The SCO Streams

Network Programmer's Guide

The Santa Cruz Operation, Inc.

Document Number: XG-11-1-88-1.0A

Processed Date: Mon Nov 21 16:01:00 PST 1988

Contents

Chapter 1

Introduction

1.1 Introduction

This document provides detailed information on the UNIX System Transport Layer Interface, along with associated examples. This guide is intended for programmers who require the services defined by this interface. Working knowledge of UNIX System programming and data communication concepts is assumed. In particular, working knowledge of the Reference Model of Open Systems Interconnection (OSI) is required.

1.2 Background

A discussion of the OSI Reference Model is presented here to place the Transport Layer Interface in perspective. The Reference Model partitions networking functions into seven layers, as depicted in Figure 1-1.

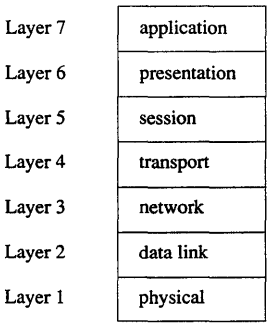

Figure 1-1 OSI Reference Model

Layer 1 The physical layer is responsible for the transmission of raw data over a communication medium.

Layer 2 The data link layer provides the exchange of data between network layer entities. It detects and corrects any errors that may occur in the physical layer transmission.

Layer 3 The network layer manages the operation of the network. In particular, it is responsible for the routing and management of data exchange between transport layer entities within the network.

Layer 4 The transport layer provides transparent data transfer services between session layer entities by relieving them from concerns with the mechanisms of data transfer.

Layer 5 The session layer provides the services needed by presentation layer entities to organize and synchronize their dialogue and manage their data exchange.

Layer 6 The presentation layer manages the representation of information which application layer entities either communicate or reference in their communication.

Layer 7 The application layer serves as the window between corresponding application processes that are exchanging information.

A basic principle of the reference model is that each layer provides services needed by the next higher layer in a way that frees the upper layer from concern about how these services are provided. This approach simplifies the design of each particular layer.

Industry standards either have been or are being defined at each layer of the reference model. Two standards are defined at each layer: one that specifies an interface to the services of the layer, and one that defines the protocol by which services are provided. A service interface standard at any layer frees users of the service from concern with the details of protocol implementation, or even the identity of the protocol used to provide the service.

The transport layer is important because it is the lowest layer in the reference model that provides the basic service of reliable, end-to-end data transfer needed by applications and higher layer protocols. In doing so, this layer hides the topology and characteristics of the underlying network from its users. More important, the transport layer defines a set of services common to layers of many contemporary protocol suites, including the International Standards Organization (ISO) protocols, the Transmission Control Protocol and Internet Protocol (TCP/IP) of the ARPANET, Xerox Network Systems (XNS), and the Systems Network Architecture (SNA).

A transport service interface enables applications and higher layer protocols to be implemented without knowledge of the underlying protocol suite. This is a principal goal of the UNIX System Transport Layer Interface. The Transport Layer Interface offers both protocol and medium independence to networking applications and higher layer protocols. because an inherent characteristic of the transport layer is that it hides

details of the physical medium being used.

The UNIX System Transport Layer Interface was modeled after the industry standard ISO Transport Service Definition (ISO 8072). As such, it is intended for those applications and protocols that require transport services. It is not intended to provide a generic networking interface for all UNIX System applications, but it is a first step in providing networking services with the UNIX System. Because the Transport Layer Interface provides reliable data transfer, and because its services are common to several protocol suites, many networking applications will find these services useful.

The Transport Layer Interface is implemented as a user library using the STREAMS input/output (I/O) mechanism. Therefore, many services available to STREAMS applications are also available to users of the Transport Layer Interface. These services are highlighted throughout this guide. The *STREAMS Primer* and *STREAMS Programmer's Guide* contain more detailed information on STREAMS for the interested reader.

1.3 Document Organization

This guide is organized as follows:

- Chapter 1, "Introduction," this introduction.
- Chapter 2, "Overview of the Transport Layer Interface," summarizes the basic set of services available to Transport Layer Interface users and presents the background information needed for the remainder of the guide.
- Chapter 3, "Connection-Mode Service," describes the services associated with connection-based (or virtual circuit) communication.
- Chapter 4, "Connectionless-Mode Service," describes the services associated with connectionless (or datagram) communication.
- Chapter 5, "A Read/Write Interface," describes how users can use the services of **read**(S) and **write**(S) to communicate over transport connection.
- Chapter 6, "Advanced Topics," discusses important concepts that are not covered in earlier chapters. These include asynchronous event handling and processing of multiple, simultaneous connect requests.

- Appendix A, "State Transitions," defines the allowable state transitions associated with the Transport Layer Interface.
- Appendix B, "Guidelines for Protocol Independence," establishes necessary guidelines for developing software that can run without change over any transport protocol developed for the Transport Layer Interface.
- Appendix C, "Examples," presents the full listing of each programming example used throughout the guide.
- Appendix D, "NSL Manual Pages," contains the NSL manual pages, a complete description of each Transport Layer Interface routine.

This guide describes the more important and common facilities of the Transport Layer Interface and is not meant to be exhaustive.

Acronyms

The following acronyms are used throughout this guide:

CLTS	Connectionless Transport Service
COTS	Connection Oriented Transport Service
ETSDU	Expedited Transport Service Data Unit
TSDU	Transport Service Data Unit

1.4 Terms You Should Know

The following terms apply to the Transport Layer Interface:

ABORTIVE RELEASE

An abrupt termination of a transport connection, which may result in the loss of data.

ASYNCHRONOUS EXECUTION

The mode of execution in which Transport Layer Interface routines will never block while waiting for specific asynchronous events to occur, but instead will return immediately if the event is not pending.

CLIENT

The transport user in connection-mode that initiates the establishment of a transport connection.

CONNECTION ESTABLISHMENT

The phase in connection-mode that enables two transport users to create a transport connection between them.

CONNECTION-MODE

A circuit-oriented mode of transfer in which data is passed from one user to another over an established connection in a reliable, sequenced manner.

CONNECTIONLESS-MODE

A mode of transfer in which data is passed from one user to another in self-contained units with no logical relationship required among multiple units.

CONNECTION RELEASE

The phase in connection-mode that terminates a previously established transport connection between two users.

DATAGRAM

A unit of data transferred between two users of the connectionless-mode service.

DATA TRANSFER

The phase in connection-mode or connectionless-mode that supports the transfer of data between two transport users.

EXPEDITED DATA

Data that is considered urgent. The specific semantics of *expedited data* are defined by the transport protocol that provides the transport service.

EXPEDITED TRANSPORT SERVICE DATA UNIT

The amount of expedited user data, the identity of which is preserved from one end of a transport connection to the other (that is, an expedited

message).

LOCAL MANAGEMENT

The phase in either connection-mode or connectionless-mode in which a transport user establishes a transport endpoint and binds a transport address to the endpoint. Functions in this phase perform local operations and require no transport layer traffic over the network.

ORDERLY RELEASE

A procedure for gracefully terminating a transport connection with no loss of data.

The user with whom a given user is communicating above the Transport Layer Interface.

SERVER

The transport user in connection-mode that offers services to other users (clients) and enables these clients to establish a transport connection to it.

SERVICE INDICATION

The notification of a pending event generated by the provider to a user of a particular service.

SERVICE PRIMITIVE

The unit of information passed across a service interface that contains either a service request or service indication.

SERVICE REQUEST

A request for some action generated by a user to the provider of a particular service.

SYNCHRONOUS EXECUTION

The mode of execution in which Transport Layer Interface routines may block while waiting for specific asynchronous events to occur.

TRANSPORT ADDRESS

The identifier used to differentiate and locate specific transport endpoints in a network.

TRANSPORT CONNECTION

The communication circuit that is established between two transport users in connection-mode.

TRANSPORT ENDPOINT

The local communication channel between a transport user and a transport provider.

TRANSPORT INTERFACE

The library routines and state transition rules that support the services of a transport protocol.

TRANSPORT PROVIDER

The transport protocol that provides the services of the Transport Layer Interface.

TRANSPORT SERVICE DATA UNIT

The amount of user data whose identity is preserved from one end of a transport connection to the other (that is, a message).

TRANSPORT USER

The user-level application or protocol that accesses the services of the Transport Layer Interface.

VIRTUAL CIRCUIT

A transport connection established in connection-mode.

Chapter 2

Overview of the Transport Layer Interface

2.1 Introduction

This chapter presents a high-level overview of the services of the Transport Layer Interface, which supports the transfer of data between two user processes. Figure 2-1 illustrates the Transport Layer Interface.

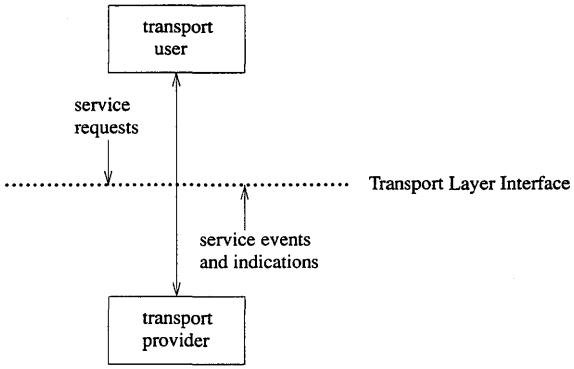

Figure 2-1 Transport Layer Interface

The transport provider is the entity that provides the services of the Transport Layer Interface, and the transport user is the entity that requires these services. An example of a transport provider is the ISO transport protocol, while a transport user may be a networking application or session layer protocol.

The transport user accesses the services of the transport provider by issuing the appropriate service requests. One example is a request to transfer data over a connection. Similarly, the transport provider notifies the user of various events, such as the arrival of data on a connection.

The Network Services Library of the UNIX System includes a set of functions that support the services of the Transport Layer Interface for user processes [see *intro*(NSL)]. These functions enable a user to initiate requests to the provider and process incoming events. Programs using the Transport Layer Interface can link the appropriate routines as follows:

```
cc prog.c -lnsl_s
```

2.2 Modes of Service

Two modes of service, connection-mode and connectionless-mode, are provided by the Transport Layer Interface. Connection-mode is circuit-oriented and enables data to be transmitted over an established connection in a reliable, sequenced manner. It also provides an identification mechanism that avoids the overhead of address resolution and transmission during the data transfer phase. This service is attractive for applications that require relatively long-lived, datastream-oriented interactions.

Connectionless-mode, in contrast, is message-oriented and supports data transfer in self-contained units with no logical relationship required among multiple units. This service requires only a preexisting association between the peer users involved. This determines the characteristics of the data to be transmitted. All the information required to deliver a unit of data is presented to the transport provider, together with the data to be transmitted, in one service access (which need not relate to any other service access). Each unit of data transmitted is entirely self-contained. Connectionless-mode service is attractive for applications which:

- involve short-term request/response interactions
- exhibit a high level of redundancy
- are dynamically reconfigurable
- do not require guaranteed in-sequence delivery of data

2.2.1 Connection-Mode Service

The connection-mode transport service is characterized by four phases: local management, connection establishment, data transfer, and connection release.

Local Management

The local management phase defines local operations between a transport user and a transport provider. For example, a user must establish a channel of communication with the transport provider, as illustrated in Figure 2-2. Each channel between a transport user and transport provider is a unique endpoint of communication, called the transport endpoint. The **t_open** routine enables a user to choose a particular transport provider that will supply the connection-mode services. This routine also

establishes the transport endpoint.

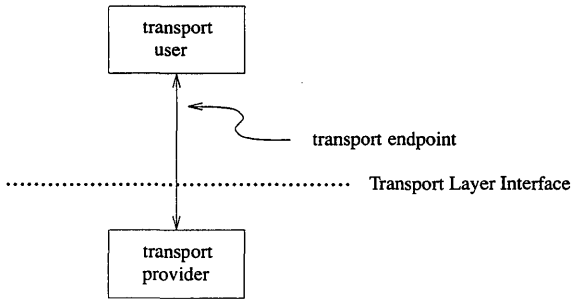

Figure 2-2 Channel Between User and Provider

Another necessary local function for each user is to establish an identity with the transport provider. Each user is identified by a transport address. More accurately, a transport address is associated with each transport endpoint, and one user process can manage several transport endpoints. In connection-mode service, one user requests a connection to another user by specifying that user's address. The structure of a transport address is defined by the address space of the transport provider. An address can be as simple as a random character string (for example, "file_server"), or as complex as an encoded bit pattern that specifies all information needed to route data through a network. Each transport provider defines its own mechanism for identifying users. Addresses can be assigned to each transport endpoint by **t_bind**.

In addition to **t_open** and **t_bind**, several routines are available to support local operations. Figure 2-3 summarizes all local management routines of the Transport Layer Interface.

Command	Description
t_alloc	Allocates Transport Layer Interface data structures [see *t_alloc*(NSL)].
t_bind	Binds a transport address to a transport endpoint [see *t_bind*(NSL)].
t_close	Closes a transport endpoint [see *t_close*(NSL)].
t_error	Prints a Transport Layer Interface error message [see *t_error*(NSL)].
t_free	Frees structures allocated using **t_alloc** [see *t_free*(NSL)].
t_getinfo	Returns a set of parameters associated with a particular transport provider [see *t_getinfo*(NSL)].
t_getstate	Returns the state of a transport endpoint [see *t_getstate*(NSL)].
t_look	Returns the current event on a transport endpoint [see *t_look*(NSL)].
t_open	Establishes a transport endpoint connected to a chosen transport provider [see *t_open*(NSL)].
t_optmgmt	Negotiates protocol-specific options with the transport provider [see *t_optmgmt*(NSL)].
t_sync	Synchronizes a transport endpoint with the transport provider [see *t_sync*(NSL)].
t_unbind	Unbinds a transport address from a transport endpoint [see *t_unbind*(NSL)].

Figure 2-3 Local Management Routines

Connection Establishment

The connection establishment phase enables two users to create a connection (virtual circuit) between them, as demonstrated in Figure 2-4.

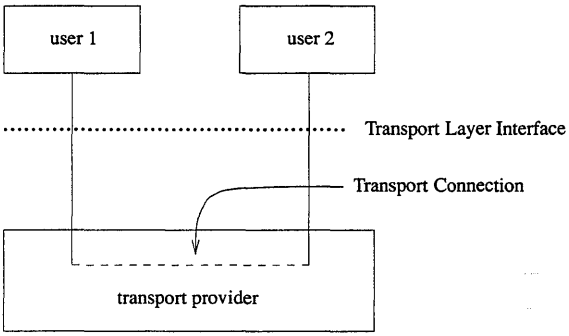

Figure 2-4 Transport Connection

This phase is illustrated by a client/server relationship between two transport users. One user, the server, typically advertises some service to a group of users and then listens for requests from those users. As each client requires the service, it attempts to connect itself to the server using the server's advertised transport address. The **t_connect** routine initiates the connect request. One argument to **t_connect**, the transport address, identifies the server the client wishes to access. The server is notified of each incoming request using **t_listen**, and can call **t_accept** to accept the client's request for access to the service. If the request is accepted, the transport connection is established.

Figure 2-5 summarizes all routines available for establishing a transport connection.

Command	Description
t_accept	Accepts a request for a transport connection [see *t_accept*(NSL)].
t_connect	Establishes a connection with the transport user at a specified destination [see *t_connect*(NSL)].
t_listen	Retrieves an indication of a connect request from another transport user [see *t_listen*(NSL)].
t_rcvconnect	Completes connection establishment if **t_connect** was called in asynchronous mode (see Chapter 6) [see *t_rcvconnect*(NSL)].

Figure 2-5 Connection Establishment Routines

Data Transfer

The data transfer phase enables users to transfer data in both directions over an established connection. Two routines, **t_snd** and **t_rcv**, send and receive data over this connection. All data sent by a user is guaranteed to be delivered to the user on the other end of the connection in the order in which it was sent. Figure 2-6 summarizes the connection-mode data transfer routines.

Command	Description
t_rcv	Retrieves data that has arrived over a transport connection [see *t_rcv*(NSL)].
t_snd	Sends data over an established transport connection [see *t_snd*(NSL)].

Figure 2-6 Connection-Mode Data Transfer Routines

Connection Release

The connection release phase provides a mechanism for breaking an established connection. When you decide that the conversation should terminate, you can request that the provider release the transport connection. Two types of connection release are supported by the Transport Layer Interface. The first is an abortive release, which directs the transport provider to release the connection immediately. Any previously sent data that has not yet reached the other transport user may be discarded by the transport provider. The **t_snddis** routine initiates this abortive disconnect, and **t_rcvdis** processes the incoming indication of an abortive disconnect.

All transport providers must support the abortive release procedure. In addition, some transport providers may also support an orderly release facility that enables users to terminate communication gracefully with no data loss. The functions **t_sndrel** and **t_rcvrel** support this capability. Figure 2-7 summarizes the connection release routines.

Command	Description
t_rcvdis	Returns an indication of an aborted connection, including a reason code and user data [see *t_rcvdis*(NSL)].
t_rcvrel	Returns an indication that the remote user has requested an orderly release of a connection [see *t_rcvrel*(NSL)].
t_snddis	Aborts a connection or rejects a connect request [see *t_snddis*(NSL)].
t_sndrel	Requests the orderly release of a connection [see *t_sndrel*(NSL)].

Figure 2-7 Connection Release Routines

2.2.2 Connectionless-Mode Service

The connectionless-mode transport service is characterized by two phases: local management and data transfer. The local management phase defines the same local operations described above for the connection-mode service.

The data transfer phase enables a user to transfer data units (sometimes called datagrams) to the specified peer user. Each data unit must be accompanied by the transport address of the destination user. Two routines, **t_sndudata** and **t_rcvudata**, support this message-based data transfer facility. Figure 2-8 summarizes all routines associated with connectionless-mode data transfer.

Command	Description
t_rcvudata	Retrieves a message sent by another transport user [see *t_rcvudata*(NSL)].
t_rcvuderr	Retrieves error information associated with a previously sent message [see *t_rcvuderr*(NSL)].
t_sndudata	Sends a message to the specified destination user [see *t_sndudata*(NSL)].

Figure 2-8 Connectionless-Mode Data Transfer Routines

2.3 State Transitions

The Transport Layer Interface has two components:

- the library routines that provide the transport services to users
- the state transition rules that define the sequence in which the transport routines can be invoked

The state transition rules are presented in Appendix A of this guide in the form of state tables. The state tables define the legal sequence of library calls based on state information and the handling of events. These events include user-generated library calls as well as provider-generated event indications.

Note

Any user of the Transport Layer Interface must completely understand all possible state transitions before writing software using the interface.

Chapter 3

Connection-Mode Service

3.1 Introduction

This chapter describes the connection-mode service of the Transport Layer Interface. As discussed in the previous chapter, the connection-mode service can be illustrated using a client/server example. The important concepts of connection-mode service are presented using two programming examples. The first example illustrates how a client establishes a connection to a server and then communicates with the server. The second example shows the server's side of the interaction. All examples discussed in this guide are presented in their entirety in Appendix C.

In the examples the client establishes a connection with a server process. The server then transfers a file to the client. The client receives the data from the server and writes it to its standard output file.

3.2 Local Management

Before the client and server can establish a transport connection, each must first establish a local channel (the transport endpoint) to the transport provider using **t_open**, and establish its identity (or address) using **t_bind**.

The set of services supported by the Transport Layer Interface may not be implemented by all transport protocols. Each transport provider has a set of characteristics associated with it that determine the services it offers and the limitations associated with those services. This information is returned to the user by **t_open**, and consists of the following:

addr	maximum size of a transport address
options	maximum bytes of protocol-specific options that can be passed between the transport user and transport provider
tsdu	maximum message size that can be transmitted in either connection-mode or connectionless-mode
etsdu	maximum expedited data message size that can be sent over a transport connection
connect	maximum number of bytes of user data that can be passed between users during connection establishment

discon	maximum bytes of user data that can be passed between users during the abortive release of a connection
servtype	the type of service supported by the transport provider

The three service types defined by the Transport Layer Interface are:

T_COTS	The transport provider supports connection-mode service but does not provide the optional orderly release facility.
T_COTS_ORD	The transport provider supports connection-mode service with the optional orderly release facility.
T_CLTS	The transport provider supports connectionless-mode service.

Only one such service can be associated with the transport provider identified by **t_open**.

Note

t_open returns the default provider characteristics associated with a transport endpoint. However, some characteristics can change after an endpoint has been opened. This occurs if the characteristics are associated with negotiated options (described later in this chapter). For example, if the support of expedited data transfer is a negotiated option, the value of this characteristic can change. **t_getinfo** can be called to retrieve the current characteristics of a transport endpoint.

Once a user establishes a transport endpoint with the chosen transport provider, it must establish its identity. As mentioned earlier, **t_bind** accomplishes this by binding a transport address to the transport endpoint. In addition, for servers, this routine informs the transport provider that the endpoint will be used to listen for incoming connect requests, also called connect indications.

An optional facility, **t_optmgmt** [see *t_optmgmt*(NSL)], is also available during the local management phase. It enables a user to negotiate the values of protocol options with the transport provider. Each transport protocol is expected to define its own set of negotiable protocol options,

which can include such information as Quality-of-Service parameters. Because of the protocol-specific nature of options, only applications written for a particular protocol environment are expected to use this facility.

3.2.1 The Client

The local management requirements of the example client and server are used to discuss details of these facilities. The following are the definitions needed by the client program, followed by its necessary local management steps.

```
#include <stdio.h>
#include <tiuser.h>
#include <fcntl.h>

#define SRV_ADDR   1   /* server's well known address */

main()
{
   int fd;
   int nbytes;
   int flags = 0;
   char buf[1024];
   struct t_call *sndcall;
   extern int t_errno;

   if ((fd = t_open("/dev/tivc", O_RDWR, NULL)) < 0) {
       t_error("t_open failed");
       exit(1);
   }

   if (t_bind(fd, NULL, NULL) < 0) {
       t_error("t_bind failed");
       exit(2);
   }
```

The first argument to **t_open** is the path name of a file system node that identifies the transport protocol that will supply the transport service. In this example, */dev/tivc* is a STREAMS **clone** device node that identifies a generic, connection-based transport protocol [see *clone*(STR)]. The **clone** device finds an available minor device of the transport provider for the user. It is opened for both reading and writing, as specified by the O_RDWR open flag. The third argument can be used to return the service characteristics of the transport provider to the user. This information is useful when writing protocol-independent software (discussed in Appendix B). For simplicity, the client and server in this example ignore this information and assume that the transport provider has the following characteristics:

- The transport address is an integer value which uniquely identifies each user.
- The transport provider supports the T_COTS_ORD service type, and the example will use the orderly release facility to release the connection.
- User data cannot be passed between users during either connection establishment or abortive release.
- The transport provider does not support protocol-specific options.

Because these characteristics are not needed by the user, NULL is specified in the third argument to **t_open**. If the user needs a service other than T_COTS_ORD, another transport provider will be opened. An example of the T_CLTS service invocation is presented in Chapter 4.

The return value of **t_open** is an identifier for the transport endpoint that is used by all subsequent Transport Layer Interface function calls. This identifier is actually a file descriptor obtained by opening the transport protocol file [see *open*(S)]. The significance of this fact is highlighted in Chapter 5.

After the transport endpoint is created, the client calls **t_bind** to assign an address to the endpoint. The first argument identifies the transport endpoint. The second argument describes the address the user would like to bind to the endpoint, and the third argument is set on return from **t_bind** to specify the address that the provider bound.

The address associated with a server's transport endpoint is important because that is the address used by all clients to access the server. However, the typical client does not care what its own address is, because no other process will try to access it. That is the case in this example, where the second and third arguments to **t_bind** are set to NULL. A NULL second argument directs the transport provider to choose an address for the user. A NULL third argument indicates that the user does not care what address was assigned to the endpoint.

If either **t_open** or **t_bind** fails, the program calls **t_error** [see *t_error*(NSL)] to print an appropriate error message to **stderr**. If any Transport Layer Interface routine fails, the global integer **t_errno** is assigned an appropriate transport error value. A set of such error values is defined in **<tiuser.h>** for the Transport Layer Interface, and **t_error**

prints an error message corresponding to the value in **t_errno**. This routine is analogous to **perror**(S), which prints an error message based on the value of **errno**. If the error associated with a transport function is a system error, **t_errno** is set to TSYSERR, and **errno** is set to the appropriate value.

3.2.2 The Server

The server in this example must take similar local management steps before communication can begin. The server must establish a transport endpoint through which it listens for connect indications. The necessary definitions and local management steps are shown below:

```
#include <tiuser.h>
#include <stropts.h>
#include <fcntl.h>
#include <stdio.h>
#include <signal.h>

#define DISCONNECT -1
#define SRV_ADDR    1  /* server's well-known address */

int conn_fd;           /* connection established here */
extern int t_errno;

main()
{

   int listen_fd;      /* listening transport endpoint */
   struct t_bind *bind;
   struct t_call *call;

   if ((listen_fd = t_open("/dev/tivc", O_RDWR, NULL)) < 0) {
       t_error("t_open failed for listen_fd");
       exit(1);
   }

   /*
    * By assuming that the address is an integer value,
    * this program may not run over another protocol.
    */

   if ((bind = (struct t_bind *)t_alloc(listen_fd, T_BIND, T_ALL)) == NULL) {
       t_error("t_alloc of t_bind structure failed");
       exit(2);
   }
```

```
bind->qlen = 1;
bind->addr.len = sizeof(int);
*(int *)bind->addr.buf = SRV_ADDR;

if (t_bind(listen_fd, bind, bind) < 0) {
    t_error("t_bind failed for listen_fd");
    exit(3);
}

/*
 * Was the correct address bound?
 */
if (*(int *)bind->addr.buf != SRV_ADDR) {
    fprintf(stderr, "t_bind bound wrong address\n");
    exit(4);
}
```

As with the client, the first step is to call **t_open** to establish a transport endpoint with the desired transport provider. This endpoint, *listen_fd*, will be used to listen for connect indications. Next, the server must bind its well-known address to the endpoint. This address is used by each client to access the server. The second argument to **t_bind** requests that a particular address be bound to the transport endpoint. This argument points to a **t_bind** structure with the following format:

```
struct t_bind {
   struct netbuf addr;
   unsigned qlen;
}
```

where *addr* describes the address to be bound, and *qlen* indicates the maximum outstanding connect indications that can arrive at this endpoint. All Transport Layer Interface structure and constant definitions are found in **<tiuser.h>**.

The address is specified using a **netbuf** structure that contains the following members:

```
struct netbuf {
   unsigned int maxlen;
   unsigned int len;
   char *buf;
}
```

where *buf* points to a buffer containing the data, *len* specifies the bytes of data in the buffer, and *maxlen* indicates the maximum number of bytes that the buffer can hold. (This need be set only when data is returned to

the user by a Transport Layer Interface routine.) For the **t_bind** structure, the data pointed to by *buf* identifies a transport address. It is expected that the structure of addresses will vary among each protocol implementation under the Transport Layer Interface. The **netbuf** structure is intended to support any such structure.

If the value of *qlen* is greater than 0, the transport endpoint can be used to listen for connect indications. In such cases, **t_bind** directs the transport provider to immediately begin queueing connect indications destined for the bound address. Furthermore, the value of *qlen* indicates the maximum outstanding connect indications the server wishes to process. The server must respond to each connect indication, either accepting or rejecting the request for connection. An outstanding connect indication is one to which the server has not yet responded. Often, a server will fully process a single connect indication and respond to it before receiving the next indication. In this case, a value of 1 is appropriate for *qlen*. However, some servers may wish to retrieve several connect indications before responding to any of them. In such cases, *qlen* indicates the maximum number of such outstanding indications the server will process. An example of a server that manages multiple outstanding connect indications is presented in Chapter 6.

t_alloc is called to allocate the **t_bind** structure needed by **t_bind**. **t_alloc** takes three arguments. The first is a file descriptor that references a transport endpoint. This is used to access the characteristics of the transport provider [see *t_open*(NSL)]. The second argument identifies the appropriate Transport Layer Interface structure to be allocated. The third argument specifies which, if any, **netbuf** buffers should be allocated for that structure. T_ALL specifies that all **netbuf** buffers associated with the structure should be allocated; in this example, T_ALL causes the *addr* buffer to be allocated. The size of this buffer is determined from the transport provider characteristic that defines the maximum address size. The *maxlen* field of this **netbuf** structure is set to the size of the newly allocated buffer by **t_alloc**. The use of **t_alloc** helps to ensure the compatibility of user programs with future releases of the Transport Layer Interface.

The server in this example processes connect indications one at a time, and so *qlen* is set to one. The address information is then assigned to the newly allocated **t_bind** structure. This **t_bind** structure is used to pass information to **t_bind** in the second argument, and it is also used to return information to the user in the third argument.

On return, the **t_bind** structure contains the address that was bound to the transport endpoint. If the provider cannot bind the requested address (perhaps because it was bound to another transport endpoint), it chooses another appropriate address.

Note

Each transport provider manages its address space differently. Some transport providers allow a single transport address to be bound to several transport endpoints, while others require a unique address per endpoint. The Transport Layer Interface supports either choice. Based on its address management rules, a provider determines if it can bind the requested address. If not, it chooses another valid address from its address space and binds it to the transport endpoint.

The server must check the bound address to ensure that it is the one previously advertised to clients. Otherwise, the clients will be unable to reach the server.

If **t_bind** succeeds, the provider begins queueing connect indications. This begins the next phase of communication, connection establishment.

3.3 Connection Establishment

The connection establishment procedures highlight the distinction between clients and servers. The Transport Layer Interface imposes a different set of procedures in this phase for each type of transport user. The client initiates the connection establishment procedure by using **t_connect** to request a connection to a particular server. The server is then notified of the client's request by calling **t_listen**. The server can either accept or reject the client's request. It calls **t_accept** to establish the connection or calls **t_snddis** to reject the request. The client is notified of the server's decision when **t_connect** completes.

The Transport Layer Interface supports two facilities during connection establishment that may not be supported by all transport providers. The first is the ability to transfer data between the client and server when establishing the connection. The client can send data to the server when it requests a connection. This data is passed to the server by **t_listen**. Similarly, the server can send data to the client when it accepts or rejects the connection. The connect characteristic returned by **t_open** determines how much data, if any, two users can transfer during connection establishment.

The second optional service supported by the Transport Layer Interface during connection establishment is the negotiation of protocol options. The client can specify protocol options that it would like the remote user

and/or transport provider to use. The Transport Layer Interface supports both local and remote option negotiation. As discussed earlier, option negotiation is inherently a protocol-specific function. Use of this facility is discouraged if protocol-independent software is a goal (see Appendix B).

3.3.1 The Client

Continuing with the client/server example, the steps needed by the client to establish a connection are as follows:

```
/*
 * By assuming that the address is an integer value,
 * this program may not run over another protocol.
 */
if ((sndcall = (struct t_call *)t_alloc(fd, T_CALL, T_ADDR)) == NULL) {
    t_error("t_alloc failed");
    exit(3);
}
sndcall->addr.len = sizeof(int);
*(int *)sndcall->addr.buf = SRV_ADDR;

if (t_connect(fd, sndcall, NULL) < 0) {
    t_error("t_connect failed for fd");
    exit(4);
}
```

The **t_connect** call establishes the connection with the server. The first argument to **t_connect** identifies the transport endpoint through which the connection is established, and the second argument identifies the destination server. This argument is a pointer to a **t_call** structure, which has the following format:

```
struct t_call {
   struct netbuf addr;
   struct netbuf opt;
   struct netbuf udata;
   int sequence;
}
```

addr identifies the address of the server, *opt* can be used to specify protocol-specific options that the client would like to associate with the connection, and *udata* identifies user data that can be sent with the connect request to the server. The *sequence* field has no meaning for **t_connect**.

t_alloc is called above to allocate the **t_call** structure dynamically. Once allocated, the appropriate values are assigned. In this example, no options or user data are associated with the **t_connect** call, but the server's address must be set. The third argument to **t_alloc** is set to T_ADDR to indicate that an appropriate **netbuf** buffer should be allocated for the address. The server's address is then assigned to *buf*, and *len* is set accordingly.

The third argument to **t_connect** can be used to return information about the newly established connection to the user, and it can retrieve any user data sent by the server in its response to the connect request. It is set to NULL by the client here to indicate that this information is not needed. The connection is established on successful return of **t_connect.** If the server rejects the connect request, **t_connect** fails and sets **t_errno** to TLOOK.

3.3.2 Event Handling

The TLOOK error has special significance in the Transport Layer Interface. Some Transport Layer Interface routines may be interrupted by an unexpected asynchronous transport event on the given transport endpoint, and TLOOK notifies the user that an event has occurred. As such, TLOOK does not indicate an error with a Transport Layer Interface routine. Rather, it indicates that the normal processing of that routine will not be performed because of the pending event. The events defined by the Transport Layer Interface are listed here:

T_LISTEN	A request for a connection, called a connect indication, has arrived at the transport endpoint.
T_CONNECT	The confirmation of a previously sent connect request, called a connect confirmation, has arrived at the transport endpoint. The confirmation is generated when a server accepts a connect request.
T_DATA	User data has arrived at the transport endpoint.
T_EXDATA	Expedited user data has arrived at the transport endpoint. (Expedited data will be discussed later in this chapter.)
T_DISCONNECT	A notification that the connection was aborted or that the server rejected a connect request, called a disconnect indication, has arrived at the

transport endpoint.

T_ORDREL — A request for the orderly release of a connection, called an orderly release indication, has arrived at the transport endpoint.

T_UDERR — The notification of an error in a previously sent datagram, called a unitdata error indication, has arrived at the transport endpoint (see Chapter 4).

As described in the state tables of Appendix A, in some states it is possible to receive one of several asynchronous events. The **t_look** routine [see *t_look*(NSL)] enables a user to determine what event has occurred if a TLOOK error is returned. The user can then process that event accordingly. In the example, if a connect request is rejected, the event passed to the client will be a disconnect indication. The client will exit if its request is rejected.

3.3.3 The Server

Returning to the example, when the client calls **t_connect**, a connect indication is generated on the server's listening transport endpoint. The steps required by the server to process the event are presented below. For each client, the server accepts the connect request and spawns a server process to manage the connection.

```
    if ((call = (struct t_call *)t_alloc(listen_fd, T_CALL, T_ALL)) == NULL) {
        t_error("t_alloc of t_call structure failed");
        exit(5);
    }
    while (1) {
        if (t_listen(listen_fd, call) < 0) {
            t_error("t_listen failed for listen_fd");
            exit(6);
        }
        if ((conn_fd = accept_call(listen_fd, call)) != DISCONNECT)
            run_server(listen_fd);
    }
}
```

The server loops forever, processing each connect indication. First the server calls **t_listen** to retrieve the next connect indication. When one arrives, the server calls **accept_call** to accept the connect request. **accept_call** accepts the connection on an alternate transport endpoint (as discussed below) and returns the value of that endpoint. *conn_fd* is a

global variable that identifies the transport endpoint where the connection is established. Because the connection is accepted on an alternate endpoint, the server can continue listening for connect indications on the endpoint that was bound for listening. If the call is accepted without error, **run_server** spawns a process to manage the connection.

The server allocates a **t_call** structure to be used by **t_listen**. The third argument to **t_alloc**, T_ALL, specifies that all necessary buffers should be allocated for retrieving the caller's address, options, and user data. As mentioned earlier, the transport provider in this example does not support the transfer of user data during connection establishment, and also does not support any protocol options. Therefore, **t_alloc** does not allocate buffers for the user data and options. However, it must allocate a buffer large enough to store the address of the caller. **t_alloc** determines the buffer size from the *addr* characteristic returned by **t_open**. The *maxlen* field of each **netbuf** structure is set to the size of the newly allocated buffer by **t_alloc**. (*maxlen* is 0 for the user data and options buffers.)

Using the **t_call** structure, the server calls **t_listen** to retrieve the next connect indication. If one is currently available, it is returned to the server immediately. Otherwise, **t_listen** blocks until a connect indication arrives.

Note

The Transport Layer Interface supports an asynchronous mode that prevents a process from blocking. This feature is discussed in Chapter 6.

When a connect indication arrives, the server calls **accept_call** to accept the client's request, as follows:

```
accept_call(listen_fd, call)
int listen_fd;
struct t_call *call;
{
    int resfd;

    if ((resfd = t_open("/dev/tivc", O_RDWR, NULL)) < 0) {
        t_error("t_open for responding fd failed");
        exit(7);
    }

    if (t_bind(resfd, NULL, NULL) < 0) {
        t_error("t_bind for responding fd failed");
        exit(8);
    }
```

```
    if (t_accept(listen_fd, resfd, call) < 0) {
        if (t_errno == TLOOK) {  /* must be a disconnect */
            if (t_rcvdis(listen_fd, NULL) < 0) {
                t_error("t_rcvdis failed for listen_fd");
                exit(9);
            }
            if (t_close(resfd) < 0) {
                t_error("t_close failed for responding fd");
                exit(10);
            }
            /* go back up and listen for other calls */
            return(DISCONNECT);
        }
        t_error("t_accept failed");
        exit(11);
    }
    return(resfd);
}
```

accept_call takes two arguments. *listen_fd* identifies the transport endpoint where the connect indication arrived, and *call* is a pointer to a **t_call** structure that contains all information associated with the connect indication. The server first establishes another transport endpoint by opening the clone device node of the transport provider and binding an address. As with the client, a NULL value is passed to **t_bind** to specify that the user does not care what address is bound by the provider. The newly established transport endpoint, *resfd*, is used to accept the client's connect request.

The first two arguments of **t_accept** specify the listening transport endpoint and the endpoint where the connection will be accepted, respectively. A connection can be accepted on the listening endpoint. However, this prevents other clients from accessing the server for the duration of that connection.

The third argument of **t_accept** points to the **t_call** structure associated with the connect indication. This structure should contain the address of the calling user and the sequence number returned by **t_listen**. The value of *sequence* has particular significance if the server manages multiple outstanding connect indications. Chapter 6 presents such an example. Also, the **t_call** structure should identify protocol options the user would like to specify and user data that can be passed to the client. Because the transport provider in this example does not support protocol options or the transfer of user data during connection establishment, the **t_call** structure returned by **t_listen** can be passed without change to **t_accept**.

For simplicity in the example, the server exits if either the **t_open** or **t_bind** call fails. **exit**(S) closes the transport endpoint associated with *listen_fd*, causing the transport provider to pass a disconnect indication to the client that requested the connection. This disconnect indication notifies the client that the connection was not established; **t_connect** then fails, setting **t_errno** to TLOOK.

t_accept can fail if an asynchronous event has occurred on the listening transport endpoint before the connection is accepted. In this case, **t_errno** is set to TLOOK. The state transition table in Appendix A shows that the only event that can occur in this state with only one outstanding connect indication is a disconnect indication. This event can occur if the client decides to undo the connect request it had previously initiated. If a disconnect indication arrives, the server must retrieve the disconnect indication using **t_rcvdis**. This routine takes a pointer to a **t_discon** structure as an argument which is used to retrieve information associated with a disconnect indication. However, in this example the server does not care to retrieve this information, and so it sets the argument to NULL. After receiving the disconnect indication, **accept_call** closes the responding transport endpoint and returns DISCONNECT, which informs the server that the connection was disconnected by the client. The server then listens for further connect indications.

Figure 3-1 illustrates how the server establishes connections.

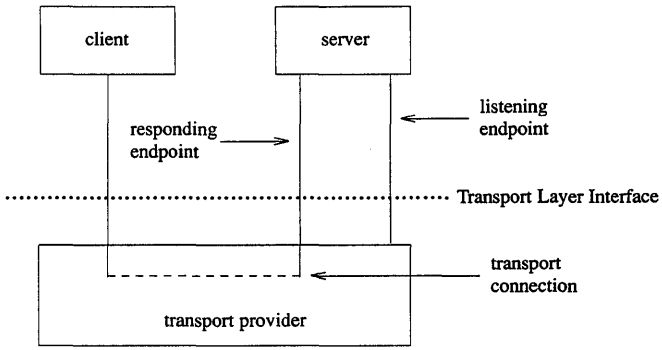

Figure 3-1 Listening and Responding Transport Endpoints

The transport connection is established on the newly created responding endpoint, and the listening endpoint is freed to retrieve further connect indications.

3.4 Data Transfer

Once the connection has been established, both the client and server can begin transferring data over the connection using **t_snd** and **t_rcv**. From this point on, the Transport Layer Interface does not differentiate the client from the server. Either user can send and receive data, or release the connection. The Transport Layer Interface guarantees reliable, sequenced delivery of data over an existing connection.

Two classes of data can be transferred over a transport connection: normal and expedited. Expedited data is typically associated with information of an urgent nature. The exact semantics of expedited data are subject to the interpretations of the transport provider. Furthermore, not all transport protocols support the notion of an expedited data class [see *t_open*(NSL)].

All transport protocols support the transfer of data in byte stream mode, where "byte stream" implies no concept of message boundaries on data which is transferred over a connection. However, some transport protocols support the preservation of message boundaries over a transport connection. This service is supported by the Transport Layer Interface, but protocol-independent software must not rely on its existence.

The message interface for data transfer is supported by a special flag of **t_snd** and **t_rcv** called T_MORE. The messages, called Transport Service

Data Units (TSDU), can be transferred between two transport users as distinct units. The maximum size of a TSDU is a characteristic of the underlying transport protocol. This information is available to the user from **t_open** and **t_getinfo**. Because the maximum TSDU size can be large (possibly unlimited), the Transport Layer Interface enables a user to transmit a message in multiple units.

To send a message in multiple units over a transport connection, the user must set the T_MORE flag on every **t_snd** call except the last. This flag indicates that the user will send more data associated with the message in a subsequent call to **t_snd**. The last message unit should be transmitted with T_MORE turned off to indicate that this is the end of the TSDU.

Similarly, a TSDU can be passed to the user on the receiving side in multiple units. Again, if **t_rcv** returns with the T_MORE flag set, the user should continue calling **t_rcv** to retrieve the remainder of the message. The last unit in the message is indicated by a call to **t_rcv** that does not set T_MORE.

Warning

The T_MORE flag implies nothing about how the data is packaged below the Transport Layer Interface. Furthermore, it implies nothing about how the data is delivered to the remote user. Each transport protocol, and each implementation of that protocol, can package and deliver the data differently.

For example, if a user sends a complete message in a single call to **t_snd**, there is no guarantee that the transport provider will deliver the data in a single unit to the remote transport user. Similarly, a TSDU transmitted in two message units may be delivered in a single unit to the remote transport user. The message boundaries can be preserved only by noting the value of the T_MORE flag on **t_snd** and **t_rcv**. This guarantees that the receiving user will see a message with the same contents and message boundaries as was sent by the remote user.

3.4.1 The Client

Continuing with the client/server example, the server now transfers a log file to the client over the transport connection. The client receives this data and writes it to its standard output file. The client and server use a byte stream interface, where message boundaries (that is, the T_MORE flag) are ignored. The client receives data using the following instructions:

```
while ((nbytes = t_rcv(fd, buf, 1024, &flags)) != -1) {
    if (fwrite(buf, 1, nbytes, stdout) < 0) {
        fprintf(stderr, "fwrite failed\n");
        exit(5);
    }
}
```

The client continuously calls **t_rcv** to process incoming data. If no data is currently available, **t_rcv** blocks until data arrives. **t_rcv** retrieves the available data up to 1024 bytes, which is the size of the client's input buffer, and returns the number of bytes that were received. The client then writes this data to standard output and continues. The data transfer phase completes when **t_rcv** fails. **t_rcv** fails if an orderly release indication or disconnect indication arrives, as discussed later in this chapter. If the **fwrite** call [see *fread*(S)] fails for any reason, the client exits, thereby closing the transport endpoint. If the transport endpoint is closed (either by **exit** or **t_close**) when it is in the data transfer phase, the connection is aborted and the remote user receives a disconnect indication.

3.4.2 The Server

Looking now at the other side of the connection, the server manages its data transfer by spawning a child process to send the data to the client. The parent process then loops back to listen for further connect indications. The server calls **run_server** to spawn this child process as follows:

```
connrelease()
{
    /* conn_fd is global because needed here */
    if (t_look(conn_fd) == T_DISCONNECT) {
        fprintf(stderr, "connection aborted\n");
        exit(12);
    }
    /* else orderly release indication - normal exit */
    exit(0);
}

run_server(listen_fd)
int listen_fd;
{
    int nbytes;
    FILE *logfp;        /* file pointer to log file */
    char buf[1024];

    switch (fork()) {

    case -1:
        perror("fork failed");
        exit(20);

    default:    /* parent */

        /* close conn_fd and then go up and listen again */
        if (t_close(conn_fd) < 0) {
            t_error("t_close failed for conn_fd");
            exit(21);
        }
        return;

    case 0:     /* child */

        /* close listen_fd and do service */
        if (t_close(listen_fd) < 0) {
            t_error("t_close failed for listen_fd");
            exit(22);
        }
        if ((logfp = fopen("logfile", "r")) == NULL) {
            perror("cannot open logfile");
            exit(23);
        }

        signal(SIGPOLL, connrelease);
        if (ioctl(conn_fd, I_SETSIG, S_INPUT) < 0) {
            perror("ioctl I_SETSIG failed");
            exit(24);
        }
        if (t_look(conn_fd) != 0) { /* was disconnect already there? */
            fprintf(stderr, "t_look returned unexpected event\n");
            exit(25);
        }

        while ((nbytes = fread(buf, 1, 1024, logfp)) > 0)
            if (t_snd(conn_fd, buf, nbytes, 0) < 0) {
                t_error("t_snd failed");
                exit(26);
            }
```

After the **fork**, the parent process returns to the main processing loop and listens for further connect indications. Meanwhile, the child process manages the newly established transport connection. If the **fork** call fails, **exit** closes the transport endpoint associated with *listen_fd*. This action causes a disconnect indication to be passed to the client, and the client's **t_connect** call fails.

The server process reads 1024 bytes of the log file at a time and sends that data to the client using **t_snd**. *buf* points to the start of the data buffer, and *nbytes* specifies the number of bytes to be transmitted. The fourth argument is used to specify optional flags. Two flags are currently supported: T_EXPEDITED can be set to indicate that the data is expedited, and T_MORE can be set to define message boundaries when transmitting messages over a connection. Neither flag is set by the server in this example.

If the user begins to flood the transport provider with data, the provider can exert back pressure to provide flow control. In such cases, **t_snd** blocks until the flow control is relieved and then resumes its operation. **t_snd** will not complete until *nbyte* bytes have been passed to the transport provider.

The **t_snd** routine does not look for a disconnect indication before passing data to the provider. Also, because the data traffic is flowing in one direction, the user will never look for incoming events. If, for some reason, the connection is aborted, the user should be notified because data may be lost. One option available to the user is to use **t_look** to check for incoming events before each **t_snd** call. A more efficient solution is the one presented in the example. The STREAMS I_SETSIG **ioctl** enables a user to request a signal when a given event occurs [see *streamio*(STR) and *signal*(S)]. The STREAMS event of concern here is S_INPUT, which causes a signal to be sent to the user if any input arrives on the Stream referenced by *conn_fd*. If a disconnect indication arrives, the signal catching routine (**connrelease**) prints an appropriate error message and then exits.

If the data traffic flowed in both directions in this example, the user would not have to monitor the connection for disconnects. If the client alternated **t_snd** and **t_rcv** calls, it could rely on **t_rcv** to recognize an incoming disconnect indication.

3.5 Connection Release

Either user can release the transport connection and end the conversation at any point during data transfer. As mentioned earlier, two forms of connection release are supported by the Transport Layer Interface. The first, abortive release, breaks a connection immediately and can result in the loss of any data that has not yet reached the destination user. **t_snddis** can be called by either user to generate an abortive release. Also, the transport provider can abort a connection if a problem occurs below the Transport Layer Interface. **t_snddis** enables a user to send data to the remote user when aborting a connection. Although the abortive release is supported by all transport providers, the ability to send data when aborting a connection is not.

When the remote user is notified of the aborted connection, **t_rcvdis** must be called to retrieve the disconnect indication. This call returns a reason code that indicates why the connection was aborted, and it returns any user data that may have accompanied the disconnect indication (if the abortive release was initiated by the remote user). This reason code is specific to the underlying transport protocol, and it should not be interpreted by protocol-independent software.

The second form of connection release is orderly release, which gracefully terminates a connection and guarantees that no data is lost. All transport providers must support the abortive release procedure, but orderly release is an optional facility that is not supported by all transport protocols.

3.5.1 The Server

The client/server example in this chapter assumes that the transport provider supports the orderly release of a connection. When all the data has been transferred by the server, the connection can be released as follows:

```
        if (t_sndrel(conn_fd) < 0) {
            t_error("t_sndrel failed");
            exit(27);
        }
        pause();    /* until orderly release indication arrives */
    }
}
```

The orderly release procedure consists of two steps by each user. The first user to complete data transfer can initiate a release using **t_sndrel**, as illustrated in the example. This routine informs the client that no more

data will be sent by the server. When the client receives such an indication, it can continue sending data back to the server if desired. When all data has been transferred, however, the client must also call **t_sndrel** to indicate that it is ready to release the connection. The connection is released only after both users have requested an orderly release and received the corresponding indication from the other user.

In this example, data is transferred in one direction from the server to the client, and so the server does not expect to receive data from the client after it has initiated the release procedure. Thus, the server simply calls **pause** [see *pause*(S)] after initiating the release. Eventually, the remote user responds with its orderly release request, and the indication generates a signal that will be caught by **connrelease**. Remember that the server earlier issued an I_SETSIG **ioctl** call to generate a signal on any incoming event. Since the only possible Transport Layer Interface events that can occur in this situation are a disconnect indication or an orderly release indication, **connrelease** terminates normally when the orderly release indication arrives. The **exit** call in **connrelease** closes the transport endpoint, thereby freeing the bound address for use by another user. If a user process wants to close a transport endpoint without exiting, it can call **t_close**.

3.5.2 The Client

The client's view of connection release is similar to that of the server. As mentioned earlier, the client continues to process incoming data until **t_rcv** fails. If the server releases the connection (using either **t_snddis** or **t_sndrel**), **t_rcv** fails and sets **t_errno** to TLOOK. The client then processes the connection release as follows:

```
    if ((t_errno == TLOOK)  &&  (t_look(fd) == T_ORDREL)) {
        if (t_rcvrel(fd) < 0) {
            t_error("t_rcvrel failed");
            exit(6);
        }
        if (t_sndrel(fd) < 0) {
            t_error("t_sndrel failed");
            exit(7);
        }
        exit(0);
    }
    t_error("t_rcv failed");
    exit(8);
}
```

Under normal circumstances, the client terminates the transfer of data by calling **t_sndrel** to initiate the connection release. When the orderly

release indication arrives at the client's side of the connection, the client checks to make sure that the expected orderly release indication has arrived. If so, it proceeds with the release procedures by calling **t_rcvrel** to process the indication and **t_sndrel** to inform the server that it is also ready to release the connection. At this point the client exits, thereby closing its transport endpoint.

Not all transport providers support the orderly release facility just described, and so users may have to use the abortive release facility provided by **t_snddis** and **t_rcvdis**. However, steps must be taken by each user to prevent any loss of data. For example, a special byte pattern can be inserted in the data stream to indicate the end of a conversation. Many mechanisms are possible for preventing data loss. Each application and high-level protocol must choose an appropriate mechanism given the target protocol environment and requirements.

Chapter 4

Connectionless-Mode Service

4.1 Introduction

This chapter describes the connectionless-mode service of the Transport Layer Interface. Connectionless-mode service is appropriate for short-term request/response interactions, such as transaction processing applications. Data are transferred in self-contained units with no logical relationship required among multiple units.

The connectionless-mode services are described using a transaction server as an example. This server waits for incoming transaction queries and then processes and responds to each query.

4.2 Local Management

Like connection-mode service, the transport users must perform appropriate local management steps before data can be transferred. A user must choose the appropriate connectionless service provider using **t_open** and establish its identity using **t_bind**.

t_optmgmt can be used to negotiate protocol options that may be associated with the transfer of each data unit. As with the connection-mode service, each transport provider specifies the options, if any, that it supports. Option negotiation is therefore a protocol-specific activity.

In the example, the definitions and local management calls needed by the transaction server are as follows:

```
#include <stdio.h>
#include <fcntl.h>
#include <tiuser.h>

#define SRV_ADDR   2       /* server's well-known address */

main()
{
   int fd;
   int flags;

   struct t_bind *bind;
   struct t_unitdata *ud;
   struct t_uderr *uderr;

   extern int t_errno;

   if ((fd = t_open("/dev/tidg", O_RDWR, NULL)) < 0) {
       t_error("unable to open /dev/provider");
       exit(1);
   }

   if ((bind = (struct t_bind *)t_alloc(fd, T_BIND, T_ADDR)) == NULL) {
       t_error("t_alloc of t_bind structure failed");
       exit(2);
   }

   bind->addr.len = sizeof(int);
   *(int *)bind->addr.buf = SRV_ADDR;
   bind->qlen = 0;

   if (t_bind(fd, bind, bind) < 0) {
       t_error("t_bind failed");
       exit(3);
   }

   /*
    * is the bound address correct?
    */

   if (*(int *)bind->addr.buf != SRV_ADDR) {
       fprintf(stderr, "t_bind bound wrong address\n");
       exit(4);
   }
```

The local management steps should look familiar by now. The server establishes a transport endpoint with the desired transport provider using **t_open**. Each provider has an associated service type, and so the user can choose a particular service by opening the appropriate transport provider

file. This connectionless-mode server ignores the characteristics of the provider returned by **t_open** in the same way as the users in the connection-mode example, setting the third argument to NULL. For simplicity, the transaction server assumes the transport provider has the following characteristics:

- The transport address is an integer value that uniquely identifies each user.
- The transport provider supports the T_CLTS service type (connectionless transport service, or datagram).
- The transport provider does not support any protocol-specific options.

The connectionless server also binds a transport address to the endpoint so that potential clients can identify and access the server. A **t_bind** structure is allocated using **t_alloc**, and the *buf* and *len* fields of the address are set accordingly.

One important difference between the connection-mode server and this connectionless-mode server is that the *qlen* field of the **t_bind** structure has no meaning for connectionless-mode service. That is because all users are capable of receiving datagrams once they have bound an address. The Transport Layer Interface defines an inherent client/server relationship between two users while establishing a transport connection in the connection-mode service. However, no such relationship exists in the connectionless-mode service. It is the context of this example, not the Transport Layer Interface, that defines one user as a server and another as a client.

Because the address of the server is known by all potential clients, the server checks the bound address returned by **t_bind** to ensure that it is correct.

4.3 Data Transfer

Once a user has bound an address to the transport endpoint, datagrams can be sent or received over that endpoint. Each outgoing message is accompanied by the address of the destination user. In addition, the Transport Layer Interface enables a user to specify protocol options that should be associated with the transfer of the data unit (for example, transit delay). As discussed earlier, each transport provider defines the set of options, if any, that can accompany a datagram. When the datagram is passed to the destination user, the associated protocol options can be

returned as well.

The following sequence of calls illustrates the data transfer phase of the connectionless-mode server:

```
    if ((ud = (struct t_unitdata *)t_alloc(fd, T_UNITDATA, T_ALL)) == NULL) {
        t_error("t_alloc of t_unitdata structure failed");
        exit(5);
    }

    if ((uderr = (struct t_uderr *)t_alloc(fd, T_UDERROR, T_ALL)) == NULL) {
        t_error("t_alloc of t_uderr structure failed");
        exit(6);
    }

    while (1) {
        if (t_rcvudata(fd, ud, &flags) < 0) {
            if (t_errno == TLOOK) {

                /*
                 * Error on previously sent datagram
                 */

                if (t_rcvuderr(fd, uderr) < 0) {
                    exit(7);
                }

                fprintf(stderr, "bad datagram, error = %d\n",
                    uderr->error);
                continue;
            }
            t_error("t_rcvudata failed");
            exit(8);
        }

        /*
         * Query() processes the request and places the
         * response in ud->udata.buf, setting ud->udata.len
         */

        query(ud);

        if (t_sndudata(fd, ud, 0) < 0) {
            t_error("t_sndudata failed");
            exit(9);
        }
    }
}

query()
{

    /* Merely a stub for simplicity */

}
```

The server must first allocate a **t_unitdata** structure for storing datagrams, which has the following format:

```
struct t_unitdata {
   struct netbuf addr;
   struct netbuf opt;
   struct netbuf udata;
}
```

addr holds the source address of incoming datagrams and the destination address of outgoing datagrams, *opt* identifies any protocol options associated with the transfer of the datagram, and *udata* holds the data itself. The *addr*, *opt*, and *udata* fields must all be allocated with buffers that are large enough to hold any possible incoming values. As described in the previous chapter, the T_ALL argument to **t_alloc** ensures this and sets the *maxlen* field of each **netbuf** structure accordingly. Because the provider does not support protocol options in this example, no options buffer is allocated, and *maxlen* is set to zero in the **netbuf** structure for options. A **t_uderr** structure is also allocated by the server for processing any datagram errors, as will be discussed later in this chapter.

The transaction server loops forever, receiving queries, processing the queries, and responding to the clients. It first calls **t_rcvudata** to receive the next query. **t_rcvudata** retrieves the next available incoming datagram. If none is currently available, **t_rcvudata** blocks, waiting for a datagram to arrive. The second argument of **t_rcvudata** identifies the **t_unitdata** structure where the datagram should be stored.

The third argument, *flags*, must point to an integer variable. *flags* can be set to T_MORE on return from **t_rcvudata** to indicate that the user's *udata* buffer was not large enough to store the full datagram. In this case, subsequent calls to **t_rcvudata** will retrieve the remainder of the datagram. **t_alloc** allocates a *udata* buffer large enough to store the maximum datagram size, and so the transaction server does not have to check the value of *flags*.

If a datagram is received successfully, the transaction server calls the *query* routine to process the request. This routine stores the response in the structure pointed to by *ud*, and sets *ud->udata.len* to indicate the number of bytes in the response. The source address returned by **t_rcvudata** in *ud->addr* is used as the destination address by **t_sndudata**.

When the response is ready, **t_sndudata** is called to return the response to the client. The Transport Layer Interface prevents a user from flooding the transport provider with datagrams using the same flow control mechanism described for the connection-mode service. In such cases, **t_sndudata** blocks until the flow control is relieved, and then resumes its operation.

4.4 Datagram Errors

If the transport provider cannot process a datagram that was passed to it by **t_sndudata**, it returns a unit data error event, T_UDERR, to the user. This event includes the destination address and options associated with the datagram, plus a protocol-specific error value that describes what may be wrong with the datagram. The reason a datagram could not be processed is protocol-specific. One reason may be that the transport provider could not interpret the destination address or options. Each transport protocol is expected to specify all reasons for which it is unable to process a datagram.

Note

The unit data error indication is not necessarily intended to indicate success or failure in delivering the datagram to the specified destination. The transport protocol decides how the indication will be used. Remember, the connectionless service does not guarantee reliable delivery of data.

The transaction server is notified of this error event when it attempts to receive another datagram. In this case, **t_rcvudata** fails, setting **t_errno** to TLOOK. If TLOOK is set, the only possible event is T_UDERR, and so the server calls **t_rcvuderr** to retrieve the event. The second argument to **t_rcvuderr** is the **t_uderr** structure which was allocated earlier. This structure is filled in by **t_rcvuderr** and has the following format:

```
struct t_uderr {
   struct netbuf addr;
   struct netbuf opt;
   long error;
}
```

where *addr* and *opt* identify the destination address and protocol options as specified in the bad datagram, and *error* is a protocol-specific error code that indicates why the provider could not process the datagram. The transaction server prints the error code and then continues by entering the processing loop again.

Chapter 5

A Read/Write Interface

5.1 Introduction

Sometimes a user may want to establish a transport connection and then **exec** [see *exec*(S)] an existing user program such as **cat** [see *cat*(C)] to process the data as it arrives over the connection. However, existing programs use **read** and **write** for their I/O needs. The Transport Layer Interface does not directly support a **read/write** interface to a transport provider, but one is available with the UNIX System. This interface enables a user to issue **read** and **write** calls over a transport connection that is in the data transfer phase. This chapter describes the **read/write** interface to the connection-mode service of the Transport Layer Interface. This interface is not available with the connectionless-mode service.

The **read/write** interface is presented using the client example of Chapter 3 with some minor modifications. The clients are identical until the data transfer phase is reached. At that point, this client uses the **read/write** interface and **cat** to process incoming data. **cat** can be run without change over the transport connection. Only the differences between this client and that of the example in Chapter 3 are shown below.

```
#include <stropts.h>
    .
    .   /*
    .    * Same local management and connection
    .    * establishment steps.
    .    */
    .
    if (ioctl(fd, I_PUSH, "tirdwr") < 0) {
        perror("I_PUSH of tirdwr failed");
        exit(5);
    }
    close(0);
    dup(fd);
    execl("/bin/cat", "/bin/cat", 0);
    perror("execl of /bin/cat failed");
    exit(6);
}
```

The client invokes the **read/write** interface by pushing the **tirdwr** module [see *tirdwr*(STR)] onto the Stream associated with the transport endpoint where the connection was established [see I_PUSH in *streamio*(STR)]. This module converts the Transport Layer Interface above the transport provider into a pure **read/write** interface. With the module in place, the client calls **close** [see *close*(S)] and **dup** [see *dup*(S)] to establish the transport endpoint as its standard input file, and uses **/bin/cat** to process the input. Because the transport endpoint identifier is a file descriptor, the facility for **dup**ing the endpoint is available to users.

Because the Transport Layer Interface has been implemented using STREAMS, the facilities of this character I/O mechanism can be used to provide enhanced user services. By pushing the **tirdwr** module above the transport provider, the user's interface is effectively changed. The semantics of **read** and **write** must be followed, and message boundaries are not preserved.

Warning

The **tirdwr** module can be pushed onto a Stream only when the transport endpoint is in the data transfer phase. Once the module is pushed, the user cannot call any Transport Layer Interface routines. If a Transport Layer Interface routine is invoked, **tirdwr** will generate a fatal protocol error, EPROTO, on that Stream, rendering it unusable. Furthermore, if the user pops the **tirdwr** module off the Stream [see I_POP in *streamio*(STR)], the transport connection will be aborted.

The exact semantics of **write**, **read**, and **close** using **tirdwr** are described below. To summarize, **tirdwr** enables a user to send and receive data over a transport connection using **read** and **write**. This module translates all Transport Layer Interface indications into the appropriate actions. The connection can be released with the **close** system call.

5.2 write

The user can transmit data over the transport connection using **write**. The **tirdwr** module passes data through to the transport provider. However, if a user attempts to send a zero-length data packet, which the STREAMS mechanism allows, **tirdwr** will discard the message. If for some reason the transport connection is aborted (for example, if the remote user aborts the connection using **t_snddis**), a STREAMS hangup condition will be generated on that Stream, and further **write** calls will fail and set *errno* to ENXIO. The user can still retrieve any available data after a hangup, however.

5.3 read

read can be used to retrieve data that has arrived over the transport connection. The **tirdwr** module passes data through to the user from the transport provider. However, any other event or indication passed to the user from the provider is processed by **tirdwr** as follows:

- **read** cannot process expedited data because it cannot distinguish expedited data from normal data for the user. If an expedited data indication is received, **tirdwr** generates a fatal protocol error, EPROTO, on that Stream. This error causes further system calls to fail. Thus, you should not communicate with a process that is sending expedited data.

- If an abortive disconnect indication is received, **tirdwr** discards the indication and generates a STREAMS hangup condition on that Stream. Subsequent **read** calls retrieve any remaining data, and then **read** returns zero for all further calls (indicating end-of-file).

- If an orderly release indication is received, **tirdwr** discards the indication and delivers a zero-length STREAMS message to the user. As described in **read**, this notifies the user of end-of-file by returning 0 to the user.

- If any other Transport Layer Interface indication is received, **tirdwr** generates a fatal protocol error, EPROTO, on that Stream. This causes further system calls to fail. If a user pushes **tirdwr** onto a Stream after the connection has been established, such indications are not generated.

5.4 Close

With **tirdwr** on a Stream, the user can send and receive data over a transport connection for the duration of that connection. Either user can terminate the connection by closing the file descriptor associated with the transport endpoint or by popping the **tirdwr** module off the Stream. In either case, **tirdwr** takes the following actions:

- If an orderly release indication was previously received by **tirdwr**, an orderly release request is passed to the transport provider to complete the orderly release of the connection. The remote user who initiated the orderly release procedure receives the expected indication when data transfer completes.

- If a disconnect indication was previously received by **tirdwr**, no

special action is taken.

- If neither an orderly release indication nor disconnect indication was previously received by **tirdwr**, a disconnect request is passed to the transport provider to abortively release the connection.

- If an error previously occurred on the Stream and a disconnect indication has not been received by **tirdwr**, a disconnect request is passed to the transport provider.

A process cannot initiate an orderly release after **tirdwr** is pushed onto a Stream, but **tirdwr** will handle an orderly release properly if it is initiated by the user on the other side of a transport connection. If the client in this chapter is communicating with the server program in Chapter 3, that server terminates the transfer of data with an orderly release request. The server then waits for the corresponding indication from the client. At that point, the client exits and the transport endpoint is closed. As explained in the first bulleted item above, when the file descriptor is closed, **tirdwr** initiates the orderly release request from the client's side of the connection. This generates the indication that the server is expecting, and the connection is properly released.

Chapter 6

Advanced Topics

6.1 Introduction

This chapter presents important concepts that have not been covered in the previous chapters. First, an optional non-blocking (asynchronous) mode for some library calls is described. Then an advanced programming example is presented. This example defines a server which supports multiple outstanding connect indications and operates in an event-driven manner.

6.2 Asynchronous Execution Mode

Many Transport Layer Interface library routines can block waiting for an incoming event or the relaxation of flow control. However, some time-critical applications should not block for any reason. Similarly, an application may wish to do local processing while waiting for some asynchronous Transport Layer Interface event.

Support for asynchronous processing of Transport Layer Interface events is available to applications using a combination of the STREAMS asynchronous features and the non-blocking mode of the Transport Layer Interface library routines. Earlier examples in this guide have illustrated the use of the STREAMS **poll** system call and the I_SETSIG **ioctl** command for processing events in an asynchronous manner.

In addition, each Transport Layer Interface routine that can block waiting for some event can be run in a special non-blocking mode. For example, **t_listen** normally blocks, waiting for a connect indication. However, a server can periodically poll a transport endpoint for existing connect indications by calling **t_listen** in the non-blocking (or asynchronous) mode. The asynchronous mode is enabled by setting O_NDELAY on the file descriptor. This can be set as a flag on **t_open** or by calling **fcntl** [see *fcntl*(S)] before calling the Transport Layer Interface routine. **fcntl** can be used to enable or disable this mode at any time. All programming examples illustrated throughout this guide use the default, synchronous mode of processing.

O_NDELAY affects each Transport Layer Interface routine in a different manner. To determine the exact semantics of O_NDELAY for a particular routine, see the appropriate pages in Section NSL in Appendix D.

6.3 Advanced Programming Example

The following example illustrates two important concepts. The first is a server's ability to manage multiple outstanding connect indications. The second is the ability to write event-driven software using the Transport Layer Interface and the STREAMS system call interface.

The server example in Chapter 3 is capable of supporting only one outstanding connect indication, but the Transport Layer Interface supports the ability to manage multiple outstanding connect indications. One reason a server might wish to receive several, simultaneous connect indications is to impose a priority scheme on each client. A server can retrieve several connect indications, and then accept them in an order based on a priority associated with each client. A second reason for handling several outstanding connect indications is that the single-threaded scheme has some limitations. Depending on the implementation of the transport provider, it is possible that while the server is processing the current connect indication, other clients will find it busy. However, if multiple connect indications can be processed simultaneously, the server will be found to be busy only if the maximum allowed number of clients attempt to call the server simultaneously.

The server example is event-driven: the process polls a transport endpoint for incoming Transport Layer Interface events and then takes the appropriate actions for the current event. The example demonstrates the ability to poll multiple transport endpoints for incoming events.

The definitions and local management functions needed by this example are similar to those of the server example in Chapter 3.

```
#include <tiuser.h>
#include <fcntl.h>
#include <stdio.h>
#include <poll.h>
#include <stropts.h>
#include <signal.h>

#define NUM_FDS          1
#define MAX_CONN_IND 4
#define SRV_ADDR     1       /* server's well-known address */

int conn_fd;                 /* server connection here */
struct t_call *calls[NUM_FDS][MAX_CONN_IND];/* holds connect indications */
extern int t_errno;

main()
{
   struct pollfd pollfds[NUM_FDS];
   struct t_bind *bind;
   int i;

   /*
    * Only opening and binding one transport endpoint,
    * but more could be supported
    */
   if ((pollfds[0].fd = t_open("/dev/tivc", O_RDWR, NULL)) < 0) {
       t_error("t_open failed");
       exit(1);
   }

   if ((bind = (struct t_bind *)t_alloc(pollfds[0].fd, T_BIND, T_ALL)) == NULL) {
       t_error("t_alloc of t_bind structure failed");
       exit(2);
   }
   bind->qlen = MAX_CONN_IND;
   bind->addr.len = sizeof(int);
   *(int *)bind->addr.buf = SRV_ADDR;
   if (t_bind(pollfds[0].fd, bind, bind) < 0) {
       t_error("t_bind failed");
       exit(3);
   }

   /*
    * Was the correct address bound?
    */
   if (*(int *)bind->addr.buf != SRV_ADDR) {
       fprintf(stderr, "t_bind bound wrong address\n");
       exit(4);
   }
```

The file descriptor returned by **t_open** is stored in a **pollfd** structure [see *poll*(S)] that is used to poll the transport endpoint for incoming data. Notice that only one transport endpoint is established in this example. However, the remainder of the example is written to manage multiple transport endpoints. Several endpoints could be supported with minor changes to the above code.

An important aspect of this server is that it sets *qlen* to a value greater than 1 for **t_bind**. This indicates that the server is willing to handle multiple outstanding connect indications. Remember that the earlier examples single-threaded the connect indications and responses. The server would accept the current connect indication before retrieving additional connect indications. However, this example can retrieve up to MAX_CONN_IND connect indications at one time before responding to any of them. The transport provider can negotiate the value of *qlen* downward if it cannot support MAX_CONN_IND outstanding connect indications.

Once the server has bound its address and is ready to process incoming connect requests, it does the following:

```
    pollfds[0].events = POLLIN;
    while (1) {
        if (poll(pollfds, NUM_FDS, -1) < 0) {
            perror("poll failed");
            exit(5);
        }
        for (i = 0; i < NUM_FDS; i++) {
            switch (pollfds[i].revents) {
            default:
                perror("poll returned error event");
                exit(6);
            case 0:
                continue;
            case POLLIN:
                do_event(i, pollfds[i].fd);
                service_conn_ind(i, pollfds[i].fd);
            }
        }
    }
}
```

The *events* field of the **pollfd** structure is set to POLLIN, which will notify the server of any incoming Transport Layer Interface events. The server then enters an infinite loop, in which it will **poll** the transport endpoint(s) for events and then process those events as they occur.

The **poll** call blocks indefinitely, waiting for an incoming event. On return, each entry (corresponding to each transport endpoint) is checked for an existing event. If *revents* is set to 0, no event has occurred on that endpoint. In this case, the server continues to the next transport endpoint. If *revents* is set to POLLIN, an event does exist on the endpoint. In this case, **do_event** is called to process the event. If *revents* contains any other value, an error must have occurred on the transport endpoint, and the server exits.

For each iteration of the loop, if any event is found on the transport endpoint, **service_conn_ind** is called to process any outstanding connect indications. However, if another connect indication is pending, **service_conn_ind** saves the current connect indication and respond to it later. This routine will be explained shortly.

If an incoming event is discovered, the following routine is called to process it:

```
do_event(slot, fd)
{
  struct t_discon *discon;
  int i;

  switch (t_look(fd)) {

  default:
    fprintf(stderr,"t_look returned an unexpected event\n");
    exit(7);

  case T_ERROR:
    fprintf(stderr,"t_look returned T_ERROR event\n");
    exit(8);

  case -1:
    t_error("t_look failed");
    exit(9);

  case 0:
    /* since POLLIN returned, this should not happen */
    fprintf(stderr,"t_look returned no event\n");
    exit(10);

  case T_LISTEN:
    /*
     * find free element in calls array
     */
    for (i = 0; i < MAX_CONN_IND; i++) {
       if (calls[slot][i] == NULL)
           break;
    }
```

```
    if ((calls[slot][i] = (struct t_call *)t_alloc(fd, T_CALL, T_ALL)) == NULL) {
        t_error("t_alloc of t_call structure failed");
        exit(11);
    }
    if (t_listen(fd, calls[slot][i]) < 0) {
        t_error("t_listen failed");
        exit(12);
    }
    break;
  case T_DISCONNECT:
    discon = (struct t_discon *)t_alloc(fd, T_DIS, T_ALL);
    if (t_rcvdis(fd, discon) < 0) {
        t_error("t_rcvdis failed");
        exit(13);
    }
    /*
     * find call ind in array and delete it
     */
    for (i = 0; i < MAX_CONN_IND; i++) {
        if (discon->sequence == calls[slot][i]->sequence) {
            t_free(calls[slot][i], T_CALL);
            calls[slot][i] = NULL;
        }
    }
    t_free(discon, T_DIS);
    break;
  }
}
```

This routine takes a number, *slot*, and a file descriptor, *fd*, as arguments. *slot* is used as an index into the global array *calls*. This array contains an entry for each polled transport endpoint, where each entry consists of an array of **t_call** structures that hold incoming connect indications for that transport endpoint. The value of *slot* is used to identify the transport endpoint of interest.

do_event calls **t_look** to determine the Transport Layer Interface event which has occurred on the transport endpoint referenced by *fd*. If a connect indication (T_LISTEN event) or disconnect indication (T_DISCONNECT event) has arrived, the event is processed. Otherwise, the server prints an appropriate error message and exits.

For connect indications, **do_event** scans the array of outstanding connect indications looking for the first free entry. A **t_call** structure is then allocated for that entry, and the connect indication is retrieved using **t_listen**. There must always be at least one free entry in the connect indication array because the array is large enough to hold the maximum number of outstanding connect indications as negotiated by **t_bind**. The processing of the connect indication is deferred until later.

If a disconnect indication arrives, it must correspond to a previously received connect indication. This scenario arises if a client attempts to undo a previous connect request. In this case, **do_event** allocates a **t_discon** structure to retrieve the relevant disconnect information. This structure has the following members:

```
struct t_discon {
   struct netbuf udata;
   int reason;
   int sequence;
}
```

where *udata* identifies any user data which might have been sent with the disconnect indication, *reason* contains a protocol-specific disconnect reason code, and *sequence* identifies the outstanding connect indication that matches this disconnect indication.

Next, **t_rcvdis** is called to retrieve the disconnect indication. The array of connect indications for *slot* is then scanned for one which contains a sequence number that matches the *sequence* number in the disconnect indication. When the connect indication is found, it is freed and the corresponding entry is set to NULL.

As mentioned earlier, if any event is found on a transport endpoint, **service_conn_ind** is called to process all currently outstanding connect indications associated with that endpoint as follows:

```
service_conn_ind(slot, fd)
{
   int i;

   for (i = 0; i < MAX_CONN_IND; i++) {
       if (calls[slot][i] == NULL)
           continue;

       if ((conn_fd = t_open("/dev/tivc", O_RDWR, NULL)) < 0) {
           t_error("open failed");
           exit(14);
       }
       if (t_bind(conn_fd, NULL, NULL) < 0) {
           t_error("t_bind failed");
           exit(15);
       }
```

```
        if (t_accept(fd, conn_fd, calls[slot][i]) < 0) {
            if (t_errno == TLOOK) {
                t_close(conn_fd);
                return;
            }
            t_error("t_accept failed");
            exit(16);
        }
        t_free(calls[slot][i], T_CALL);
        calls[slot][i] = NULL;

        run_server(fd);
    }
}
```

For the given slot (the transport endpoint), the array of outstanding connect indications is scanned. For each indication, the server opens a responding transport endpoint, binds an address to the endpoint, and then accepts the connection on that endpoint. If another event (connect indication or disconnect indication) arrives before the current indication is accepted, **t_accept** will fail and set **t_errno** to TLOOK.

Note

The user cannot accept an outstanding connect indication if any pending connect indication events or disconnect indication events exist on that transport endpoint.

If this error occurs, the responding transport endpoint is closed and **service_conn_ind** returns immediately (saving the current connect indication for later processing). This causes the server's main processing loop to be entered, and the new event is discovered by the next call to **poll**. In this way, multiple connect indications can be queued by the user.

Eventually, all events are processed, and **service_conn_ind** is able to accept each connect indication in turn. Once the connection is established, the **run_server** routine used by the server in Chapter 3 is called to manage the data transfer.

Appendix A

State Transitions

A.1 Introduction

The tables in this appendix describe all state transitions associated with the Transport Layer Interface. The states and events are described first, followed by the state transition tables.

A.2 Transport Layer Interface States

Figure A-1 defines the states used to describe the Transport Layer Interface state transitions.

State	Description	Service Type
T_UNINIT	uninitialized - initial and final state of interface	T_COTS, T_COTS_ORD, T_CLTS
T_UNBND	initialized but not bound	T_COTS, T_COTS_ORD, T_CLTS
T_IDLE	no connection established	T_COTS, T_COTS_ORD, T_CLTS
T_OUTCON	outgoing connection pending for client	T_COTS, T_COTS_ORD
T_INCON	incoming connection pending for server	T_COTS, T_COTS_ORD
T_DATAXFER	data transfer	T_COTS, T_COTS_ORD
T_OUTREL	outgoing orderly release (waiting for orderly release indication)	T_COTS_ORD
T_INREL	incoming orderly release (waiting to send orderly release request)	T_COTS_ORD

Figure A-1 Transport Layer Interface States

A.3 Outgoing Events

The outgoing events described in Figure A-2 correspond to the return of the specified transport routines, where these routines send a request or response to the transport provider.

In the figure, some events (such as *acceptN*) are distinguished by the context in which they occur. The context is based on the values of the

following variables:

ocnt count of outstanding connect indications

fd file descriptor of the current transport endpoint

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

Event	Description	Service Type
opened	successful return of **t_open**	T_COTS, T_COTS_ORD, T_CLTS
bind	successful return of **t_bind**	T_COTS, T_COTS_ORD, T_CLTS
optmgmt	successful return of **t_optmgmt**	T_COTS, T_COTS_ORD, T_CLTS
unbind	successful return of **t_unbind**	T_COTS, T_COTS_ORD, T_CLTS
closed	successful return of **t_close**	T_COTS, T_COTS_ORD, T_CLTS
connect1	successful return of **t_connect** in synchronous mode	T_COTS, T_COTS_ORD
connect2	TNODATA error on **t_connect** in asynchronous mode, or TLOOK error due to a disconnect indication arriving on the transport endpoint	T_COTS, T_COTS_ORD
accept1	successful return of **t_accept** with *ocnt* == 1, *fd* == *resfd*	T_COTS, T_COTS_ORD
accept2	successful return of **t_accept** with *ocnt* == 1, *fd* != *resfd*	T_COTS, T_COTS_ORD
accept3	successful return of **t_accept** with *ocnt* > 1	T_COTS, T_COTS_ORD
snd	successful return of **t_snd**	T_COTS, T_COTS_ORD
snddis1	successful return of **t_snddis** with *ocnt* <= 1	T_COTS, T_COTS_ORD
snddis2	successful return of **t_snddis** with *ocnt* > 1	T_COTS, T_COTS_ORD
sndrel	successful return of **t_sndrel**	T_COTS_ORD

sndudata	successful return of **t_sndudata**	T_CLTS

Figure A-2 Transport Layer Interface Outgoing Events

A.4 Incoming Events

The incoming events correspond to the successful return of the specified routines, where these routines retrieve data or event information from the transport provider. The only incoming event not associated directly with the return of a routine is *pass_conn*, which occurs when a user transfers a connection to another transport endpoint. This event occurs on the endpoint to which the connection is being passed, despite the fact that no Transport Layer Interface routine is issued on that endpoint. *pass_conn* is included in the state tables to describe the behavior when a user accepts a connection on another transport endpoint.

In Figure A-3, the *rcvdis* events are distinguished by the context in which they occur. The context is based on the value of *ocnt*, which is the count of outstanding connect indications on the transport endpoint.

Incoming Event	Description	Service Type
listen	successful return of **t_listen**	T_COTS, T_COTS_ORD
rcvconnect	successful return of **t_rcvconnect**	T_COTS, T_COTS_ORD
rcv	successful return of **t_rcv**	T_COTS, T_COTS_ORD
rcvdis1	successful return of **t_rcvdis** with **ocnt** <= 0	T_COTS, T_COTS_ORD
rcvdis2	successful return of **t_rcvdis** with *ocnt* == 1	T_COTS, T_COTS_ORD
rcvdis3	successful return of **t_rcvdis** with *ocnt* > 1	T_COTS, T_COTS_ORD
rcvrel	successful return of **t_rcvrel**	T_COTS_ORD
rcvudata	successful return of **t_rcvudata**	T_CLTS
rcvuderr	successful return of **t_rcvuderr**	T_CLTS
pass_conn	receive a passed connection	T_COTS, T_COTS_ORD

Figure A-3 Transport Layer Interface Incoming Events

A.5 Transport User Actions

In the state tables that follow, some state transitions are accompanied by a list of actions the transport user must take. These actions are represented by the notation [*n*], where *n* is the number of the specific action as described below:

[1] Set the count of outstanding connect indications to zero.

[2] Increment the count of outstanding connect indications.

[3] Decrement the count of outstanding connect indications.

[4] Pass a connection to another transport endpoint as indicated in *t_accept*.

A.6 State Tables

The following tables describe the Transport Layer Interface state transitions. Given a current state and an event, the transition to the next state is shown, as well as any actions that must be taken by the transport user (indicated by [*n*]). The state is that of the transport provider as seen by the transport user.

The contents of each box represent the next state, given the current state (column) and the current incoming or outgoing event (row). An empty box represents a state/event combination that is invalid. Along with the next state, each box may include an action list (as specified in the previous section). The transport user must take the specific actions in the order specified in the state table.

The following should be understood when studying the state tables:

- The **t_close** routine is referenced in the state tables (see *closed* event in Figure A-2), but can be called from any state to close a transport endpoint. If **t_close** is called when a transport address is bound to an endpoint, the address will be unbound. Also, if **t_close** is called when the transport connection is still active, the connection will be aborted.

- If a transport user issues a routine out of sequence, the transport provider will recognize this and the routine will fail, setting **t_errno** to TOUTSTATE. The state will not change.

- If any other transport error occurs, the state will not change unless

explicitly stated on the manual page for that routine. The exception to this is a TLOOK or TNODATA error on **t_connect**, as described in Figure A-2. The state tables assume correct use of the Transport Layer Interface.

- The support routines **t_getinfo, t_getstate, t_alloc, t_free, t_sync, t_look,** and **t_error** are excluded from the state tables because they do not affect the state.

A separate table is shown for common local management steps, data transfer in connectionless-mode, and connection-establishment/ connection-release/ data-transfer in connection-mode.

state / event	T_UNINIT	T_UNBND	T_IDLE
opened	T_UNBND		
bind		T_IDLE [1]	
optmgmt			T_IDLE
unbind			T_UNBND
closed		T_UNINIT	

Figure A-4 Common Local Management State Table

state / event	T_IDLE
sndudata	T_IDLE
rcvudata	T_IDLE
rcvuderr	T_IDLE

Figure A-5 Connectionless-Mode State Table

state / event	T_IDLE	T_OUTCON	T_INCON	T_DATAXFER	T_OUTREL	T_INREL
connect1	T_DATAXFER					
connect2	T_OUTCON					
rcvconnect		T_DATAXFER				
listen	T_INCON [2]		T_INCON [2]			
accept1			T_DATAXFER[3]			
accept2			T_IDLE [3][4]			
accept3			T_INCON [3][4]			
snd				T_DATAXFER		T_INREL
rcv				T_DATAXFER	T_OUTREL	
snddis1		T_IDLE	T_IDLE [3]	T_IDLE	T_IDLE	T_IDLE
snddis2			T_INCON [3]			
rcvdis1		T_IDLE		T_IDLE	T_IDLE	T_IDLE
rcvdis2			T_IDLE [3]			
rcvdis3			T_INCON [3]			
sndrel				T_OUTREL		T_IDLE
rcvrei				T_INREL	T_IDLE	
pass_conn	T_DATAXFER					

Figure A-6 Connection-Mode State Table

Appendix B
Protocol Independence

B.1 Guidelines for Protocol Independence

The Transport Layer Interface offers protocol independence for user software by defining a set of services common to many transport protocols. However, not all transport protocols support all the services supported by the Transport Layer Interface. If software must be run in a variety of protocol environments, only the common services should be accessed. The following guidelines highlight services that may not be common to all transport protocols:

- In the connection-mode service, the concept of a transport service data unit (TSDU) may not be supported by all transport providers. The user should make no assumptions about the preservation of logical data boundaries across a connection. If messages must be transferred over a connection, a protocol should be implemented above the Transport Layer Interface to support message boundaries.

- Protocol- and implementation-specific service limits are returned by the **t_open** and **t_getinfo** routines. These limits are useful when allocating buffers to store protocol-specific transport addresses and options. It is the responsibility of the user to access these limits and then adhere to the limits throughout the communication process.

- User data should not be transmitted with connect requests or disconnect requests [see *t_connect*(NSL) and *t_snddis*(NSL)]. Not all transport protocols support this capability.

- The buffers in the **t_call** structure used for **t_listen** must be large enough to hold any information passed by the client during connection establishment. The server should use the T_ALL argument to **t_alloc**, which will determine the maximum buffer sizes needed to store the address, options, and user data for the current transport provider.

- The user program should not look at or change options that are associated with any Transport Layer Interface routine. These options are specific to the underlying transport protocol. The user should choose not to pass options with **t_connect** or **t_sndudata**. In such cases, the transport provider will use default values. Also, a server should use the options returned by **t_listen** when accepting a connection.

- Protocol-specific addressing issues should be hidden from the user program. A client should not specify any protocol address on

t_bind, but instead should allow the transport provider to assign an appropriate address to the transport endpoint. Similarly, a server should retrieve its address for **t_bind** in such a way that it does not require knowledge of the transport provider's address space. Such addresses should not be hard-coded into a program. A name server mechanism could be useful in this scenario, but the details for providing such a service are outside the scope of the Transport Layer Interface.

- The reason codes associated with **t_rcvdis** are protocol-dependent. The user should not interpret this information if protocol independence is a concern.

- The error codes associated with **t_rcvuderr** are protocol-dependent. The user should not interpret this information if protocol independence is a concern.

- The names of devices should not be hard-coded into programs, because the device node identifies a particular transport provider and is not protocol-independent.

- The optional orderly release facility of the connection-mode service (provided by **t_sndrel** and **t_rcvrel**) should not be used by programs targeted for multiple protocol environments. This facility is not supported by all connection-based transport protocols. In particular, its use will prevent programs from successfully communicating with ISO open systems.

Appendix C

Examples

C.1 Introduction

The examples presented throughout this guide are shown in their entirety in this appendix.

C.2 Connection-Mode Client

The following code is for the connection-mode client program described in Chapter 3. This client establishes a transport connection with a server and then receives data from the server and writes it to its standard output. The connection is released using the orderly release facility of the Transport Layer Interface. This client will communicate with each of the connection-mode servers presented in the guide.

```
#include <stdio.h>
#include <tiuser.h>
#include <fcntl.h>

#define SRV_ADDR   1   /* server's well-known address */

main()
{
   int fd;
   int nbytes;
   int flags = 0;
   char buf[1024];
   struct t_call *sndcall;
   extern int t_errno;

   if ((fd = t_open("/dev/tivc", O_RDWR, NULL)) < 0) {
       t_error("t_open failed");
       exit(1);
   }

   if (t_bind(fd, NULL, NULL) < 0) {
       t_error("t_bind failed");
       exit(2);
   }

   /*
    * By assuming that the address is an integer value,
    * this program may not run over another protocol.
    */
   if ((sndcall = (struct t_call *)t_alloc(fd, T_CALL, T_ADDR)) == NULL) {
       t_error("t_alloc failed");
       exit(3);
   }
   sndcall->addr.len = sizeof(int);
   *(int *)sndcall->addr.buf = SRV_ADDR;
```

```
    if (t_connect(fd, sndcall, NULL) < 0) {
        t_error("t_connect failed for fd");
        exit(4);
    }

    while ((nbytes = t_rcv(fd, buf, 1024, &flags)) != -1) {
        if (fwrite(buf, 1, nbytes, stdout) < 0) {
            fprintf(stderr, "fwrite failed\n");
            exit(5);
        }
    }

    if ((t_errno == TLOOK)  &&  (t_look(fd) == T_ORDREL)) {
        if (t_rcvrel(fd) < 0) {
            t_error("t_rcvrel failed");
            exit(6);
        }
        if (t_sndrel(fd) < 0) {
            t_error("t_sndrel failed");
            exit(7);
        }
        exit(0);
    }
    t_error("t_rcv failed");
    exit(8);
}
```

C.3 Connection-Mode Server

The following code is for the connection-mode server program described in Chapter 3. This server establishes a transport connection with a client and then transfers a log file to the client on the other side of the connection. The connection is released using the orderly release facility of the Transport Layer Interface. The connection-mode client presented earlier will communicate with this server.

```
#include <tiuser.h>
#include <stropts.h>
#include <fcntl.h>
#include <stdio.h>
#include <signal.h>

#define DISCONNECT -1
#define SRV_ADDR    1  /* server's well-known address */

int conn_fd;           /* connection established here */
extern int t_errno;
```

```
main()
{
    int listen_fd;      /* listening transport endpoint */
    struct t_bind *bind;
    struct t_call *call;

    if ((listen_fd = t_open("/dev/tivc", O_RDWR, NULL)) < 0) {
        t_error("t_open failed for listen_fd");
        exit(1);
    }

    /*
     * By assuming that the address is an integer value,
     * this program may not run over another protocol.
     */
    if ((bind = (struct t_bind *)t_alloc(listen_fd, T_BIND, T_ALL)) == NULL) {
        t_error("t_alloc of t_bind structure failed");
        exit(2);
    }
    bind->qlen = 1;
    bind->addr.len = sizeof(int);
    *(int *)bind->addr.buf = SRV_ADDR;

    if (t_bind(listen_fd, bind, bind) < 0) {
        t_error("t_bind failed for listen_fd");
        exit(3);
    }

    /*
     * Was the correct address bound?
     */
    if (*(int *)bind->addr.buf != SRV_ADDR) {
        fprintf(stderr, "t_bind bound wrong address\n");
        exit(4);
    }

    if ((call = (struct t_call *)t_alloc(listen_fd, T_CALL, T_ALL)) == NULL) {
        t_error("t_alloc of t_call structure failed");
        exit(5);
    }

    while (1) {
        if (t_listen(listen_fd, call) < 0) {
            t_error("t_listen failed for listen_fd");
            exit(6);
        }

        if ((conn_fd = accept_call(listen_fd, call)) != DISCONNECT)
            run_server(listen_fd);
    }
}
```

```
accept_call(listen_fd, call)
int listen_fd;
struct t_call *call;
{
    int resfd;

    if ((resfd = t_open("/dev/tivc", O_RDWR, NULL)) < 0) {
        t_error("t_open for responding fd failed");
        exit(7);
    }

    if (t_bind(resfd, NULL, NULL) < 0) {
        t_error("t_bind for responding fd failed");
        exit(8);
    }

    if (t_accept(listen_fd, resfd, call) < 0) {
        if (t_errno == TLOOK) {  /* must be a disconnect */
            if (t_rcvdis(listen_fd, NULL) < 0) {
                t_error("t_rcvdis failed for listen_fd");
                exit(9);
            }
            if (t_close(resfd) < 0) {
                t_error("t_close failed for responding fd");
                exit(10);
            }
            /* go back up and listen for other calls */
            return(DISCONNECT);
        }
        t_error("t_accept failed");
        exit(11);
    }
    return(resfd);
}

connrelease()
{
    /* conn_fd is global because needed here */
    if (t_look(conn_fd) == T_DISCONNECT) {
        fprintf(stderr, "connection aborted\n");
        exit(12);
    }

    /* else orderly release indication - normal exit */
    exit(0);
}
```

```
run_server(listen_fd)
int listen_fd;
{
   int nbytes;
   FILE *logfp;        /* file pointer to log file */
   char buf[1024];

   switch (fork()) {

   case -1:
       perror("fork failed");
       exit(20);

   default:    /* parent */

       /* close conn_fd and then go up and listen again */
       if (t_close(conn_fd) < 0) {
           t_error("t_close failed for conn_fd");
           exit(21);
       }
       return;

   case 0:     /* child */

       /* close listen_fd and do service */
       if (t_close(listen_fd) < 0) {
           t_error("t_close failed for listen_fd");
           exit(22);
       }
       if ((logfp = fopen("logfile", "r")) == NULL) {
           perror("cannot open logfile");
           exit(23);
       }

       signal(SIGPOLL, connrelease);
       if (ioctl(conn_fd, I_SETSIG, S_INPUT) < 0) {
           perror("ioctl I_SETSIG failed");
           exit(24);
       }
       if (t_look(conn_fd) != 0) { /* was disconnect already there? */
           fprintf(stderr, "t_look returned unexpected event\n");
           exit(25);
       }

       while ((nbytes = fread(buf, 1, 1024, logfp)) > 0)
           if (t_snd(conn_fd, buf, nbytes, 0) < 0) {
               t_error("t_snd failed");
               exit(26);
           }

       if (t_sndrel(conn_fd) < 0) {
           t_error("t_sndrel failed");
           exit(27);
       }
       pause();    /* until orderly release indication arrives */
   }
}
```

C.4 Connectionless-Mode Transaction Server

The following code is for the connectionless-mode transaction server program described in Chapter 4. This server waits for incoming datagram queries and then processes each query and sends a response.

```
#include <stdio.h>
#include <fcntl.h>
#include <tiuser.h>

#define SRV_ADDR   2       /* server's well-known address */

main()
{
   int fd;
   int flags;
   struct t_bind *bind;
   struct t_unitdata *ud;
   struct t_uderr *uderr;
   extern int t_errno;


   if ((fd = t_open("/dev/tidg", O_RDWR, NULL)) < 0) {
       t_error("unable to open /dev/provider");
       exit(1);
   }

   if ((bind = (struct t_bind *)t_alloc(fd, T_BIND, T_ADDR)) == NULL) {
       t_error("t_alloc of t_bind structure failed");
       exit(2);
   }
   bind->addr.len = sizeof(int);
   *(int *)bind->addr.buf = SRV_ADDR;
   bind->qlen = 0;

   if (t_bind(fd, bind, bind) < 0) {
       t_error("t_bind failed");
       exit(3);
   }

   /*
    * is the bound address correct?
    */
   if (*(int *)bind->addr.buf != SRV_ADDR) {
       fprintf(stderr, "t_bind bound wrong address\n");
       exit(4);
   }
```

```
    if ((ud = (struct t_unitdata *)t_alloc(fd, T_UNITDATA, T_ALL)) == NULL) {
        t_error("t_alloc of t_unitdata structure failed");
        exit(5);
    }
    if ((uderr = (struct t_uderr *)t_alloc(fd, T_UDERROR, T_ALL)) == NULL) {
        t_error("t_alloc of t_uderr structure failed");
        exit(6);
    }

    while (1) {
        if (t_rcvudata(fd, ud, &flags) < 0) {
            if (t_errno == TLOOK) {
                /*
                 * Error on previously sent datagram
                 */
                if (t_rcvuderr(fd, uderr) < 0) {
                    t_error("t_rcvuderr failed");
                    exit(7);
                }
                fprintf(stderr, "bad datagram, error = %d\n",
                    uderr->error);
                continue;
            }
            t_error("t_rcvudata failed");
            exit(8);
        }

        /*
         * Query() processes the request and places the
         * response in ud->udata.buf, setting ud->udata.len
         */
        query(ud);

        if (t_sndudata(fd, ud, 0) < 0) {
            t_error("t_sndudata failed");
            exit(9);
        }
    }
}

query()
{
    /* Merely a stub for simplicity */
}
```

C.5 Read/Write Client

The following code represents the connection-mode **read/write** client program described in Chapter 5. This client establishes a transport connection with a server. It then uses **cat** to retrieve the data sent by the server and write it to its standard output. This client will communicate with each of the connection-mode servers presented in the guide.

```
#include <stdio.h>
#include <tiuser.h>
#include <fcntl.h>
#include <stropts.h>

#define SRV_ADDR   1   /* server's well-known address */

main()
{
   int fd;
   int nbytes;
   int flags = 0;
   char buf[1024];
   struct t_call *sndcall;
   extern int t_errno;

   if ((fd = t_open("/dev/tivc", O_RDWR, NULL)) < 0) {
       t_error("t_open failed");
       exit(1);
   }

   if (t_bind(fd, NULL, NULL) < 0) {
       t_error("t_bind failed");
       exit(2);
   }

   /*
    * By assuming that the address is an integer value,
    * this program may not run over another protocol.
    */

   if ((sndcall = (struct t_call *)t_alloc(fd, T_CALL, T_ADDR)) == NULL) {
       t_error("t_alloc failed");
       exit(3);
   }

   sndcall->addr.len = sizeof(int);
   *(int *)sndcall->addr.buf = SRV_ADDR;

   if (t_connect(fd, sndcall, NULL) < 0) {
       t_error("t_connect failed for fd");
       exit(4);
   }

   if (ioctl(fd, I_PUSH, "tirdwr") < 0) {
       perror("I_PUSH of tirdwr failed");
       exit(5);
   }

   close(0);
   dup(fd);

   execl("/bin/cat", "/bin/cat", 0);

   perror("execl of /bin/cat failed");
   exit(6);
}
```

C.6 Event-Driven Server

The following code represents the connection-mode server program described in Chapter 6. This server manages multiple connect indications in an event-driven manner. Either connection-mode client presented earlier will communicate with this server.

```
#include <tiuser.h>
#include <fcntl.h>
#include <stdio.h>
#include <poll.h>
#include <stropts.h>
#include <signal.h>

#define NUM_FDS         1
#define MAX_CONN_IND    4
#define SRV_ADDR        1       /* server's well-known address */

int conn_fd;                    /* server connection here */
struct t_call *calls[NUM_FDS][MAX_CONN_IND];/* holds connect indications */
extern int t_errno;

main()
{
  struct pollfd pollfds[NUM_FDS];
  struct t_bind *bind;
  int i;

  /*
   * Only opening and binding one transport endpoint,
   * but more could be supported
   */
  if ((pollfds[0].fd = t_open("/dev/tivc", O_RDWR, NULL)) < 0) {
     t_error("t_open failed");
     exit(1);
  }

  if ((bind =(struct t_bind *)t_alloc(pollfds[0].fd, T_BIND, T_ALL))== NULL) {
     t_error("t_alloc of t_bind structure failed");
     exit(2);
  }
  bind->qlen = MAX_CONN_IND;
  bind->addr.len = sizeof(int);
  *(int *)bind->addr.buf = SRV_ADDR;

  if (t_bind(pollfds[0].fd, bind, bind) < 0) {
     t_error("t_bind failed");
     exit(3);
  }
```

```
    /*
     * Was the correct address bound?
     */
    if (*(int *)bind->addr.buf != SRV_ADDR) {
       fprintf(stderr, "t_bind bound wrong address\n");
       exit(4);
    }

    pollfds[0].events = POLLIN;

    while (1) {
       if (poll(pollfds, NUM_FDS, -1) < 0) {
          perror("poll failed");
          exit(5);
       }

       for (i = 0; i < NUM_FDS; i++) {

          switch (pollfds[i].revents) {

          default:
             perror("poll returned error event");
             exit(6);

          case 0:
             continue;

          case POLLIN:
             do_event(i, pollfds[i].fd);
             service_conn_ind(i, pollfds[i].fd);
          }
       }
    }
}


do_event(slot, fd)
{
    struct t_discon *discon;
    int i;

    switch (t_look(fd)) {

    default:
       fprintf(stderr,"t_look returned an unexpected event\n");
       exit(7);
```

```
    case T_ERROR:
        fprintf(stderr,"t_look returned T_ERROR event\n");
        exit(8);

    case -1:
        t_error("t_look failed");
        exit(9);

    case 0:
        /* since POLLIN returned, this should not happen */
        fprintf(stderr,"t_look returned no event\n");
        exit(10);

    case T_LISTEN:
        /*
         * find free element in calls array
         */
        for (i = 0; i < MAX_CONN_IND; i++) {
            if (calls[slot][i] == NULL)
                break;
        }

        if ((calls[slot][i]=(struct t_call *)t_alloc(fd, T_CALL, T_ALL))==NULL)
        {
            t_error("t_alloc of t_call structure failed");
            exit(11);
        }

        if (t_listen(fd, calls[slot][i]) < 0) {
            t_error("t_listen failed");
            exit(12);
        }
        break;

    case T_DISCONNECT:
        discon = (struct t_discon *)t_alloc(fd, T_DIS, T_ALL);

        if (t_rcvdis(fd, discon) < 0) {
            t_error("t_rcvdis failed");
            exit(13);
        }

        /*
         * find call ind in array and delete it
         */
        for (i = 0; i < MAX_CONN_IND; i++) {
            if (discon->sequence == calls[slot][i]->sequence) {
                t_free(calls[slot][i], T_CALL);
                calls[slot][i] = NULL;
            }
        }
        t_free(discon, T_DIS);
        break;
    }
}
```

```
service_conn_ind(slot, fd)
{
  int i;

  for (i = 0; i < MAX_CONN_IND; i++) {
     if (calls[slot][i] == NULL)
        continue;

     if ((conn_fd = t_open("/dev/tivc", O_RDWR, NULL)) < 0) {
        t_error("open failed");
        exit(14);
     }
     if (t_bind(conn_fd, NULL, NULL) < 0) {
        t_error("t_bind failed");
        exit(15);
     }

     if (t_accept(fd, conn_fd, calls[slot][i]) < 0) {
        if (t_errno == TLOOK) {
           t_close(conn_fd);
           return;
        }
        t_error("t_accept failed");
        exit(16);
     }
     t_free(calls[slot][i], T_CALL);
     calls[slot][i] = NULL;

     run_server(fd);
  }
}


connrelease()
{
  /* conn_fd is global because needed here */
  if (t_look(conn_fd) == T_DISCONNECT) {
     fprintf(stderr, "connection aborted\n");
     exit(12);
  }

  /* else orderly release indication - normal exit */
  exit(0);
}
```

```
run_server(listen_fd)
int listen_fd;
{
  int nbytes;
  FILE *logfp;     /* file pointer to log file */
  char buf[1024];

  switch (fork()) {

  case -1:
     perror("fork failed");
     exit(20);

  default: /* parent */

     /* close conn_fd and then go up and listen again */
     if (t_close(conn_fd) < 0) {
        t_error("t_close failed for conn_fd");
        exit(21);
     }
     return;

  case 0:      /* child */

     /* close listen_fd and do service */
     if (t_close(listen_fd) < 0) {
        t_error("t_close failed for listen_fd");
        exit(22);
     }
     if ((logfp = fopen("logfile", "r")) == NULL) {
        perror("cannot open logfile");
        exit(23);
     }

     signal(SIGPOLL, connrelease);
     if (ioctl(conn_fd, I_SETSIG, S_INPUT) < 0) {
        perror("ioctl I_SETSIG failed");
        exit(24);
     }
     if (t_look(conn_fd) != 0) {/* was disconnect already there? */
        fprintf(stderr, "t_look returned unexpected event\n");
        exit(25);
     }

     while ((nbytes = fread(buf, 1, 1024, logfp)) > 0)
        if (t_snd(conn_fd, buf, nbytes, 0) < 0) {
           t_error("t_snd failed");
           exit(26);
        }

     if (t_sndrel(conn_fd) < 0) {
        t_error("t_sndrel failed");
        exit(27);
     }
     pause(); /* until orderly release indication arrives */
  }
}
```

Appendix D

NSL Manpages

D.1 Appendix D: (NSL) Manpages

This appendix contains the (NSL) manpages.

Name

intro - introduction to the Network Services library.

Description

This section contains sets of functions constituting the Network Services library. These sets provide protocol independent interfaces to networking services based on the service definitions of the OSI (Open Systems Interconnection) reference model. Application developers access the function sets that provide services at a particular level.

The function sets contained in the library are:

TRANSPORT LAYER INTERFACE (TLI)—provide the services of the OSI Transport Layer. These services provide reliable end-to-end data transmission using the services of an underlying network. Applications written using the TLI functions are independent of the underlying protocols. Declarations for these functions may be obtained from the **#include** file **<tiuser.h>**. The link editor *ld*(M) searches this library under the **-lnsl_s** option.

Definitions

netbuf

In the Network Services library, *netbuf* is a structure used in various Transport Layer Interface (TLI) functions to send and receive data and information. It contains the following members:

```
unsigned int maxlen;
unsigned int len;
char     *buf;
```

buf points to a user input and/or output buffer. *len* generally specifies the number of bytes contained in the buffer. If the structure is used for both input and output, the function will replace the user value of *len* on return.

maxlen generally has significance only when *buf* is used to receive output from the TLI function. In this case, it specifies the physical size of the buffer, the maximum value of *len* that can be set by the function. If *maxlen* is not large enough to hold the returned information, a TBUFOVFLW error will generally result. However, certain functions may return part of the data and not generate an error.

Name

t_accept - accept a connect request

SYNOPSIS

#include <tiuser.h>

int t_accept(fd, resfd, call)
int fd;
int resfd;
struct t_call *call;

Description

This function is issued by a transport user to accept a connect request. *fd* identifies the local transport endpoint where the connect indication arrived, *resfd* specifies the local transport endpoint where the connection is to be established, and *call* contains information required by the transport provider to complete the connection. *call* points to a *t_call* structure which contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

netbuf is described in *intro*(NSL). In *call*, *addr* is the address of the caller, *opt* indicates any protocol-specific parameters associated with the connection, *udata* points to any user data to be returned to the caller, and *sequence* is the value returned by *t_listen* that uniquely associates the response with a previously received connect indication.

A transport user may accept a connection on either the same, or on a different, local transport endpoint than the one on which the connect indication arrived. If the same endpoint is specified (that is, *resfd=fd*), the connection can be accepted unless the following condition is true: The user has received other indications on that endpoint but has not responded to them (with *t_accept* or *t_snddis*). For this condition, *t_accept* will fail and set *t_errno* to TBADF.

If a different transport endpoint is specified (*resfd*!=*fd*), the endpoint must be bound to a protocol address and must be in the T_IDLE state [see *t_getstate*(NSL)] before the *t_accept* is issued.

For both types of endpoints, *t_accept* will fail and set *t_errno* to TLOOK if there are indications (for example, a connect or disconnect) waiting to be received on that endpoint.

The values of parameters specified by *opt* and the syntax of those values are protocol specific. The *udata* argument enables the called transport user to send user data to the caller and the amount of user data must not exceed the limits supported by the transport provider as returned by *t_open* or *t_getinfo*. If the *len* [see *netbuf* in *intro*(NSL)] field of *udata* is zero, no data will be sent to the caller.

On failure, *t_errno* may be set to one of the following:

[TBADF] The specified file descriptor does not refer to a transport endpoint, or the user is illegally accepting a connection on the same transport endpoint on which the connect indication arrived.

[TOUTSTATE] The function was issued in the wrong sequence on the transport endpoint referenced by *fd*, or the transport endpoint referred to by *resfd* is not in the T_IDLE state.

[TACCES] The user does not have permission to accept a connection on the responding transport endpoint or use the specified options.

[TBADOPT] The specified options were in an incorrect format or contained illegal information.

[TBADDATA] The amount of user data specified was not within the bounds allowed by the transport provider.

[TBADSEQ] An invalid sequence number was specified.

[TLOOK] An asynchronous event has occurred on the transport endpoint referenced by *fd* and requires immediate attention.

[TNOTSUPPORT] This function is not supported by the underlying transport provider.

[TSYSERR] A system error has occurred during execution of this function.

See Also

intro(NSL), t_connect(NSL), t_getstate(NSL), t_listen(NSL), t_open(NSL), t_rcvconnect(NSL).
Network Programmer's Guide.

Diagnostics

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *t_errno* is set to indicate the error.

Name

t_alloc - allocate a library structure

Syntax

#include <tiuser.h>

char *t_alloc(fd, struct_type, fields)
int fd;
int struct_type;
int fields;

Description

The *t_alloc* function dynamically allocates memory for the various transport function argument structures as specified below. This function will allocate memory for the specified structure, and will also allocate memory for buffers referenced by the structure.

The structure to allocate is specified by *struct_type*, and can be one of the following:

T_BIND	struct t_bind
T_CALL	struct t_call
T_OPTMGMT	struct t_optmgmt
T_DIS	struct t_discon
T_UNITDATA	struct t_unitdata
T_UDERROR	struct t_uderr
T_INFO	struct t_info

where each of these structures may subsequently be used as an argument to one or more transport functions.

Each of the above structures, except T_INFO, contains at least one field of type *struct netbuf*. *netbuf* is described in *intro*(NSL). For each field of this type, the user may specify that the buffer for that field should be allocated as well. The *fields* argument specifies this option, where the argument is the bitwise-OR of any of the following:

T_ADDR The *addr* field of the *t_bind*, *t_call*, *t_unitdata*, or *t_uderr* structures.

T_OPT The *opt* field of the *t_optmgmt*, *t_call*, *t_unitdata*, or *t_uderr* structures.

T_UDATA The *udata* field of the *t_call*, *t_discon*, or *t_unitdata* structures.

T_ALL All relevant fields of the given structure.

For each field specified in *fields*, *t_alloc* will allocate memory for the buffer associated with the field, and initialize the *buf* pointer and *maxlen* [see *netbuf* in *intro*(NSL) for description of *buf* and *maxlen*] field accordingly. The length of the buffer allocated will be based on the same size information that is returned to the user on *t_open* and *t_getinfo*. Thus, *fd* must refer to the transport endpoint through which the newly allocated structure will be passed, so that the appropriate size information can be accessed. If the size value associated with any specified field is -1 or -2 (see *t_open* or *t_getinfo*), *t_alloc* will be unable to determine the size of the buffer to allocate and will fail, setting *t_errno* to TSYSERR and *errno* to EINVAL. For any field not specified in *fields*, *buf* will be set to NULL and *maxlen* will be set to zero.

Use of *t_alloc* to allocate structures will help ensure the compatibility of user programs with future releases of the transport interface.

On failure, *t_errno* may be set to one of the following:

[TBADF] The specified file descriptor does not refer to a transport endpoint.

[TSYSERR] A system error has occurred during execution of this function.

See Also

intro(NSL), t_free(NSL), t_getinfo(NSL), t_open(NSL).
Network Programmer's Guide.

Diagnostics

On successful completion, *t_alloc* returns a pointer to the newly allocated structure. On failure, NULL is returned.

Name

t_bind - bind an address to a transport endpoint

Syntax

#include <tiuser.h>

int t_bind(fd, req, ret)
int fd;
struct t_bind *req;
struct t_bind *ret;

Description

This function associates a protocol address with the transport endpoint specified by *fd* and activates that transport endpoint. In connection mode, the transport provider may begin accepting or requesting connections on the transport endpoint. In connectionless mode, the transport user may send or receive data units through the transport endpoint.

The *req* and *ret* arguments point to a *t_bind* structure containing the following members:

```
struct netbuf addr;
unsigned qlen;
```

netbuf is described in *intro*(NSL). The *addr* field of the *t_bind* structure specifies a protocol address and the *qlen* field is used to indicate the maximum number of outstanding connect indications.

req is used to request that an address, represented by the *netbuf* structure, be bound to the given transport endpoint. *len* [see *netbuf* in *intro*(NSL); also for *buf* and *maxlen*] specifies the number of bytes in the address and *buf* points to the address buffer. *maxlen* has no meaning for the *req* argument. On return, *ret* contains the address that the transport provider actually bound to the transport endpoint; this may be different from the address specified by the user in *req*. In *ret*, the user specifies *maxlen* which is the maximum size of the address buffer and *buf* which points to the buffer where the address is to be placed. On return, *len* specifies the number of bytes in the bound address and *buf* points to the bound address. If *maxlen* is not large enough to hold the returned address, an error will result.

If the requested address is not available, or if no address is specified in *req* (the *len* field of *addr* in *req* is zero) the transport provider will assign an appropriate address to be bound, and will return that address in the *addr* field of *ret*. The user can compare the addresses in *req* and *ret* to determine whether the transport provider bound the transport endpoint to a different address than that requested.

req may be NULL if the user does not wish to specify an address to be bound. Here, the value of *qlen* is assumed to be zero, and the transport provider must assign an address to the transport endpoint. Similarly, *ret* may be NULL if the user does not care what address was bound by the provider and is not interested in the negotiated value of *qlen*. It is valid to set *req* and *ret* to NULL for the same call, in which case the provider chooses the address to bind to the transport endpoint and does not return that information to the user.

The *qlen* field has meaning only when initializing a connection-mode service. It specifies the number of outstanding connect indications the transport provider should support for the given transport endpoint. An outstanding connect indication is one that has been passed to the transport user by the transport provider. A value of *qlen* greater than zero is only meaningful when issued by a passive transport user that expects other users to call it. The value of *qlen* will be negotiated by the transport provider and may be changed if the transport provider cannot support the specified number of outstanding connect indications. On return, the *qlen* field in *ret* will contain the negotiated value.

This function allows more than one transport endpoint to be bound to the same protocol address (however, the transport provider must support this capability also), but it is not allowable to bind more than one protocol address to the same transport endpoint. If a user binds more than one transport endpoint to the same protocol address, only one endpoint can be used to listen for connect indications associated with that protocol address. In other words, only one *t_bind* for a given protocol address may specify a value of *qlen* greater than zero. In this way, the transport provider can identify which transport endpoint should be notified of an incoming connect indication. If a user attempts to bind a protocol address to a second transport endpoint with a value of *qlen* greater than zero, the transport provider will assign another address to be bound to that endpoint. If a user accepts a connection on the transport endpoint that is being used as the listening endpoint, the bound protocol address will be found to be busy for the duration of that connection. No other transport endpoints may be bound for listening while that initial listening endpoint is in the data transfer phase. This will prevent more than one transport endpoint bound to the same protocol address from accepting connect indications.

On failure, *t_errno* may be set to one of the following:

[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TOUTSTATE]	The function was issued in the wrong sequence.
[TBADADDR]	The specified protocol address was in an incorrect format or contained illegal information.
[TNOADDR]	The transport provider could not allocate an address.
[TACCES]	The user does not have permission to use the specified address.
[TBUFOVFLW]	The number of bytes allowed for an incoming argument is not sufficient to store the value of that argument. The provider's state will change to T_IDLE and the information to be returned in *ret* will be discarded.
[TSYSERR]	A system error has occurred during execution of this function.

See Also

intro(NSL), t_open(NSL), t_optmgmt(NSL), t_unbind(NSL).
Network Programmer's Guide.

Diagnostics

The *t_bind* function returns 0 on success and -1 on failure, and *t_errno* is set to indicate the error.

Name

t_close - close a transport endpoint

Syntax

#include <tiuser.h>

int t_close(fd)
int fd;

Description

The *t_close* function informs the transport provider that the user is finished with the transport endpoint specified by *fd*, and frees any local library resources associated with the endpoint. In addition, *t_close* closes the file associated with the transport endpoint.

The *t_close* function should be called from the T_UNBND state [see *t_getstate* (NSL)]. However, this function does not check state information, so it may be called from any state to close a transport endpoint. If this occurs, the local library resources associated with the endpoint will be freed automatically. In addition, *close*(S) will be issued for that file descriptor; the close will be abortive if no other process has that file open, and will break any transport connection that may be associated with that endpoint.

On failure, *t_errno* may be set to the following:

[TBADF] The specified file descriptor does not refer to a transport endpoint.

See Also

t_getstate(NSL), t_open(NSL), t_unbind(NSL).
Network Programmer's Guide.

Diagnostics

The *t_close* function returns 0 on success and -1 on failure, and *t_errno* is set to indicate the error.

Name

t_connect - establish a connection with another transport user

Syntax

#include <tiuser.h>

int t_connect(fd, sndcall, rcvcall)
int fd;
struct t_call *sndcall;
struct t_call *rcvcall;

Description

This function enables a transport user to request a connection to the specified destination transport user. *fd* identifies the local transport endpoint where communication will be established, while *sndcall* and *rcvcall* point to a *t_call* structure which contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

sndcall specifies information needed by the transport provider to establish a connection, and *rcvcall* specifies information that is associated with the newly established connection.

netbuf is described in *intro*(NSL). In *sndcall*, *addr* specifies the protocol address of the destination transport user, *opt* presents any protocol-specific information that might be needed by the transport provider, *udata* points to optional user data that may be passed to the destination transport user during connection establishment, and *sequence* has no meaning for this function.

On return in *rcvcall*, *addr* returns the protocol address associated with the responding transport endpoint; *opt* presents any protocol-specific information associated with the connection; *udata* points to optional user data that may be returned by the destination transport user during connection establishment; and *sequence* has no meaning for this function.

The *opt* argument implies no structure on the options that may be passed to the transport provider. The transport provider is free to specify the structure of any options passed to it. These options are specific to the underlying protocol of the transport provider. The user may choose not to negotiate protocol options by setting the *len* field of *opt* to zero. In this case, the provider may use default options.

The *udata* argument enables the caller to pass user data to the destination transport user and receive user data from the destination user during connection establishment. However, the amount of user data must not exceed the limits supported by the transport provider as returned by *t_open*(NSL) or *t_getinfo*(NSL). If the *len* [see *netbuf* in *intro*(NSL)] field of *udata* is zero in *sndcall*, no data will be sent to the destination transport user.

On return, the *addr*, *opt*, and *udata* fields of *rcvcall* will be updated to reflect values associated with the connection. Thus, the *maxlen* [see *netbuf* in *intro*(NSL)] field of each argument must be set before issuing this function to indicate the maximum size of the buffer for each. However, *rcvcall* may be NULL, in which case no information is given to the user on return from *t_connect*.

By default, *t_connect* executes in synchronous mode, and will wait for the destination user's response before returning control to the local user. A successful return (that is, a return value of zero) indicates that the requested connection has been established. However, if O_NDELAY is set (via *t_open* or *fcntl*), *t_connect* executes in asynchronous mode. In this case, the call will not wait for the remote user's response, but will return control immediately to the local user and return -1 with *t_errno* set to TNODATA to indicate that the connection has not yet been established. In this way, the function simply initiates the connection establishment procedure by sending a connect request to the destination transport user.

On failure, *t_errno* may be set to one of the following:

[TBADF] The specified file descriptor does not refer to a transport endpoint.

[TOUTSTATE] The function was issued in the wrong sequence.

[TNODATA] O_NDELAY was set, so the function successfully initiated the connection establishment procedure, but did not wait for a response from the remote user.

[TBADADDR] The specified protocol address was in an incorrect format or contained illegal information.

[TBADOPT] The specified protocol options were in an incorrect format or contained illegal information.

[TBADDATA] The amount of user data specified was not within the bounds allowed by the transport provider.

[TACCES]	The user does not have permission to use the specified address or options.
[TBUFOVFLW]	The number of bytes allocated for an incoming argument is not sufficient to store the value of that argument. If executed in synchronous mode, the provider's state, as seen by the user, changes to T_DATAXFER, and the connect indication information to be returned in *rcvcall* is discarded.
[TLOOK]	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.

See Also

intro(NSL), t_accept(NSL), t_getinfo(NSL), t_listen(NSL), t_open(NSL), t_optmgmt(NSL), t_rcvconnect(NSL).

Network Programmer's Guide.

Diagnostics

The *t_connect* function returns 0 on success and -1 on failure, and *t_errno* is set to indicate the error.

Name

t_error - produce error message

Syntax

#include <tiuser.h>

void t_error(errmsg)
char *errmsg;
extern int t_errno;
extern char *t_errlist[];
extern int t_nerr;

Description

t_error produces a message on the standard error output which describes the last error encountered during a call to a transport function. The argument string *errmsg* is a user-supplied error message that gives context to the error.

t_error prints the user-supplied error message followed by a colon and the standard transport function error message for the current value contained in *t_errno*. If *t_errno* is TSYSERR, *t_error* will also print the standard error message for the current value contained in *errno* [see *intro*(S)].

t_errlist is the array of message strings, to allow user message formatting. *t_errno* can be used as an index into this array to retrieve the error message string (without a terminating newline). *t_nerr* is the maximum index value for the *t_errlist* array.

t_errno is set when an error occurs and is not cleared on subsequent successful calls.

Example

If a *t_connect* function fails on transport endpoint *fd2* because a bad address was given, the following call might follow the failure:

```
t_error("t_connect failed on fd2");
```

The diagnostic message would print as:

```
t_connect failed on fd2: Incorrect transport address format
```

where "t_connect failed on fd2" tells the user which function failed on which transport endpoint, and "Incorrect transport address format" identifies the specific error that occurred.

See Also

Network Programmer's Guide.

Name

t_free - free a library structure

Syntax

#include <tiuser.h>

int t_free(ptr, struct_type)
char *ptr;
int struct_type;

Description

The *t_free* function frees memory previously allocated by *t_alloc*. This function will free memory for the specified structure and will also free memory for buffers referenced by the structure.

ptr points to one of the six structure types described for *t_alloc*, and *struct_type* identifies the type of that structure which can be one of the following:

T_BIND	struct t_bind
T_CALL	struct t_call
T_OPTMGMT	struct t_optmgmt
T_DIS	struct t_discon
T_UNITDATA	struct t_unitdata
T_UDERROR	struct t_uderr
T_INFO	struct t_info

where each of these structures is used as an argument to one or more transport functions.

The *t_free* function will check the *addr*, *opt*, and *udata* fields of the given structure (as appropriate) and free the buffers pointed to by the *buf* field of the *netbuf* [see *intro*(NSL)] structure. If *buf* is NULL, *t_free* will not attempt to free memory. After all buffers are freed, *t_free* will free the memory associated with the structure pointed to by *ptr*.

Undefined results will occur if *ptr* or any of the *buf* pointers points to a block of memory that was not previously allocated by *t_alloc*.

On failure, *t_errno* may be set to the following:

[TSYSERR] A system error has occurred during execution of this function.

See Also

intro(NSL), t_alloc(NSL).
Network Programmer's Guide.

Diagnostics

The *t_free* function returns 0 on success and -1 on failure, and *t_errno* is set to indicate the error.

Name

t_getinfo - get protocol-specific service information

Syntax

```
#include <tiuser.h>

int t_getinfo(fd, info)
int fd;
struct t_info *info;
```

Description

This function returns the current characteristics of the underlying transport protocol associated with file descriptor *fd*. The *info* structure is used to return the same information returned by *t_open*. This function enables a transport user to access this information during any phase of communication.

This argument points to a *t_info* structure which contains the following members:

```
long addr;      /* max size of the transport protocol address */
long options;   /* max number of bytes of protocol-specific options */
long tsdu;      /* max size of a transport service data unit (TSDU) */
long etsdu;     /* max size of an expedited transport service data
                   unit (ETSDU) */
long connect;   /* max amount of data allowed on connection establishment
                   functions */
long discon;    /* max amount of data allowed on t_snddis and
                   t_rcvdis
                   functions */
long servtype;  /* service type supported by the transport provider */
```

The values of the fields have the following meanings:

addr — A value greater than or equal to zero indicates the maximum size of a transport protocol address; a value of -1 specifies that there is no limit on the address size; and a value of -2 specifies that the transport provider does not provide user access to transport protocol addresses.

options A value greater than or equal to zero indicates the maximum number of bytes of protocol-specific options supported by the provider; a value of -1 specifies that there is no limit on the option size; and a value of -2 specifies that the transport provider does not support user-settable options.

tsdu A value greater than zero specifies the maximum size of a transport service data unit (TSDU); a value of zero specifies that the transport provider does not support the concept of TSDU, although it does support the sending of a data stream with no logical boundaries preserved across a connection; a value of -1 specifies that there is no limit on the size of a TSDU; and a value of -2 specifies that the transfer of normal data is not supported by the transport provider.

etsdu A value greater than zero specifies the maximum size of an expedited transport service data unit (ETSDU); a value of zero specifies that the transport provider does not support the concept of ETSDU, although it does support the sending of an expedited data stream with no logical boundaries preserved across a connection; a value of -1 specifies that there is no limit on the size of an ETSDU; and a value of -2 specifies that the transfer of expedited data is not supported by the transport provider.

connect A value greater than or equal to zero specifies the maximum amount of data that may be associated with connection establishment functions; a value of -1 specifies that there is no limit on the amount of data sent during connection establishment; and a value of -2 specifies that the transport provider does not allow data to be sent with connection establishment functions.

discon A value greater than or equal to zero specifies the maximum amount of data that may be associated with the *t_snddis* and *t_rcvdis* functions; a value of -1 specifies that there is no limit on the amount of data sent with these abortive release functions; and a value of -2 specifies that the transport provider does not allow data to be sent with the abortive release functions.

servtype This field specifies the service type supported by the transport provider, as described below.

If a transport user is concerned with protocol independence, the above sizes may be accessed to determine how large the buffers must be to hold each piece of information. Alternatively, the *t_alloc* function may be used to allocate these buffers. An error will result if a transport user exceeds the allowed data size on any function. The value of each field may change as a result of option negotiation, and *t_getinfo* enables a user to retrieve the current characteristics.

The *servtype* field of *info* may specify one of the following values on return:

T_COTS — The transport provider supports a connection-mode service but does not support the optional orderly release facility.

T_COTS_ORD — The transport provider supports a connection-mode service with the optional orderly release facility.

T_CLTS — The transport provider supports a connectionless-mode service. For this service type, *t_open* will return -2 for *etsdu*, *connect*, and *discon*.

On failure, *t_errno* may be set to one of the following:

[TBADF] — The specified file descriptor does not refer to a transport endpoint.

[TSYSERR] — A system error has occurred during execution of this function.

See Also

t_open(NSL).
Network Programmer's Guide.

Diagnostics

The *t_getinfo* function returns 0 on success and -1 on failure, and *t_errno* is set to indicate the error.

Name

t_getstate - get the current state

Syntax

#include <tiuser.h>

int t_getstate(fd)
int fd;

Description

The *t_getstate* function returns the current state of the provider associated with the transport endpoint specified by *fd*.

On failure, *t_errno* may be set to one of the following:

[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TSTATECHNG]	The transport provider is undergoing a state change.
[TSYSERR]	A system error has occurred during execution of this function.

See Also

t_open(NSL).
Network Programmer's Guide.

Diagnostics

The *t_getstate* function returns the current state on successful completion and -1 on failure, and *t_errno* is set to indicate the error. The current state may be one of the following:

T_UNBND	unbound
T_IDLE	idle
T_OUTCON	outgoing connection pending
T_INCON	incoming connection pending

T_DATAXFER	data transfer
T_OUTREL	outgoing orderly release (waiting for an orderly release indication)
T_INREL	incoming orderly release (waiting for an orderly release request)

If the provider is undergoing a state transition when *t_getstate* is called, the function will fail.

Name

t_listen - listen for a connect request

Synopsis

#include <tiuser.h>

int t_listen(fd, call)
int fd;
struct t_call *call;

Description

This function listens for a connect request from a calling transport user. *fd* identifies the local transport endpoint where connect indications arrive, and on return, *call* contains information describing the connect indication. *call* points to a *t_call* structure which contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

netbuf is described in *intro*(NSL). In *call*, *addr* returns the protocol address of the calling transport user; *opt* returns protocol-specific parameters associated with the connect request; *udata* returns any user data sent by the caller on the connect request; and *sequence* is a number that uniquely identifies the returned connect indication. The value of *sequence* enables the user to listen for multiple connect indications before responding to any of them.

Since this function returns values for the *addr*, *opt*, and *udata* fields of *call*, the *maxlen* [see *netbuf* in *intro*(NSL)] field of each must be set before issuing the *t_listen* to indicate the maximum size of the buffer for each.

By default, *t_listen* executes in synchronous mode and waits for a connect indication to arrive before returning to the user. However, if O_NDELAY is set (via *t_open* or *fcntl*), *t_listen* executes asynchronously, reducing to a poll for existing connect indications. If none are available, it returns -1 and sets *t_errno* to TNODATA.

On failure, *t_errno* may be set to one of the following:

[TBADF]	The specified file descriptor does not refer to a transport endpoint.

[TBUFOVFLW]	The number of bytes allocated for an incoming argument is not sufficient to store the value of that argument. The provider's state, as seen by the user, changes to T_INCON, and the connect indication information to be returned in *call* is discarded.
[TNODATA]	O_NDELAY was set, but no connect indications had been queued.
[TLOOK]	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.

See Also

intro(NSL), t_accept(NSL), t_bind(NSL), t_connect(NSL), t_open(NSL), t_rcvconnect(NSL).

Network Programmer's Guide.

Diagnostics

The *t_listen* function returns 0 on success and -1 on failure, and *t_errno* is set to indicate the error.

Warning

If a user issues *t_listen* in synchronous mode on a transport endpoint that was not bound for listening (that is, *qlen* was zero on *t_bind*), the call will wait forever because no connect indications will arrive on that endpoint.

Name

t_look - look at the current event on a transport endpoint

Syntax

#include <tiuser.h>

int t_look(fd)
int fd;

Description

This function returns the current event on the transport endpoint specified by *fd*. This function enables a transport provider to notify a transport user of an asynchronous event when the user is issuing functions in synchronous mode. Certain events require immediate notification of the user and are indicated by a specific error, TLOOK, on the current or next function to be executed.

This function also enables a transport user to poll a transport endpoint periodically for asynchronous events.

On failure, *t_errno* may be set to one of the following:

[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TSYSERR]	A system error has occurred during execution of this function.

See Also

t_open(NSL).
Network Programmer's Guide.

Diagnostics

Upon success, *t_look* returns a value that indicates which of the allowable events has occurred, or returns zero if no event exists. One of the following events is returned:

T_LISTEN	connection indication received
T_CONNECT	connect confirmation received

T_DATA	normal data received
T_EXDATA	expedited data received
T_DISCONNECT	disconnect received
T_ERROR	fatal error indication
T_UDERR	datagram error indication
T_ORDREL	orderly release indication

On failure, -1 is returned, and *t_errno* is set to indicate the error.

Name

t_open - establish a transport endpoint

Syntax

#include <tiuser.h>

int t_open(path, oflag, info)
char *path;
int oflag;
struct t_info *info;

Description

The *t_open* function must be called as the first step in the initialization of a transport endpoint. This function establishes a transport endpoint by opening a UNIX system file that identifies a particular transport provider (that is, transport protocol) and returning a file descriptor that identifies that endpoint. For example, opening the file **/dev/iso_cots** identifies an OSI connection-oriented transport layer protocol as the transport provider.

path points to the path name of the file to open, and *oflag* identifies any open flags [as in *open*(S)]. *t_open* returns a file descriptor that will be used by all subsequent functions to identify the particular local transport endpoint.

This function also returns various default characteristics of the underlying transport protocol by setting fields in the *t_info* structure. This argument points to a *t_info* which contains the following members:

```
long addr;      /* max size of the transport protocol address */
long options;   /* max number of bytes of protocol-specific
                   options */
long tsdu;      /* max size of a transport service data unit (TSDU) */
long etsdu;     /* max size of an expedited transport service data
                   unit (ETSDU) */
long connect;   /* max amount of data allowed on connection
                   establishment functions */
long discon;    /* max amount of data allowed on t_snddis and
                   t_rcvdis functions */
long servtype;  /* service type supported by the transport provider */
```

The values of the fields have the following meanings:

addr A value greater than or equal to zero indicates the maximum size of a transport protocol address; a value of -1 specifies that there is no limit on the address size; and a value of -2 specifies that the

transport provider does not provide user access to transport protocol addresses.

options A value greater than or equal to zero indicates the maximum number of bytes of protocol-specific options supported by the provider; a value of -1 specifies that there is no limit on the option size; and a value of -2 specifies that the transport provider does not support user-settable options.

tsdu A value greater than zero specifies the maximum size of a transport service data unit (TSDU); a value of zero specifies that the transport provider does not support the concept of TSDU, although it does support the sending of a data stream with no logical boundaries preserved across a connection; a value of -1 specifies that there is no limit on the size of a TSDU; and a value of -2 specifies that the transfer of normal data is not supported by the transport provider.

etsdu A value greater than zero specifies the maximum size of an expedited transport service data unit (ETSDU); a value of zero specifies that the transport provider does not support the concept of ETSDU, although it does support the sending of an expedited data stream with no logical boundaries preserved across a connection; a value of -1 specifies that there is no limit on the size of an ETSDU; and a value of -2 specifies that the transfer of expedited data is not supported by the transport provider.

connect A value greater than or equal to zero specifies the maximum amount of data that may be associated with connection establishment functions; a value of -1 specifies that there is no limit on the amount of data sent during connection establishment; and a value of -2 specifies that the transport provider does not allow data to be sent with connection establishment functions.

discon A value greater than or equal to zero specifies the maximum amount of data that may be associated with the *t_snddis* and *t_rcvdis* functions; a value of -1 specifies that there is no limit on the amount of data sent with these abortive release functions; and a value of -2 specifies that the transport provider does not allow data to be sent with the abortive release functions.

servtype This field specifies the service type supported by the transport provider, as described below.

If a transport user is concerned with protocol independence, the above sizes may be accessed to determine how large the buffers must be to hold each piece of information. Alternatively, the *t_alloc* function may be used to allocate these buffers. An error will result if a transport user exceeds the allowed data size on any function.

The *servtype* field of *info* may specify one of the following values on return:

T_COTS The transport provider supports a connection-mode service but does not support the optional orderly release facility.

T_COTS_ORD The transport provider supports a connection-mode service with the optional orderly release facility.

T_CLTS The transport provider supports a connectionless-mode service. For this service type, *t_open* will return -2 for *etsdu*, *connect*, and *discon*.

A single transport endpoint may support only one of the above services at one time.

If *info* is set to NULL by the transport user, no protocol information is returned by *t_open*.

On failure, *t_errno* may be set to the following:

[TSYSERR] A system error has occurred during execution of this function.

See Also

open(S) in the *XENIX Reference*.
Network Programmer's Guide.

Diagnostics

The *t_open* function returns a valid file descriptor on success and -1 on failure, and *t_errno* is set to indicate the error.

Name

t_optmgmt - manage options for a transport endpoint

Syntax

```
#include <tiuser.h>

int t_optmgmt(fd, req, ret)
int fd;
struct t_optmgmt *req;
struct t_optmgmt *ret;
```

Description

The *t_optmgmt* function enables a transport user to retrieve, verify, or negotiate protocol options with the transport provider. *fd* identifies a bound transport endpoint.

The *req* and *ret* arguments point to a *t_optmgmt* structure containing the following members:

```
struct netbuf opt;
long     flags;
```

The *opt* field identifies protocol options, and the *flags* field is used to specify the action to take with those options.

The options are represented by a *netbuf* [see *intro*(NSL); also for *len*, *buf*, and *maxlen*] structure in a manner similar to the address in *t_bind*. *req* is used to request a specific action of the provider and to send options to the provider. *len* specifies the number of bytes in the options, *buf* points to the options buffer, and *maxlen* has no meaning for the *req* argument. The transport provider may return options and flag values to the user through *ret*. For *ret*, *maxlen* specifies the maximum size of the options buffer and *buf* points to the buffer where the options are to be placed. On return, *len* specifies the number of bytes of options returned. *maxlen* has no meaning for the *req* argument, but must be set in the *ret* argument to specify the maximum number of bytes the options buffer can hold. The actual structure and content of the options is imposed by the transport provider.

The *flags* field of *req* can specify one of the following actions:

T_NEGOTIATE — This action enables the user to negotiate the values of the options specified in *req* with the transport provider. The provider will evaluate the requested options and negotiate the values, returning the negotiated values through *ret*.

T_CHECK This action enables the user to verify whether the options specified in *req* are supported by the transport provider. On return, the *flags* field of *ret* will have either T_SUCCESS or T_FAILURE set to indicate to the user whether the options are supported. These flags are only meaningful for the T_CHECK request.

T_DEFAULT This action enables a user to retrieve the default options supported by the transport provider into the *opt* field of *ret*. In *req*, the *len* field of *opt* must be zero, and the *buf* field may be NULL.

If issued as part of the connectionless-mode service, *t_optmgmt* may block due to flow control constraints. The function will not complete until the transport provider has processed all previously sent data units.

On failure, *t_errno* may be set to one of the following:

[TBADF] The specified file descriptor does not refer to a transport endpoint.

[TOUTSTATE] The function was issued in the wrong sequence.

[TACCES] The user does not have permission to negotiate the specified options.

[TBADOPT] The specified protocol options were in an incorrect format or contained illegal information.

[TBADFLAG] An invalid flag was specified.

[TBUFOVFLW] The number of bytes allowed for an incoming argument is not sufficient to store the value of that argument. The information to be returned in *ret* will be discarded.

[TSYSERR] A system error has occurred during execution of this function.

See Also

intro(NSL), t_getinfo(NSL), t_open(NSL).
Network Programmer's Guide.

Diagnostics

The *t_optmgmt* function returns 0 on success and -1 on failure, and *t_errno* is set to indicate the error.

Name

t_rcv - receive data or expedited data sent over a connection

Syntax

int t_rcv(fd, buf, nbytes, flags)
int fd;
char *buf;
unsigned nbytes;
int *flags;

Description

This function receives either normal or expedited data. *fd* identifies the local transport endpoint through which data will arrive; *buf* points to a receive buffer where user data will be placed; and *nbytes* specifies the size of the receive buffer. *flags* may be set on return from *t_rcv* and specifies optional flags as described below.

By default, *t_rcv* operates in synchronous mode and will wait for data to arrive if none is currently available. However, if O_NDELAY is set (via *t_open* or *fcntl*), *t_rcv* will execute in asynchronous mode and will fail if no data is available. (See TNODATA below.)

On return from the call, if T_MORE is set in *flags*, this indicates that there is more data and the current transport service data unit (TSDU) or expedited transport service data unit (ETSDU) must be received in multiple *t_rcv* calls. Each *t_rcv* with the T_MORE flag set indicates that another *t_rcv* must follow immediately to get more data for the current TSDU. The end of the TSDU is identified by the return of a *t_rcv* call with the T_MORE flag not set. If the transport provider does not support the concept of a TSDU as indicated in the *info* argument on return from *t_open* or *t_getinfo*, the T_MORE flag is not meaningful and should be ignored.

On return, the data returned is expedited data if T_EXPEDITED is set in *flags*. If the number of bytes of expedited data exceeds *nbytes*, *t_rcv* will set T_EXPEDITED and T_MORE on return from the initial call. Subsequent calls to retrieve the remaining ETSDU will not have T_EXPEDITED set on return. The end of the ETSDU is identified by the return of a *t_rcv* call with the T_MORE flag not set.

If expedited data arrives after part of a TSDU has been retrieved, receipt of the remainder of the TSDU will be suspended until the ETSDU has been processed. Only after the full ETSDU has been retrieved (T_MORE not set) will the remainder of the TSDU be available to the user.

On failure, *t_errno* may be set to one of the following:

[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TNODATA]	O_NDELAY was set, but no data is currently available from the transport provider.
[TLOOK]	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.

See Also

t_open(NSL), t_snd(NSL).
Network Programmer's Guide.

Diagnostics

On successful completion, *t_rcv* returns the number of bytes received, and it returns -1 on failure, and *t_errno* is set to indicate the error.

Name

t_rcvconnect - receive the confirmation from a connect request

Syntax

```
#include <tiuser.h>

int t_rcvconnect(fd, call)
int fd;
struct t_call *call;
```

Description

This function enables a calling transport user to determine the status of a previously sent connect request and is used in conjunction with *t_connect* to establish a connection in asynchronous mode. The connection will be established on successful completion of this function.

fd identifies the local transport endpoint where communication will be established, and *call* contains information associated with the newly established connection. *call* points to a *t_call* structure which contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

netbuf is described in *intro*(NSL). In *call*, *addr* returns the protocol address associated with the responding transport endpoint, *opt* presents any protocol-specific information associated with the connection, *udata* points to optional user data that may be returned by the destination transport user during connection establishment, and *sequence* has no meaning for this function.

The *maxlen* [see *netbuf* in *intro*(NSL)] field of each argument must be set before issuing this function to indicate the maximum size of the buffer for each. However, *call* may be NULL, in which case no information is given to the user on return from *t_rcvconnect*. By default, *t_rcvconnect* executes in synchronous mode and waits for the connection to be established before returning. On return, the *addr*, *opt*, and *udata* fields reflect values associated with the connection.

If O_NDELAY is set (via *t_open* or *fcntl*), *t_rcvconnect* executes in asynchronous mode and reduces to a poll for existing connect confirmations. If none are available, *t_rcvconnect* fails and returns immediately without waiting for the connection to be established. (See TNODATA below.) *t_rcvconnect* must be re-issued at a later time to complete the connection establishment phase and retrieve the information returned in *call*.

On failure, *t_errno* may be set to one of the following:

[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TBUFOVFLW]	The number of bytes allocated for an incoming argument is not sufficient to store the value of that argument and the connect information to be returned in *call* will be discarded. The provider's state, as seen by the user, will be changed to DATAXFER.
[TNODATA]	O_NDELAY was set, but a connect confirmation has not yet arrived.
[TLOOK]	An asynchronous event has occurred on this transport connection and requires immediate attention.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.

See Also

intro(NSL), t_accept(NSL), t_bind(NSL), t_connect(NSL), t_listen(NSL), t_open(NSL).

Network Programmer's Guide.

Diagnostics

t_rcvconnect returns 0 on success and -1 on failure, and *t_errno* is set to indicate the error.

Name

t_rcvdis - retrieve information from disconnect

Syntax

#include <tiuser.h>

t_rcvdis(fd, discon)
int fd;
struct t_discon *discon;

Description

This function is used to identify the cause of a disconnect, and to retrieve any user data sent with the disconnect. *fd* identifies the local transport endpoint where the connection existed, and *discon* points to a *t_discon* structure containing the following members:

```
struct netbuf udata;
int reason;
int sequence;
```

netbuf is described in *intro*(NSL). *reason* specifies the reason for the disconnect through a protocol-dependent reason code, *udata* identifies any user data that was sent with the disconnect, and *sequence* may identify an outstanding connect indication with which the disconnect is associated. *sequence* is only meaningful when *t_rcvdis* is issued by a passive transport user who has executed one or more *t_listen* functions and is processing the resulting connect indications. If a disconnect indication occurs, *sequence* can be used to identify which of the outstanding connect indications is associated with the disconnect.

If a user does not care if there is incoming data and does not need to know the value of *reason* or *sequence*, *discon* may be NULL and any user data associated with the disconnect will be discarded. However, if a user has retrieved more than one outstanding connect indication (via *t_listen*) and *discon* is NULL, the user will be unable to identify with which connect indication the disconnect is associated.

On failure, *t_errno* may be set to one of the following:

[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TNODIS]	No disconnect indication currently exists on the specified transport endpoint.

[TBUFOVFLW]	The number of bytes allocated for incoming data is not sufficient to store the data. The provider's state, as seen by the user, will change to T_IDLE, and the disconnect indication information to be returned in *discon* will be discarded.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.

See Also

intro(NSL), t_connect(NSL), t_listen(NSL), t_open(NSL), t_snddis(NSL).

Network Programmer's Guide.

Diagnostics

The *t_rcvdis* function returns 0 on success and -1 on failure, and *t_errno* is set to indicate the error.

Name

t_rcvrel - acknowledge receipt of an orderly release indication

Syntax

#include <tiuser.h>

t_rcvrel(fd)
int fd;

Description

This function is used to acknowledge receipt of an orderly release indication. *fd* identifies the local transport endpoint where the connection exists. After receipt of this indication, the user may not attempt to receive more data because such an attempt will block forever. However, the user may continue to send data over the connection if *t_sndrel* has not been issued by the user.

This function is an optional service of the transport provider, and is only supported if the transport provider returned service type T_COTS_ORD on *t_open* or *t_getinfo*.

On failure, *t_errno* may be set to one of the following:

[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TNOREL]	No orderly release indication currently exists on the specified transport endpoint.
[TLOOK]	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.

See Also

t_open(NSL), t_sndrel(NSL).
Network Programmer's Guide.

Diagnostics

The *t_rcvrel* function returns 0 on success and -1 on failure, and *t_errno* is set to indicate the error.

Name

t_rcvudata - receive a data unit

Syntax

#include <tiuser.h>

int t_rcvudata(fd, unitdata, flags)
int fd;
struct t_unitdata *unitdata;
int *flags;

Description

This function is used in connectionless mode to receive a data unit from another transport user. *fd* identifies the local transport endpoint through which data will be received, *unitdata* holds information associated with the received data unit, and *flags* is set on return to indicate that the complete data unit was not received. *unitdata* points to a *t_unitdata* structure containing the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
```

The *maxlen* [see *netbuf* in *intro*(NSL)] field of *addr*, *opt*, and *udata* must be set before issuing this function to indicate the maximum size of the buffer for each.

On return from this call, *addr* specifies the protocol address of the sending user, *opt* identifies protocol-specific options that were associated with this data unit, and *udata* specifies the user data that was received.

By default, *t_rcvudata* operates in synchronous mode and will wait for a data unit to arrive if none is currently available. However, if O_NDELAY is set (via *t_open* or *fcntl*), *t_rcvudata* will execute in asynchronous mode and will fail if no data units are available.

If the buffer defined in the *udata* field of *unitdata* is not large enough to hold the current data unit, the buffer will be filled and T_MORE will be set in *flags* on return to indicate that another *t_rcvudata* should be issued to retrieve the rest of the data unit. Subsequent *t_rcvudata* call(s) will return zero for the length of the address and options until the full data unit has been received.

On failure, *t_errno* may be set to one of the following:

[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TNODATA]	O_NDELAY was set, but no data units are currently available from the transport provider.
[TBUFOVFLW]	The number of bytes allocated for the incoming protocol address or options is not sufficient to store the information. The unit data information to be returned in *unitdata* will be discarded.
[TLOOK]	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.

See Also

intro(NSL), t_rcvuderr(NSL), t_sndudata(NSL).
Network Programmer's Guide.

Diagnostics

The *t_rcvudata* function returns 0 on successful completion and -1 on failure, and *t_errno* is set to indicate the error.

Name

t_rcvuderr - receive a unit data error indication

Syntax

#include <tiuser.h>

int t_rcvuderr(fd, uderr)
int fd;
struct t_uderr *uderr;

Description

This function is used in connectionless mode to receive information concerning an error on a previously sent data unit, and should only be issued following a unit data error indication. It informs the transport user that a data unit with a specific destination address and protocol options produced an error. *fd* identifies the local transport endpoint through which the error report will be received, and *uderr* points to a *t_uderr* structure containing the following members:

```
struct netbuf addr;
struct netbuf opt;
long    error;
```

netbuf is described in *intro*(NSL). The *maxlen* [see *netbuf* in *intro*(NSL)] field of *addr* and *opt* must be set before issuing this function to indicate the maximum size of the buffer for each.

On return from this call, the *addr* structure specifies the destination protocol address of the erroneous data unit; the *opt* structure identifies protocol-specific options that were associated with the data unit; and *error* specifies a protocol-dependent error code.

If the user does not care to identify the data unit that produced an error, *uderr* may be set to NULL and *t_rcvuderr* will simply clear the error indication without reporting any information to the user.

On failure, *t_errno* may be set to one of the following:

[TBADF] The specified file descriptor does not refer to a transport endpoint.

[TNOUDERR] No unit data error indication currently exists on the specified transport endpoint.

[TBUFOVFLW] The number of bytes allocated for the incoming protocol address or options is not sufficient to store the information. The unit data error

	information to be returned in *uderr* will be discarded.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.

See Also

intro(NSL), t_rcvudata(NSL), t_sndudata(NSL).
Network Programmer's Guide.

Diagnostics

The *t_rcvuderr* function returns 0 on successful completion and -1 on failure, and *t_errno* is set to indicate the error.

Name

t_snd - send data or expedited data over a connection

Syntax

#include <tiuser.h>

int t_snd(fd, buf, nbytes, flags)
int fd;
char *buf;
unsigned nbytes;
int flags;

Description

This function is used to send either normal or expedited data. *fd* identifies the local transport endpoint over which data should be sent, *buf* points to the user data, *nbytes* specifies the number of bytes of user data to be sent, and *flags* specifies any optional flags described below.

By default, *t_snd* operates in synchronous mode and may wait if flow control restrictions prevent the data from being accepted by the local transport provider at the time the call is made. However, if O_NDELAY is set (via *t_open* or *fcntl*), *t_snd* will execute in asynchronous mode, and will fail immediately if there are flow control restrictions.

Even when there are no flow control restrictions, *t_snd* will wait if STREAMS internal resources are not available, regardless of the state of O_NDELAY.

On successful completion, *t_snd* returns the number of bytes accepted by the transport provider. Normally this will equal the number of bytes specified in *nbytes*. However, if O_NDELAY is set, it is possible that only part of the data will be accepted by the transport provider. In this case, *t_snd* will set T_MORE for the data that was sent (see below) and will return a value less than *nbytes*. If *nbytes* is zero, no data will be passed to the provider and *t_snd* will return zero.

If T_EXPEDITED is set in *flags*, the data will be sent as expedited data, and will be subject to the interpretations of the transport provider.

If T_MORE is set in *flags*, or is set as described above, an indication is sent to the transport provider that the transport service data unit (TSDU) or expedited transport service data unit (ETSDU) is being sent through multiple *t_snd* calls. Each *t_snd* with the T_MORE flag set indicates that another *t_snd* will follow with more data for the current TSDU. The end of the TSDU (or ETSDU) is identified by a *t_snd* call

with the T_MORE flag not set. Use of T_MORE enables a user to break up large logical data units without losing the boundaries of those units at the other end of the connection. The flag implies nothing about how the data is packaged for transfer below the transport interface. If the transport provider does not support the concept of a TSDU as indicated in the *info* argument on return from *t_open* or *t_getinfo*, the T_MORE flag is not meaningful and should be ignored.

The size of each TSDU or ETSDU must not exceed the limits of the transport provider as returned by *t_open* or *t_getinfo*. If the size is exceeded, a TSYSERR with system error EPROTO will occur. However, the *t_snd* may not fail because EPROTO errors may not be reported immediately. In this case, a subsequent call that accesses the transport endpoint will fail with the associated TSYSERR.

If *t_snd* is issued from the T_IDLE state, the provider may silently discard the data. If *t_snd* is issued from any state other than T_DATAXFER, T_INREL or T_IDLE, the provider will generate a TSYSERR with system error EPROTO (which may be reported in the manner described above).

On failure, *t_errno* may be set to one of the following:

[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TFLOW]	O_NDELAY was set, but the flow control mechanism prevented the transport provider from accepting data at this time.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error [see *intro*(S)] has been detected during execution of this function.

See Also

t_open(NSL), t_rcv(NSL).
Network Programmer's Guide.

Diagnostics

On successful completion, *t_snd* returns the number of bytes accepted by the transport provider, and it returns -1 on failure and *t_errno* is set to indicate the error.

Name

t_snddis - send user-initiated disconnect request

Syntax

#include <tiuser.h>

int t_snddis(fd, call)
int fd;
struct t_call *call;

Description

This function is used to initiate an abortive release on an already established connection or to reject a connect request. *fd* identifies the local transport endpoint of the connection, and *call* specifies information associated with the abortive release. *call* points to a *t_call* structure which contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

netbuf is described in *intro*(NSL). The values in *call* have different semantics, depending on the context of the call to *t_snddis*. When rejecting a connect request, *call* must be non-NULL and contain a valid value of *sequence* to uniquely identify the rejected connect indication to the transport provider. The *addr* and *opt* fields of *call* are ignored. In all other cases, *call* need only be used when data is being sent with the disconnect request. The *addr*, *opt*, and *sequence* fields of the *t_call* structure are ignored. If the user does not wish to send data to the remote user, the value of *call* may be NULL.

udata specifies the user data to be sent to the remote user. The amount of user data must not exceed the limits supported by the transport provider as returned by *t_open* or *t_getinfo*. If the *len* field of *udata* is zero, no data will be sent to the remote user.

On failure, *t_errno* may be set to one of the following:

[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TOUTSTATE]	The function was issued in the wrong sequence. The transport provider's outgoing queue may be flushed, so data may be lost.

[TBADDATA]	The amount of user data specified was not within the bounds allowed by the transport provider. The transport provider's outgoing queue will be flushed, so data may be lost.
[TBADSEQ]	An invalid sequence number was specified, or a NULL call structure was specified when rejecting a connect request. The transport provider's outgoing queue will be flushed, so data may be lost.
[TLOOK]	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.

See Also

intro(NSL), t_connect(NSL), t_getinfo(NSL), t_listen(NSL), t_open(NSL).

Network Programmer's Guide.

Diagnostics

The *t_snddis* function returns 0 on success and -1 on failure, and *t_errno* is set to indicate the error.

Name

t_sndrel - initiate an orderly release

Syntax

#include <tiuser.h>

int t_sndrel(fd)
int fd;

Description

This function is used to initiate an orderly release of a transport connection and indicates to the transport provider that the transport user has no more data to send. *fd* identifies the local transport endpoint where the connection exists. After issuing *t_sndrel*, the user may not send any more data over the connection. However, a user may continue to receive data if an orderly release indication has been received.

This function is an optional service of the transport provider and is only supported if the transport provider returned service type T_COTS_ORD on *t_open* or *t_getinfo*.

On failure, *t_errno* may be set to one of the following:

[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TFLOW]	O_NDELAY was set, but the flow control mechanism prevented the transport provider from accepting the function at this time.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.

See Also

t_open(NSL), t_rcvrel(NSL).
Network Programmer's Guide.

Diagnostics

The *t_sndrel* function returns 0 on success and -1 on failure, and *t_errno* is set to indicate the error.

Name

t_sndudata - send a data unit

Syntax

#include <tiuser.h>

int t_sndudata(fd, unitdata)
int fd;
struct t_unitdata *unitdata;

Description

This function is used in connectionless mode to send a data unit to another transport user. *fd* identifies the local transport endpoint through which data will be sent, and *unitdata* points to a *t_unitdata* structure containing the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
```

netbuf is described in *intro*(NSL). In *unitdata*, *addr* specifies the protocol address of the destination user, *opt* identifies protocol-specific options that the user wants associated with this request, and *udata* specifies the user data to be sent. The user may choose not to specify what protocol options are associated with the transfer by setting the *len* field of *opt* to zero. In this case, the provider may use default options.

If the *len* field of *udata* is zero, no data unit will be passed to the transport provider; *t_sndudata* will not send zero-length data units.

By default, *t_sndudata* operates in synchronous mode and may wait if flow control restrictions prevent the data from being accepted by the local transport provider at the time the call is made. However, if O_NDELAY is set (via *t_open* or *fcntl*), *t_sndudata* will execute in asynchronous mode and will fail under such conditions.

If *t_sndudata* is issued from an invalid state, or if the amount of data specified in *udata* exceeds the TSDU size as returned by *t_open* or *t_getinfo*, the provider will generate an EPROTO protocol error. (See TSYSERR below.)

On failure, *t_errno* may be set to one of the following:

[TBADF] The specified file descriptor does not refer to a transport endpoint.

[TFLOW]	O_NDELAY was set, but the flow control mechanism prevented the transport provider from accepting data at this time.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.

See Also

intro(NSL), t_rcvudata(NSL), t_rcvuderr(NSL).
Network Programmer's Guide.

Diagnostics

The *t_sndudata* function returns 0 on successful completion and -1 on failure, and *t_errno* is set to indicate the error.

Name

t_sync - synchronize transport library

Syntax

#include <tiuser.h>

int t_sync(fd)
int fd;

Description

For the transport endpoint specified by *fd*, *t_sync* synchronizes the data structures managed by the transport library with information from the underlying transport provider. In doing so, it can convert a raw file descriptor [obtained via *open*(S), *dup*(S), or as a result of a *fork*(S) and *exec*(S)] to an initialized transport endpoint, assuming that file descriptor referenced a transport provider. This function also allows two cooperating processes to synchronize their interaction with a transport provider.

For example, if a process *forks* a new process and issues an *exec*, the new process must issue a *t_sync* to build the private library data structure associated with a transport endpoint and to synchronize the data structure with the relevant provider information.

It is important to remember that the transport provider treats all users of a transport endpoint as a single user. If multiple processes are using the same endpoint, they should coordinate their activities so as not to violate the state of the provider. *t_sync* returns the current state of the provider to the user, thereby enabling the user to verify the state before taking further action. This coordination is only valid among cooperating processes; it is possible that a process or an incoming event could change the provider's state after a *t_sync* is issued.

If the provider is undergoing a state transition when *t_sync* is called, the function will fail.

On failure, *t_errno* may be set to one of the following:

[TBADF]	The specified file descriptor is a valid open file descriptor but does not refer to a transport endpoint.
[TSTATECHNG]	The transport provider is undergoing a state change.

[TSYSERR]	A system error has occurred during execution of this function.

See Also

dup(S), exec(S), fork(S), open(S) in the *XENIX Reference*.
Network Programmer's Guide.

Diagnostics

The *t_sync* function returns the state of the transport provider on successful completion and -1 on failure, and *t_errno* is set to indicate the error. The state returned may be one of the following:

T_UNBND	unbound
T_IDLE	idle
T_OUTCON	outgoing connection pending
T_INCON	incoming connection pending
T_DATAXFER	data transfer
T_OUTREL	outgoing orderly release (waiting for an orderly release indication)
T_INREL	incoming orderly release (waiting for an orderly release request).

Name

t_unbind - disable a transport endpoint

Syntax

#include <tiuser.h>

int t_unbind(fd)
int fd;

Description

The *t_unbind* function disables the transport endpoint specified by *fd*, which was previously bound by *t_bind* (NSL). On completion of this call, no further data or events destined for this transport endpoint will be accepted by the transport provider.

On failure, *t_errno* may be set to one of the following:

[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TOUTSTATE]	The function was issued in the wrong sequence.
[TLOOK]	An asynchronous event has occurred on this transport endpoint.
[TSYSERR]	A system error has occurred during execution of this function.

See Also

t_bind(NSL).
Network Programmer's Guide.

Diagnostics

The *t_unbind* function returns 0 on success and -1 on failure, and *t_errno* is set to indicate the error.