or receiving data, deactivate the socket by issuing the close system call.

F.5 DLI Programming Examples

This section includes the following DLI programming examples:

- A sample DLI client program using Ethernet format packets
- A sample DLI server program using Ethernet format packets
- A sample DLI client program using 802.3 format packets
- A sample DLI server program using 802.3 format packets
- A sample DLI program using getsockopt and setsockopt system calls

These programming examples are also available on line in the /usr/examples/dli directory.

F.5.1 Sample DLI Client Program Using Ethernet Format Packets

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <memory.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <net/if.h>
#include <net/route.h>
#include <dli/dli_var.h>

/*
 *      d l i _ e x a m p l e : d l i _ e t h
 *
 * Description: This program sends out a message to a node where a
 *              companion program, dli_ethd, echoes the message back.
 *              The ethernet packet format is used.  The ethernet
 *              address of the node where the companion program is
 *              running, the protocol type, and the message are
 *              supplied by the user.  The companion program should
 *              be started before executing this program.
 *
 * Inputs:      device, target address, protocol type, short message.
 *
 * Outputs:     Exit status.
 *
 * To compile:  cc -o dli_eth dli_eth.c
 *
 * Example:     dli_eth ln0 08-00-2b-02-e2-ff 6006 "Echo this"
 *
 * Comments:    This example demonstrates the use of the "NORMAL" I/O
 *              control flag.  The use of the "NORMAL" flag means that
 *              we can communicate only with a single specific node
 *              whose address is specified during the bind.  Because
 *              of this, we can use the normal write & read system
 *              calls on the socket, because the source/destination of
 *              all data that is read/written on the socket is fixed.
 *
 */

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

main(
    int argc,
    char **argv)
{
    struct sockaddr_dl sdl;
    size_t sdllen;
    int ch, fd, rsize, itarget[6], ptype, ioctlflg = DLI_NORMAL, errflg = 0;
    u_char inbuf[4800], u_char *src;

    memset(&sdl, 0, sizeof(sdl));
    while ((ch = getopt(argc, argv, "xp:")) != EOF) {
     case ’x’: ioctlflg = DLI_EXCLUSIVE; break;
     case ’p’: {
         if (sscanf(optarg, "%x", &ptype, &ch) != 1) {
            fprintf(stderr, "%s: invalid protocol type "s
               argv[0], optarg);
          errflg++;
          break;
         }
     }
     default: errflg++; break;
    }

    if (errflg || argc - optind < 5) {
        fprintf(stderr, "%s %s %s\n",
  "usage:",
  argv[0],
  "device lan-address short-message");
        exit(1);
    }

    /*
     * Get device name and unit number.
     */
    if (sscanf(argv[optind], "%[a-z]%hd%c", sdl.dli_device.dli_devname,
    &sdl.dli_device.dli_devnumber, &ch) != 2) {
 fprintf(stderr, "%s: invalid device name
        argv[0], argv[optind]);
 exit(1);
```

```
   }

   /*
    * Get the address to which we will be sending
    */
   if (sscanf(argv[++optind], "%x%*[:-]%x%*[:-]%x%*[:-]\
                               %x%*[:-]%x%*[:-]%x%c",
   &itarget[0], &itarget[1], &itarget[2],
   &itarget[3], &itarget[4], &itarget[5], &ch) != 6) {
fprintf(stderr, "%s: invalid lan address
       argv[0], argv[optind]);
exit(1);
   }

   /*
    * If the LAN Address is a multicast, then we can’t
    *  use DLI_NORMAL.  Use DLI_DEFAULT instead.
    */
   if ((itarget[0] & 1) && ioctflg == DLI_NORMAL)
    ioctlflg = DLI_DEFAULT;

   /*
    * Fill out sockaddr structure for bind/sento/recvfrom
    */
   sdl.dli_family = AF_DLI;
   if (ptype < GLOBAL_SAP) {
    sdl.dli_substructype = DLI_802;
    sdl.choose_addr.dli_802addr.ioctl = ioctlflg;
    sdl.choose_addr.dli_802addr.svc = TYPE1;
    sdl.choose_addr.dli_802addr.eh_802.dsap = ptype;
    sdl.choose_addr.dli_802addr.eh_802.ssap = ptype;
    sdl.choose_addr.dli_802addr.eh_802.ctl.U_fmt = UI_NPCMD;
    src = sdl.choose_addr.dli_802addr.eh_802.dst;
   } else {
    sdl.dli_substructype = DLI_ETHERNET;
    sdl.choose_addr.dli_eaddr.dli_ioctlflg = ioctlflg;
    sdl.choose_addr.dli_eaddr.dli_protype = ptype;
    src = sdl.choose_addr.dli_eaddr.dli_target;
   }
   /*
    * If we are using DLI_NORMAL, we must supply
    */
   if (ioctlflg == DLI_NORMAL) {
    src[0] = itarget[0]; src[1] = itarget[1]; src[2] = itarget[2];
    src[3] = itarget[3]; src[4] = itarget[4]; src[5] = itarget[5];
   }

   /*
    * Open a socket to DLI and then bind to our protocol/address.
    */
   if ((fd = socket(AF_DLI, SOCK_DGRAM, DLPROTO_DLI)) < 0) {
    fprintf(stderr, "%s: DLI open failed: %s\n",
         argv[0], strerror(errno));
    exit(1);
   }

   if (bind(fd, (struct sockaddr *) &sdl, sizeof(sdl)) < 0) {
    fprintf(stderr, "%s: DLI bind failed: %s\n",
         argv[0], strerror(errno));
    exit(2);
   }

   if (ioctlflg != DLI_NORMAL) {
    src[0] = itarget[0]; src[1] = itarget[1]; src[2] = itarget[2];
```

```
     src[3] = itarget[3]; src[4] = itarget[4]; src[5] = itarget[5];
    }

    /* send response to originator. */
    sdllen = sizeof(sdl);
    if (sendto(fd, argv[4], strlen(argv[4]), 0,
        (struct sockaddr *) &sdl, sdllen) < 0) {
     fprintf(stderr, "%s: DLI transmission failed: %s\n",
          argv[0], strerror(errno));
     exit(1);
    }

    if ((rsize = recvfrom(fd, inbuf, sizeof(inbuf), 0,
                (struct sockaddr *) &sdl, &sdllen)) < 0 ) {
     fprintf(stderr, "%s: DLI reception failed: %s\n",
          argv[0], strerror(errno));
   exit(1);
    }

    /* check header */
    if (sdllen != sizeof(struct sockaddr_dl)) {
     fprintf(stderr, "%s, incorrect header supplied\n", argv[0]);
     exit(1);
    }

    if (from.dli_substructype == DLI_802)
     src = from.dli_choose_addr.dli_802addr.eh_802.dst;
    else
     src = from.dli_choose_addr.dli_eaddr.dli_target;

    /* any data? */
    fprintf(stderr, "%s: %sdata received from ", argv[0],
            rsize ?  : "NO ");
    fprintf(stderr, "%02x-%02x-%02x-%02x-%02x-%02x",
         src[0], src[1], src[2], src[3], src[4], src[5]);
    if (from.dli_substructype == DLI_802)
     fprintf(stderr, " SAP %02x\n\n",
          sdl.choose_addr.dli_802addr.eh_802.ssap & ~1);
    else
     fprintf(stderr, " on protocol type %04x\n\n",
          sdl.choose_addr.dli_eaddr.dli_protype);

    /* print results */
    printf("%s\n", inbuf);
    close(fd);
    return 0;
}
```

F.5.2 Sample DLI Server Program Using Ethernet Format Packets

```
#ifndef lint
static char *rcsid = "@(#)$RCSfile: ap-dli.sgml,v $ \
         $Revision: 1.1.6.3 $ (DEC) $Date: 1999/07/08 20:46:48 $";
#endif

#include <stdio.h>
#include <ctype.h>
#include <errno.h>
#include <strings.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <net/if.h>
#include <netinet/in.h>
```

```
#include <netinet/if_ether.h>
#include <dli/dli_var.h>
#include <sys/ioctl.h>

extern int errno;


/*
 *      d l i _ e x a m p l e : d l i _ e t h d
 *
 * Description: This daemon program transmits any message it
 *  receives to the originating system, i.e., it echoes the
 *  message back.  The device and protocol type are supplied
 *  by the user.  The program uses ethernet format packets.
 *
 * Inputs:      device, protocol type.
 *
 * Outputs:     Exit status.
 *
 * To compile:  cc -o dli_ethd dli_ethd.c
 *
 * Example:     dli_ethd de0 6006
 *
 * Comments:    This example demonstrates the use of the "DEFAULT"
 *  I/O control flag, and the recvfrom & sendto system calls.
 *  By specifying "DEFAULT" when binding the DLI socket to
 *  the device we inform the system that this program will
 *  receive any ethernet format packet with the given
 *  protocol type which is not meant for any other program
 *  on the system.  Since packets may arrive from
 *  different systems we use the recvfrom call to read the
 *  packets.  This call gives us access to the packet
 *  header information so that we can determine where the
 *  packet came from.  When we write on the socket we must
 *  use the sendto system call to explicitly give the
 *  destination of the packet.
 */

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

main(argc, argv, envp)
int argc;
char **argv, **envp;

{

    u_char inbuf[1500], outbuf[1500];
    u_char devname[16];
    u_char target_eaddr[6];
    char *cp;
    int rsize;
    unsigned int devunit;
    int i, sock, fromlen;
    unsigned int ptype;
    struct sockaddr_dl from;

    if ( argc < 3 )
    {
        fprintf(stderr,
                "usage: %s device hex-protocol-type\n", argv[0]);
        exit(1);
    }

    /* get device name and unit number. */
    bzero(devname, sizeof(devname));
    i = 0;
    cp = argv[1];
    while ( isalpha(*cp) )
        devname[i++] = *cp++;
    sscanf(cp, "%d", &devunit);

    /* get protocol type */
    sscanf(argv[2], "%x", &ptype);

    /* open dli socket */
    if
    ((sock = dli_econn(devname, devunit, ptype, NULL, \
             DLI_DEFAULT))<0)
    {
        perror("dli_ethd, dli_econn failed");
        exit(1);
    }

    while ( 1 ) {
        /* wait for message */
        from.dli_family = AF_DLI;
        fromlen = sizeof(struct sockaddr_dl);
        if ((rsize = recvfrom(sock, inbuf, sizeof(inbuf),
                              NULL, &from, &fromlen)) < 0 ) {
            sprintf(inbuf, "%s: DLI reception failed", argv[0]);
            perror(inbuf);
            exit(2);
        }

        /* check header */
        if ( fromlen != sizeof(struct sockaddr_dl) ) {
            fprintf(stderr,"%s, incorrect header supplied\n",argv[0]);
            continue;
        }

        /* any data? */
        if ( ! rsize )
            fprintf(stderr, "%s, NO data received from ", argv[0]);
        else
            fprintf(stderr, "%s, data received from ", argv[0]);
        for ( i = 0; i < 6; i++ )
            fprintf(stderr, "%x%s",
                    from.choose_addr.dli_eaddr.dli_target[i],
                    ((i<5)?"-":" "));
        fprintf(stderr, "on protocol type %x\n",
                from.choose_addr.dli_eaddr.dli_protype);

        /* send response to originator. */
        if ( sendto(sock, inbuf, rsize, NULL, &from, fromlen) < 0 ) {
            sprintf(outbuf, "%s: DLI transmission failed", argv[0]);
            perror(outbuf);
            exit(2);
        }
    }
}
/*
 *              d l i _ e c o n n
 *
 *
 *
 * Description:
 *      This subroutine opens a dli socket, then binds an associated
 *      device name and protocol type to the socket.
 *
 * Inputs:
 *      devname         = ptr to device name
 *      devunit         = device unit number
 *      ptype           = protocol type
 *      taddr           = target address
 *      ioctl           = io control flag
 *
 * Outputs:
 *      returns         = socket handle if success, otherwise -1
 */

dli_econn(devname, devunit, ptype, taddr, ioctl)
char *devname;
unsigned devunit;
unsigned ptype;
u_char *taddr;
u_char ioctl;
{
    int i, sock;
    struct sockaddr_dl out_bind;

    if ( (i = strlen(devname)) >
          sizeof(out_bind.dli_device.dli_devname) )
    {
         fprintf(stderr, "dli_ethd: bad device name");
         return(-1);
    }

    if ((sock = socket(AF_DLI, SOCK_DGRAM, DLPROTO_DLI)) < 0)
    {
         perror("dli_ethd, can’t open DLI socket");
         return(-1);
    }
    /*   Fill out bind structure    */

    bzero(&out_bind, sizeof(out_bind));
    out_bind.dli_family = AF_DLI;
    out_bind.dli_substructype = DLI_ETHERNET;
    bcopy(devname, out_bind.dli_device.dli_devname, i);
    out_bind.dli_device.dli_devnumber = devunit;
    out_bind.choose_addr.dli_eaddr.dli_ioctlflg = ioctl;
    out_bind.choose_addr.dli_eaddr.dli_protype = ptype;
    if ( taddr )
         bcopy(taddr, out_bind.choose_addr.dli_eaddr.dli_target,
               DLI_EADDRSIZE);

    if ( bind(sock, &out_bind, sizeof(out_bind)) < 0 )
    {
         perror("dli_ethd, can’t bind DLI socket");
         return(-1);
    }

    return(sock);
}
```

F.5.3 Sample DLI Client Program Using 802.3 Format Packets

```
#ifndef lint
static  char  *sccsid = "@(#)dli_802.c 1.1 (DEC OSF/1) 5/29/92";
#endif lint

#include <stdio.h>
#include <ctype.h>
#include <errno.h>
#include <strings.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <net/if.h>
#include <netinet/in.h>
#include <netinet/if_ether.h>
#include <dli/dli_var.h>
#include <sys/ioctl.h>

extern int errno;

#define PROTOCOL_ID      {0x00, 0x00, 0x00, 0x00, 0x5}
u_char protocolid[] = PROTOCOL_ID;


/*
 *      d l i _ e x a m p l e : d l i _ 8 0 2
 *
 * Description: This program sends out a message to a system
 *    where a companion program, dli_802d, echoes the message
 *    back.  The 802.3 packet format is used.  The ethernet
 *    address of the system where the companion program is
 *    running, the sap, and the message are supplied by the
 *    user.  The companion program should be started before
 *    executing this program.
 *
 * Inputs:      device, target address, sap, short message.
 *
 * Outputs:     Exit status.
 *
 */
#ifndef lint
static char *rcsid = "@(#)$RCSfile: ap-dli.sgml,v $ \
            $Revision: 1.1.6.3 $ (DEC) $Date: 1999/07/08 20:46:48 $";
#endif


#include <stdio.h>
#include <ctype.h>
#include <errno.h>
#include <strings.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <net/if.h>
#include <netinet/in.h>
#include <netinet/if_ether.h>
#include <dli/dli_var.h>
#include <sys/ioctl.h>

extern int errno;

#define PROTOCOL_ID      {0x00, 0x00, 0x00, 0x00, 0x5}
u_char protocolid[] = PROTOCOL_ID;
```

```
/*
 *      d l i _ e x a m p l e : d l i _ 8 0 2
 *
 * Description: This program sends out a message to a system
 *    where a companion program, dli_802d, echoes the message
 *    back.  The 802.3 packet format is used.  The ethernet
 *    address of the system where the companion program is
 *    running, the sap, and the message are supplied by the
 *    user.  The companion program should be started before
 *    executing this program.
 *
 * Inputs:      device, target address, sap, short message.
 *
 * Outputs:     Exit status.
 *
 * To compile:  cc -o dli_802 dli_802.c
 *
 * Example:     dli_802 qe0 08-00-2b-02-e2-ff ac "Echo this"
 *
 * Comments:    This example demonstrates the use of 802 "TYPE1"
 *    service.  With TYPE1 service, the processing of
 *    XID and TEST messages is handled transparently by
 *    DLI, i.e., this program doesn't have to be concerned
 *    with handling them.  If the SNAP SAP (0xAA) is
 *    selected, a 5 byte protocol id is also required.
 *    This example automatically uses a protocol id of
 *    of PROTOCOL_ID when the SNAP SAP is used.  Also,
 *    note the use of DLI_NORMAL for the i/o control flag.
 *    DLI makes use of this only when that SNAP_SAP/Protocol
 *    ID pair is used. DLI will filter all incoming messages
 *    by comparing the Ethernet source address and Protocol
 *    ID against the target address and Protocol ID set up
 *    in the bind call.  Only if a match occurs will DLI
 *    pass the message up to the application.
 */

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

main(argc, argv, envp)
int argc;
char **argv, **envp;

{

    u_char inbuf[1500], outbuf[1500];
    u_char target_eaddr[6];
    u_char devname[16];
    int rsize, devunit;
    char *cp;
    int i, sock, fromlen;
    struct sockaddr_dl from;
    unsigned int obsiz, byteval;
    u_int sap;
    u_char  *pi = 0;


    if ( argc < 5 )
    {
        fprintf(stderr, "%s %s %s\n",
                "usage:",
                argv[0],
                "device ethernet-address hex-sap short-message");
        exit(1);
    }


    /* get device name and unit number. */
    bzero(devname, sizeof(devname));
    i = 0;
    cp = argv[1];
    while ( isalpha(*cp) )
        devname[i++] = *cp++;
    sscanf(cp, "%d", &devunit);

    /* get phys addr of remote system */
    bzero(target_eaddr, sizeof(target_eaddr));
    i = 0;
    cp = argv[2];
    while ( *cp ) {
        if ( *cp == ’-’ ) {
            cp++;
            continue;
        }
        else {
            sscanf(cp, "%2x", &byteval );
     target_eaddr[i++] = byteval;
            cp += 2;
        }
    }

    /* get sap */
    sscanf(argv[3], "%x", &sap);

    /* get message */
    bzero(outbuf, sizeof(outbuf));
    if ( (obsiz = strlen(argv[4])) > 1500 ) {
        fprintf(stderr, "%s: message is too long\n", argv[0]);
        exit(2);
    }
    strcpy(outbuf, argv[4]);


    /* open dli socket. notice that if (and only if) the */
    /* snap sap was selected then a protocol id must also */
    /* be provided. */
    if ( sap == SNAP_SAP )
        pi = protocolid;
    if ( (sock = dli_802_3_conn(devname, devunit, pi, target_eaddr,
                  DLI_NORMAL, TYPE1, sap, sap, UI_NPCMD)) < 0 ) {
        perror("dli_802, dli_econn failed");
        exit(3);
    }


    /* send message to target. minimum message size is 46 bytes. */
    if ( write(sock, outbuf, (obsiz < 46 ? 46 : obsiz)) < 0 ) {
        sprintf(outbuf, "%s: DLI transmission failed", argv[0]);
        perror(outbuf);
        exit(4);
    }

    /* wait for response from correct address */
    while (1) {
        bzero(&from, sizeof(from));
        from.dli_family = AF_DLI;
        fromlen = sizeof(struct sockaddr_dl);
        if ((rsize = recvfrom(sock, inbuf, sizeof(inbuf),
                              NULL, &from, &fromlen)) < 0 ) {
              sprintf(inbuf, "%s: DLI reception failed", argv[0]);
              perror(inbuf);
              exit(5);
        }
        if ( fromlen != sizeof(struct sockaddr_dl) ) {
              fprintf(stderr,"%s, invalid address size\n",argv[0]);
              exit(6);
        }
        if ( bcmp(from.choose_addr.dli_802addr.eh_802.dst,
                  target_eaddr, sizeof(target_eaddr)) == 0 )
                break;
    }

    if ( ! rsize ) {
        fprintf(stderr, "%s, no data returned\n", argv[0]);
        exit(7);
    }
    /* print message */
    printf("%s\n", inbuf);

    close(sock);

}

/*
 *              d l i _8 0 2 _ 3 _ c o n n
 *
 *
 *
 * Description:
 *      This subroutine opens a dli 802.3 socket, then binds an
 *      associated device name and protocol type to the socket.
 *
 * Inputs:
 *      devname         = ptr to device name
 *      devunit         = device unit number
 *      ptype           = protocol type
 *      taddr           = target address
 *      ioctl           = io control flag
 *      svc             = service class
 *      sap             = source sap
 *      dsap            = destination sap
 *      ctl             = control field
 *
 *
 * Outputs:
 *      returns         = socket handle if success, otherwise -1
 *
 *
 */

dli_802_3_conn (devname,devunit,ptype,taddr,ioctl,svc,sap,dsap,ctl)
char *devname;
u_short devunit;
u_char *ptype;
u_char *taddr;
u_char ioctl;
u_char svc;
u_char sap;
u_char dsap;
u_short ctl;

{
    int i, sock;
    struct sockaddr_dl out_bind;

    if ( (i = strlen(devname)) >
         sizeof(out_bind.dli_device.dli_devname) )
    {
         fprintf(stderr, "dli_802: bad device name");
         return(-1);
    }

    if ((sock = socket(AF_DLI, SOCK_DGRAM, DLPROTO_DLI)) < 0)
    {
         perror("dli_802, can’t open DLI socket");
         return(-1);
    }

    /*
     * fFill out bind structure.  Note that we need to determine
     * whether the ctl field is 8 bits (unnumbered format) or
     * 16 bits (informational/supervisory format).  We do this
     * by checking the low order 2 bits, which are both 1 only
     * for unnumbered control fields.
     */
    bzero(&out_bind, sizeof(out_bind));
    out_bind.dli_family = AF_DLI;
    out_bind.dli_substructype = DLI_802;
    bcopy(devname, out_bind.dli_device.dli_devname, i);
    out_bind.dli_device.dli_devnumber = devunit;
    out_bind.choose_addr.dli_802addr.ioctl = ioctl;
    out_bind.choose_addr.dli_802addr.svc = svc;
    if(ctl & 3)
        out_bind.choose_addr.dli_802addr.eh_802.ctl.U_fmt=\
              (u_char)ctl;
    else
        out_bind.choose_addr.dli_802addr.eh_802.ctl.I_S_fmt = \
              ctl;
    out_bind.choose_addr.dli_802addr.eh_802.ssap = sap;
    out_bind.choose_addr.dli_802addr.eh_802.dsap = dsap;
    if ( ptype )
        bcopy(ptype,out_bind.choose_addr.dli_802addr.eh_802.osi_pi,\
              5);
    if ( taddr )
         bcopy(taddr, out_bind.choose_addr.dli_802addr.eh_802.dst,
               DLI_EADDRSIZE);
    if ( bind(sock, &out_bind, sizeof(out_bind)) < 0 )
    {
         perror("dli_802, can’t bind DLI socket");
         return(-1);
    }

    return(sock);
}
```

F.5.4 Sample DLI Server Program Using 802.3 Format Packets

```
#ifndef lint
static char *rcsid = "@(#)$RCSfile: ap-dli.sgml,v $ \
           $Revision: 1.1.6.3 $ (DEC) $Date: 1999/07/08 20:46:48 $";
#endif


#include <stdio.h>
#include <ctype.h>
#include <errno.h>
#include <strings.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <net/if.h>
#include <netinet/in.h>
#include <netinet/if_ether.h>
#include <dli/dli_var.h>
#include <sys/ioctl.h>

extern int errno;

#define PROTOCOL_ID      {0x00, 0x00, 0x00, 0x00, 0x5}
u_char protocolid[] = PROTOCOL_ID;

/*
 *      d l i _ e x a m p l e : d l i _ 8 0 2 d
 *
 * Description: This daemon program transmits any message it
 *   receives to the originating system, i.e., it echoes the
 *   message back.  The device and sap are supplied by the
 *   user.  The program uses 802.3 format packets.
 *
 * Inputs:      device, sap.
 *
 * Outputs:     Exit status.
 *
 * To compile:  cc -o dli_802d dli_802d.c
 *
 * Example:     dli_802d de0 ac
 *
 * Comments:  This example demonstrates the recvfrom & sendto
 *   system calls.  Since packets may arrive from different
 *   systems we use the recvfrom call to read the packets.
 *   This call gives us access to the packet header information
 *   so that we can determine where the packet came from.
 *   When we write on the socket we must use the sendto
 *   system call to explicitly give the destination of
 *   the packet.  The use of the "DEFAULT" I/O control flag
 *   only applies (i.e. only has an affect) when the SNAP SAP
 *   is used.  When the SNAP SAP is used, any arriving packets
 *   which have the specified protocol id and which are not
 *   destined for some other program will be given to this
 *   program.
 */

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

```
main(argc, argv, envp)
int argc;
char **argv, **envp;

{

    u_char inbuf[1500], outbuf[1500];
    u_char devname[16];
    u_char target_eaddr[6];
    char *cp;
    int rsize, devunit;
    int i, sock, fromlen;
    u_char tmpsap, sap;
    struct sockaddr_dl from;
    u_char *pi = 0;

    if ( argc < 3 )
    {
        fprintf(stderr, "usage: %s device hex-sap\n", argv[0]);
        exit(1);
    }


    /* get device name and unit number. */
    bzero(devname, sizeof(devname));
    i = 0;
    cp = argv[1];
    while ( isalpha(*cp) )
        devname[i++] = *cp++;
    sscanf(cp, "%d", &devunit);

    /* get sap */
    sscanf(argv[2], "%x", &sap);

    /* open dli socket. note that if (and only if) the snap sap */
    /* was selected then a protocol id must also be specified. */
    if ( sap == SNAP_SAP )
        pi = protocolid;
    if ((sock = dli_802_3_conn(devname, devunit, pi, target_eaddr,
                    DLI_DEFAULT, TYPE1, sap, sap, UI_NPCMD)) < 0) {
        perror("dli_802d, dli_conn failed");
        exit(1);
    }

    /* listen and respond */
    while ( 1 ) {
        /* wait for message */
        from.dli_family = AF_DLI;
        fromlen = sizeof(struct sockaddr_dl);
        if ((rsize = recvfrom(sock, inbuf, sizeof(inbuf), NULL,
                              &from, &fromlen)) < 0 ) {
            sprintf(inbuf, "%s: DLI reception failed", argv[0]);
            perror(inbuf);
            exit(2);
        }

        /* check header */
        if ( fromlen != sizeof(struct sockaddr_dl) ) {
            fprintf(stderr,"%s, incorrect header supplied\n",\
                 argv[0]);
            continue;
        }
        /*
         * Note that DLI swaps the source & destination saps and
         * lan addresses in the sockaddr_dl structure returned
         * by the recvfrom call.  That is, it places the DSAP in
         * eh_802.ssap and the SSAP in eh_802.dsap; it also places
         * the destination lan address in eh_802.src and the source
         * lan address in eh_802.dst.  This allows for minimal to
         * no manipulation of the address structure for subsequent
         * sendto or dli connection calls.
         */

        /* any data? */
        if ( ! rsize )
            fprintf(stderr, "%s: NO data received from ", \
                 argv[0]);
        else
            fprintf(stderr, "%s: data received from ", argv[0]);
        for ( i = 0; i < 6; i++ )
            fprintf(stderr, "%x%s",
                    from.choose_addr.dli_802addr.eh_802.dst[i],
                    ((i<5)?"-":" "));
        fprintf(stderr, "\n     on dsap %x ",
                from.choose_addr.dli_802addr.eh_802.ssap);
        if ( from.choose_addr.dli_802addr.eh_802.dsap == \
                 SNAP_SAP )
            fprintf(stderr,
               "(SNAP SAP), protocol id = %x-%x-%x-%x-%x\n    ",
               from.choose_addr.dli_802addr.eh_802.osi_pi[0],
               from.choose_addr.dli_802addr.eh_802.osi_pi[1],
               from.choose_addr.dli_802addr.eh_802.osi_pi[2],
               from.choose_addr.dli_802addr.eh_802.osi_pi[3],
               from.choose_addr.dli_802addr.eh_802.osi_pi[4]);
        fprintf(stderr, " from ssap %x ",
                from.choose_addr.dli_802addr.eh_802.dsap);
        fprintf(stderr, "\n\n");

        /* send response to originator. */
        if ( from.choose_addr.dli_802addr.eh_802.dsap == \
                 SNAP_SAP )
            bcopy(protocolid,
                  from.choose_addr.dli_802addr.eh_802.osi_pi, 5);
        if ( sendto(sock, inbuf, rsize, NULL, &from, fromlen) \
                 < 0 ) {
            sprintf(outbuf, "%s: DLI transmission failed", \
                 argv[0]);
            perror(outbuf);
            exit(2);
        }
    }
}

/*
 *              d l i _8 0 2 _ 3 _ c o n n
 *
 *
 *
 * Description:
 *      This subroutine opens a dli 802.3 socket, then binds an
 *      associated device name and protocol type to the socket.
 *
 * Inputs:
 *      devname         = ptr to device name
 *      devunit         = device unit number
 *      ptype           = protocol type
 *      taddr           = target address
 *      ioctl           = io control flag
 *      svc             = service class
 *      sap             = source sap
 *      dsap            = destination sap
 *      ctl             = control field
 *
 *
 * Outputs:
 *      returns         = socket handle if success, otherwise -1
 *
 *
 */

dli_802_3_conn (devname,devunit,ptype,taddr,ioctl,svc,sap,\
        dsap,ctl)
char *devname;
u_short devunit;
u_char *ptype;
u_char *taddr;
u_char ioctl;
u_char svc;
u_char sap;
u_char dsap;
u_short ctl;
{
    int i, sock;
    struct sockaddr_dl out_bind;

    if ( (i = strlen(devname)) >
         sizeof(out_bind.dli_device.dli_devname) )
    {
         fprintf(stderr, "dli_802d: bad device name");
         return(-1);
    }

    if ((sock = socket(AF_DLI, SOCK_DGRAM, DLPROTO_DLI)) < 0)
    {
         perror("dli_802d, can't open DLI socket");
         return(-1);
    }

    /*
     * fill out bind structure.  note that we need to determine
     * whether the ctl field is 8 bits (unnumbered format) or
     * 16 bits (informational/supervisory format).  We do this
     * by checking the low order 2 bits, which are both 1 only
     * for unnumbered control fields.
     */
    bzero(&out_bind, sizeof(out_bind));
    out_bind.dli_family = AF_DLI;
    out_bind.dli_substructype = DLI_802;
    bzero(&out_bind, sizeof(out_bind));
    out_bind.dli_family = AF_DLI;
    out_bind.dli_substructype = DLI_802;
    bcopy(devname, out_bind.dli_device.dli_devname, i);
    out_bind.dli_device.dli_devnumber = devunit;
    out_bind.choose_addr.dli_802addr.ioctl = ioctl;
    out_bind.choose_addr.dli_802addr.svc = svc;
    if(ctl & 3)
        out_bind.choose_addr.dli_802addr.eh_802.ctl.U_fmt=\
            (u_char)ctl;
    else
        out_bind.choose_addr.dli_802addr.eh_802.ctl.I_S_fmt = \
            ctl;
    out_bind.choose_addr.dli_802addr.eh_802.ssap = sap;
    out_bind.choose_addr.dli_802addr.eh_802.dsap = dsap;
    if ( ptype )
        bcopy(ptype,out_bind.choose_addr.dli_802addr.eh_802.osi_pi,\
            5);
    if ( taddr )
         bcopy(taddr, out_bind.choose_addr.dli_802addr.eh_802.dst,
               DLI_EADDRSIZE);

    if ( bind(sock, &out_bind, sizeof(out_bind)) < 0 )
    {
         perror("dli_802d, can't bind DLI socket");
         return(-1);
    }

    return(sock);
}
```

F.5.5 Sample DLI Program Using getsockopt and setsockopt

```
#ifndef lint
static  char  *sccsid = "@(#)dli_setsockopt.c   1.5  3/27/90";
#endif lint

#include <stdio.h>
#include <ctype.h>
#include <errno.h>
#include <strings.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <net/if.h>
#include <netinet/in.h>
#include <netinet/if_ether.h>
#include <dli/dli_var.h>
#include <sys/ioctl.h>

extern int errno;
int debug = 0;

#define PROTOCOL_ID      {0x00, 0x00, 0x00, 0x00, 0x5}
#define CUSTOMER0        {0xab, 0x00, 0x04, 0x00, 0x00, 0x00}
#define CUSTOMER1        {0xab, 0x00, 0x04, 0x00, 0x00, 0x01}

u_char mcast0[] = CUSTOMER0;
u_char mcast1[] = CUSTOMER1;
u_char protocolid[] = PROTOCOL_ID;

/*
 *
 *      d l i  e x a m p l e : d l i  s e t s o c k o p t
 *
 * Description: This program demonstrates the use of the DLI
 *    get- and setsockopt calls.  It opens a socket, enables
 *    2 multicast addresses, changes the 802 control
 *    field, enables a number of group saps supplied by
 *    the user, and reads the group saps that are enabled.
 *
 * Inputs:      device, sap, group-saps.
 *
 * Outputs:     Exit status.
 *
 * To compile:  cc -o dli_setsockopt dli_setsockopt.c
 *
 * Example:     dli_setsockopt qe0 ac  5 9 d
 *
 * Comments:    When a packet arrives with a group dsap,
 *    all dli programs that have that group sap enabled will
 *    receive copies of that packet.  Group saps are
 *    those with the low order bit set.  Group sap 1
 *    is currently not allowed for customer use. Group
 *    saps with the second bit set (eg 3,7,etc) are
 *    reserved by IEEE.
 */

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

main(argc, argv, envp)
int argc;
char **argv, **envp;

{

    u_char inbuf[1500], outbuf[1500];
    u_char devname[16];
    u_char target_eaddr[6];
    char *cp;
    int rsize, devunit;
    int i, j, k, sock, fromlen;
    u_short obsiz;
    u_char tmpsap, sap;
    struct sockaddr_dl from;
    u_char *pi = 0;
    u_char out_opt[1000], in_opt[1000];
    int optlen, ioptlen = sizeof(in_opt);

    if ( argc < 4 )
    {
        fprintf(stderr, "usage: %s device hex-sap hex-groupsaps\n",
        argv[0]);
        exit(1);
    }

    /* get device name and unit number. */
    bzero(devname, sizeof(devname));
    i = 0;
    cp = argv[1];
    while ( isalpha(*cp) )
    devname[i++] = *cp++;
    sscanf(cp, "%d", &devunit);

    /* get protocol type */
    sscanf(argv[2], "%x", &sap);

    /* open dli socket */
    if ( sap == SNAP_SAP ) {
        fprintf(stderr,
                "%s: can’t use SNAP_SAP in USER mode\n", argv[0]);
        exit(1);
    }
    if ( (sock = dli_802_3_conn(devname, devunit, pi,\
           target_eaddr,
                     DLI_DEFAULT, USER, sap, sap, UI_NPCMD)) \
           < 0 ) {
        perror("dli_setsockopt: dli_conn failed");
        exit(1);
    }

    /* enable two multicast addresses */
    bcopy(mcast0, out_opt, sizeof(mcast0));
    bcopy(mcast1, out_opt+sizeof(mcast0), sizeof(mcast1));

    if ( setsockopt(sock, DLPROTO_DLI, DLI_MULTICAST, \
           &out_opt[0],
                    (sizeof(mcast0) + sizeof(mcast1))) < 0 ) {
        perror("dli_setsockopt: can’t enable multicast");
    }

    /* set 802 control field */
    out_opt[0] = TEST_PCMD;
    optlen = 1;
    if
    (setsockopt(sock,DLPROTO_DLI,DLI_SET802CTL,&out_opt[0],\
           optlen)<0){
        perror("dli_setsockopt: Can’t set 802 control");
        exit(1);
    }

    /* enable GSAPs supplied by user */
    j = 3;
    i = 0;
    while (j < argc ) {
        sscanf(argv[j++], "%x", &k);
        out_opt[i++] = k;
    }
    optlen = i;
    if
    (setsockopt(sock,DLPROTO_DLI,DLI_ENAGSAP,&out_opt[0],\
           optlen) < 0){
        perror("dli_setsockopt: Can’t enable gsap");
        exit(1);
    }

    /* verify all gsaps are enabled */
    bzero(in_opt, (ioptlen = sizeof(in_opt)));
    if
    (getsockopt(sock,DLPROTO_DLI,DLI_GETGSAP,in_opt,\
           &ioptlen) < 0){
        perror("dli_setsockopt: DLI getsockopt 2 failed");
        exit(1);
    }
    printf("number of enabled GSAPs = %d, GSAPS:", ioptlen);
    for(i = 0; i < ioptlen; i++) {
        if ( ! (i % 10) )
            printf("\n");
        printf("%2x ",in_opt[i]);
    }
    printf("\n");

    /* disable all but the last 4 or all GSAPs, */
    /* whichever is smallest */
    if ( optlen > 4 )
```

```
        optlen -= 4;
    if
    (setsockopt(sock,DLPROTO_DLI,DLI_DISGSAP,&out_opt[0],\
           optlen) < 0){
        perror("dli_setsockopt: Can't disable gsap");
    }

    /* verify some gsaps still enabled */
    bzero(in_opt, (ioptlen = sizeof(in_opt)));
    if
    (getsockopt(sock,DLPROTO_DLI,DLI_GETGSAP,in_opt,\
           &ioptlen) < 0){
        perror("dli_setsockopt: getsockopt 3 failed");
        exit(1);
    }
    printf("number of enabled GSAPs = %d, GSAPS:", ioptlen);
    for(i = 0; i < ioptlen; i++) {
        if ( ! (i % 10) )
    printf("\n");
        printf("%2x ",in_opt[i]);
    }
    printf("\n");

}

/*
 *              d l i _8 0 2 _ 3 _ c o n n
 *
 *
 *
 * Description:
 *      This subroutine opens a dli 802.3 socket and then binds
 *      an associated device name and protocol type to it.
 *
 * Inputs:
 *      devname    = ptr to device name
 *      devunit    = device unit number
 *      ptype      = protocol type
 *      taddr      = target address
 *      ioctl      = io control flag
 *      svc        = service class
 *      sap        = source sap
 *      dsap       = destination sap
 *      ctl        = control field
 *
 *
 * Outputs:
 *      returns    = socket handle if success, otherwise -1
 *
 */

dli_802_3_conn (devname,devunit,ptype,taddr,ioctl,svc,sap,\
       dsap,ctl)
char *devname;
u_short devunit;
u_char *ptype;
u_char *taddr;
u_char ioctl;
u_char svc;
u_char sap;
u_char dsap;
u_short ctl;
{
    int i, sock;
    struct sockaddr_dl out_bind;

    if ( (i = strlen(devname)) >
         sizeof(out_bind.dli_device.dli_devname) )
    {
        fprintf(stderr, "dli_setsockopt: bad device name");
        return(-1);
    }

    if ((sock = socket(AF_DLI, SOCK_DGRAM, DLPROTO_DLI)) < 0)
    {
        perror("dli_setsockopt: can't open DLI socket");
        return(-1);
    }

    /*
     * Fill out bind structure
     */
    bzero(&out_bind, sizeof(out_bind));
    out_bind.dli_family = AF_DLI;
    out_bind.dli_substructype = DLI_802;
    bcopy(devname, out_bind.dli_device.dli_devname, i);
    out_bind.dli_device.dli_devnumber = devunit;
    out_bind.choose_addr.dli_802addr.ioctl = ioctl;
    out_bind.choose_addr.dli_802addr.svc = svc;
    if(ctl & 3)
        out_bind.choose_addr.dli_802addr.eh_802.ctl.U_fmt=\
            (u_char)ctl;
    else
        out_bind.choose_addr.dli_802addr.eh_802.ctl.I_S_fmt = \
            ctl;
    out_bind.choose_addr.dli_802addr.eh_802.ssap = sap;
    out_bind.choose_addr.dli_802addr.eh_802.dsap = dsap;
    if ( ptype )
        bcopy(ptype,out_bind.choose_addr.dli_802addr.eh_802.osi_pi,\
            5);
    if ( taddr )
        bcopy(taddr, out_bind.choose_addr.dli_802addr.eh_802.dst,
              DLI_EADDRSIZE);
    if ( bind(sock, &out_bind, sizeof(out_bind)) < 0 )
    {
        perror("dli_setsockopt: can't bind DLI socket");
        return(-1);
    }

    return(sock);
}
```

Glossary

active user

In an XTI transport connection, the transport user that initiated the connection. See also **client process** and **passive user**.

Address Resolution Protocol (ARP)

The Internet (TCP/IP) Protocol that can dynamically resolve an Internet address to a physical hardware address. ARP can be used only across a single physical network and in networks that support the hardware broadcast feature.

asynchronous event

See **event**.

asynchronous execution

1. Execution of processes or threads in which each process or thread does not await the completion of the others before starting.
2. In XTI, a mode of execution that notifies the transport user of an event without forcing it to wait.

Berkeley Software Distribution

UNIX software release of the Computer Systems Research Group (CSRG) of the University of California at Berkeley.

blocking mode

See **synchronous execution**.

BSD socket interface

A transport-layer interface provided for applications to perform interprocess communication between two unrelated processes on a single system or on multiply connected systems. This interprocess communications facility allows programs to use sockets for communications between other programs, protocols, and devices.

client process

In the client/server model of communication, a process that requests services from a server process. See also **active user**.

communication domain

An abstraction used by the interprocess communication facility of a system to define the properties of a network. Properties include a set of communication protocols, rules for manipulating and interpreting names, and the ability to transmit access rights.

connection-oriented mode

A mode of service supported by a transport endpoint for transmitting data over an established connection.

connectionless mode

A mode of service supported by a transport endpoint that requires no established connection for transmitting data. Data is delivered in self-contained units, called **datagrams**.

datagram

A unit of data that is transmitted across a network by the connectionless service of a transport provider. In addition to user data, a datagram includes the information needed for its delivery. It is self-contained, in that it has no relationship to any datagrams previously or successively transmitted.

datagram socket

Socket that provides datagrams consisting of individual messages for transmission in connectionless mode.

error

In XTI, an indicator that is returned by a function when it encounters a system or library error in the process of executing. The object is to allow applications to take an action based on the returned error code.

eSNMP

The Extensible Simple Network Protocol (eSNMP) enables you to create subagents to be mananged by an SNMP management station. See Chapter 6.

Ethernet

A 10-megabit baseband local area network (LAN) using carrier sense multiple access with collision detection (CSMA/CD). The network allows multiple stations to access the medium at will without prior coordination, and avoids contention by using carrier sense and deference, and detection and transmission.

ETSDU

See **Expedited Transport Service Data Unit** and **out-of-band data**.

event

An occurrence, or happening, that is significant to a transport user. Events are asynchronous, in that they do not happen as a result of an action taken by the user.

event management

A mechanism by which transport providers notify transport users of the occurrence of significant events.

expedited data

Data that is considered urgent. The semantics of this data are defined by the transport provider. See also **out-of-band data**.

Expedited Transport Service Data Unit

In XTI, an expedited message in which the identity of the data unit is preserved from one end of a transport connection to the other.

file descriptor

A small unsigned integer that a UNIX system uses to identify a file. A file descriptor is created by a process through issuing an open system call for the file name. A file descriptor ceases to exist when it is no longer held by any process.

host group

A group of zero or more hosts that, for the purposes of IP multicasting, are identified by a single class D IP destination address. Class D IP addresses have 1110 as their high-order four bits. See **IP Multicasting** for more information.

ICMP

See **Internet Control Message Protocol**.

#include *file.h*

A C language precompiler directive specifying interpolation of a named file into the file being compiled. The interpolated file is a standard header file (indicated by placing its name in angle brackets) or any other file containing C language code (indicated by placing its name in double quotation marks).

The absolute path name of header files whose names are placed in angle brackets (lt; gt;) is `/usr/include/`*`file.h`*.

International Standards Organization (ISO)

An international body composed of the national standards organizations of 89 countries. ISO issues standards on a vast number of goods and services, including networking software.

Internet Control Message Protocol (ICMP)

A host-to-host protocol from the Internet Protocol (IP) suite that provides error and informational messages on the operations of the IP.

Internet Protocol (IP)

The Internet Protocol that provides a connectionless service for the delivery of datagrams across a network.

ISO

See **International Standards Organization**.

IP Multicasting

IP Multicasting is a method for transmitting IP datagrams to a group of hosts identified by a single IP destination address, or host group. Host groups are identified by class D IP addresses. See **host group** for more information.

Management Information Base

See MIB.

MIB

The Management Information Base (MIB) definitions are a set of data elements that relate to network management. See Chapter 6.

name server

A daemon running on a system that client processes contact to obtain the addresses of hosts or other objects in a network. This daemon translates a machine's network name to its network IP address.

name service

The service provided to client processes for identifying peer systems for communications purposes.

nonblocking mode

See **asynchronous execution**.

normal data

Regular data that is sent or received in band by a transport user. See also **out-of-band data**.

Object Identifier

See OID.

OID

Object Identifiers (OID) are data elements in MIB definitions that can be referred to by name or by a corresponding sequence of numbers. See Chapter 6.

Open Systems Interconnection (OSI)

The interconnection of open systems in accordance with ISO standards.

orderly release

In XTI, an optional feature that allows a transport user to gracefully terminate a transport connection with no loss of data.

OSI

See **Open Systems Interconnection**.

out-of-band data

Data that is transmitted out of the flow of normal data because it is considered urgent. The receiving process is notified of the presence of this data so that it can be retrieved.

passive user

In an XTI transport connection, the peer transport user that responded to the connection request. See also **active user** and **client process**.

pipe

An I/O stream that has a descriptor and can be used in unidirectional communications between related processes. See also **socketpair**.

raw socket

A socket that provides privileged users access to internal network protocols and interfaces. These socket types can be used to take advantage of protocol features not available through more normal interfaces or to communicate with hardware interfaces.

Serial Line Internet Protocol (SLIP)

A transmission line protocol that encapsulates and transfers IP datagrams over asynchronous serial lines.

server process

In the client/server model of communication, a process that provides services to client processes. See also **passive user**.

SLIP

See **Serial Line Internet Protocol**.

socket

In interprocess communications, an endpoint of communication. Also, the system call that creates a socket and the associated data structure.

socketpair

A pair of sockets that can be created in the UNIX domain for 2-way communication. Like pipes, socketpairs require communicating processes to be related. See also **pipe**.

state

In XTI, the current condition of a process that reflects the function in progress. XTI uses eight different states to manage communications over a transport endpoint.

stream socket

A socket that provides 2-way byte streams across a transport connection. Also includes a mechanism for handling out-of-band data.

STREAMS

A kernel mechanism specified by AT&T that supports the implementation of device drivers and networking protocol stacks. See also **STREAMS framework**.

STREAMS framework

Components of the AT&T STREAMS mechanism which define the interface standards for character I/O within the kernel and between the kernel and user levels. It consists of functions, utility routines, kernel facilities, and data structures.

synchronous execution

A mode of execution that forces transport primitives to wait for specific events before returning control to the transport user.

TCP

See **Transmission Control Protocol**.

TCP/IP

The two fundamental protocols of the Internet Protocol suite, and an acronym that is frequently used to refer to the Internet Protocol suite. TCP provides for the reliable transfer of data, while IP transmits the data through the network in the form of datagrams. See also **Transmission Control Protocol** and **Internet Protocol**.

TLI

See **Transport Layer Interface**.

Transmission Control Protocol (TCP)

The Internet transport-layer protocol that provides a reliable, full-duplex, connection-oriented service for applications. TCP uses the IP protocol to transmit information through the network.

transport endpoint

A communication path over which a transport user can exchange data with a transport provider. See also **Transport Layer Interface**.

Transport Layer Interface (TLI)

An interface to the transport layer of the OSI model, designed on the ISO Transport service definition.

transport provider

A transport protocol that offers transport layer services in a system.

Transport Service Data Unit (TSDU)

In OSI terminology, the item of information, or message, that the transport user passes to the transport provider.

transport services

The support given by the transport layer in a system to the application layer for the transfer of data between user processes. The two types of services provided are connection-oriented and connectionless. See also **Transport Layer Interface**.

transport user

A program needing the services of a transport protocol to send data to or receive data from another program or point in a network. See also **Transport Layer Interface**.

TSDU

See **Transport Service Data Unit**.

UDP

See **User Datagram Protocol**.

User Datagram Protocol (UDP)

The Internet Protocol that allows application programs on remote machines to send datagrams to one another. UDP uses IP to deliver the datagrams.

X/Open Transport Interface

Protocol-independent, transport-layer interface for applications. XTI consists of a series of C language functions based on TLI, which in turn was based on the transport service definition for the OSI model.

XTI

See **X/Open Transport Interface**.

Index