ULTRIX

digital

Guide to the X/Open Transport Interface

ULTRIX

Guide to X/Open Transport Interface

This manual contains information on writing network applications using the X/Open Transport Interface. It describes the system calls and subroutines used with the X/Open Transport Interface.

The following are trademarks of Digital Equipment Corporation:

digital	DECUS	ULTRIX Worksystem Software
	DECwindows	UNIBUS
CDA	DTIF	VAX
DDIF	MASSBUS	VAXstation
DDIS	MicroVAX	VMS
DEC	Q-bus	VMS/ULTRIX Connection
DECnet	ULTRIX	VT
DECstation	ULTRIX Mail Connection	XUI

UNIX is a registered trademark of AT&T in the USA and other countries.

X/Open is a trademark of X/Open Company Ltd.

Contents

2 Connection-Mode Service

3 Connectionless-Mode Service

4 Advanced Topics

A State Transitions

B Guidelines for Writing Protocol-Independent Software

C Migrating from Socket-Based Software to XTI-Based Software

D Connection-Mode Programming Examples

E Connectionless-Mode Programming Examples

Glossary

Examples

Figures

Tables

About This Manual

This guide contains information on the X/Open Transport Interface (XTI) with information necessary for developing network application programs on the ULTRIX operating system. The manual also contains information on migrating from socket-based software to the XTI-based software.

Audience

This guide is intended for experienced programmers who want to write network application programs using the X/Open Transport Interface. Readers should be familiar with the C programming language and ULTRIX networking concepts.

Organization

This guide consists of four chapters and five appendixes:

Chapter 1: Overview of the Transport Service Interface
This chapter provides a high level overview of the transport service interface (XTI), that supports the transfer of data between two user processes: transport user and transport provider.

Chapter 2: Connection-Mode Service
This chapter describes the connection-mode service of the transport service interface. The client-server paradigm is used to describe the connection-mode service.

Chapter 3: Connectionless-Mode Service
This chapter describes the connectionless-mode service of the transport service interface. The connectionless-mode service is used for short-term request/response interactions.

Chapter 4: Advanced Topics
This chapter describes: The characteristics associated with a transport endpoint that can be changed after an endpoint is opened. How memory resources can be managed. Choosing a mode of execution for an application. Reporting events to an application. Using the two levels of error reporting.

Appendix A: States and Events in XTI
This appendix contains tables that list the possible states of the transport provider as seen by the transport user, the incoming and outgoing events that may occur on any connection, and identifies the allowable sequence of functions.

Appendix B: Guidelines for Writing Protocol-Independent Software
This appendix describes how applications can be written to run over several transport providers without significant changes.

Appendix C: Migrating from Socket-based Software to XTI-based Software
This appendix describes how to migrate a program that uses sockets to a program that uses the XTI interface.

Appendix D: Connection-Mode Programming Code Examples
This appendix contains the entire connection-mode programming code from which the examples used in Chapter 2 are taken.

Appendix E: Connectionless-Mode Programming Code Examples
This appendix contains the connectionless-mode programming code from which the examples used in Chapter 3 were taken.

Related Documents

You should have available the documents in the ULTRIX documentation set, including the *ULTRIX Reference Pages*, appropriate C programming documentation, and the *Guide to Network Programming*.

Conventions

macro	In text, bold type is used to introduce new terms.
{	}	In syntax descriptions and function definitions, braces enclose lists from which one item must be chosen. Vertical bars are used to separate items.
cat(1)	Cross-references to the *ULTRIX Reference Pages* include the appropriate section number in parentheses. For example, a reference to cat(1) indicates that you can find the material on the cat command in Section 1 of the reference pages.
system output	This typeface is used in interactive examples to indicate system output and also in code examples and other screen displays. In text, this typeface is used to indicate the exact name of a command, option, partition, pathname, directory, or file.
rlogin	In syntax descriptions and function definitions, this typeface is used to indicate terms that you must type exactly as shown.
UPPERCASE lowercase	The ULTRIX system differentiates between lowercase and uppercase characters. Literal strings that appear in text, examples, syntax descriptions, and function definitions must be typed exactly as shown.
filename	In examples, syntax descriptions, and function definitions, italics are used to indicate variable values; and in text, to give references to other documents.
. . .	A vertical ellipsis indicates that a portion of an example that would normally be present is not shown.

New and Changed Information

This is a new manual.

Overview of the Transport Service Interface 1

This chapter provides a high-level overview of the transport service interface, which supports the transfer of data between two user processes: transport user and transport provider. Figure 1-1 illustrates the transport service interface.

Figure 1-1: Transport Service Interface
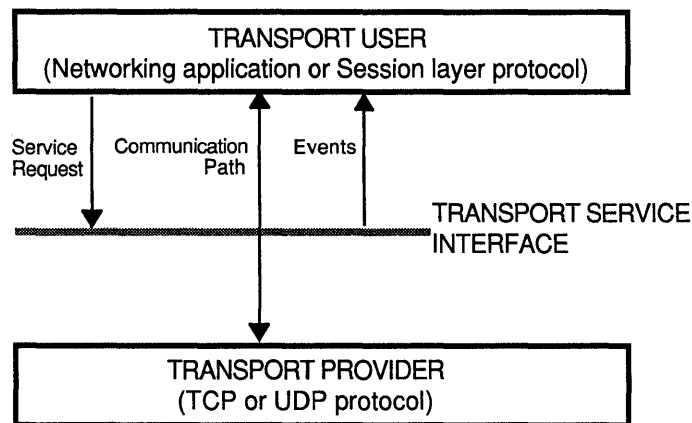
ZK-0098U-R

The transport provider is the entity that provides the services of the transport service interface, and the transport user is the entity that requires these services. Examples of transport providers are Transport Control Protocol (TCP) and User Datagram Protocol (UDP). A transport user may be a networking application or session layer protocol.

To access the services of the transport provider, the transport user issues the appropriate service requests. An example of a service request would be to request a data transfer over a connection. In response, the transport provider notifies the user of various events, such as the arrival of data on a connection.

1.1 Transport Service Interface

The transport service interface (XTI) consists of a set of transport-independent C library functions that conform to the X/Open Transport Interface specifications. XTI applications can be written to support both BSD sockets and System V streams. A network application that uses the XTI calls is portable across systems, as long as both systems incorporate the XTI calls and support the same underlying transport provider. At present, ULTRIX operating system supports TCP and UDP transport providers using XTI.

1.1.1 Transport Service Interface Characteristics

In many ways, XTI is similar to the existing Berkeley Software Distributions (BSD) socket-based interprocess communication (IPC) primitives. Both provide a programming interface to access the underlying transport services and both use a file descriptor to identify the endpoint for communication. In XTI, the endpoint (file descriptor) is called a transport endpoint.

1.1.2 Application Portability

Compared to IPC, XTI provides additional functionality to facilite application portability. The additional functionality consists of the following:

- XTI provides calls that return the characteristics of the transport protocol. A portable application can use this information to identify the underlying transport provider. XTI also provides calls to retrieve, verify, or negotiate protocol options with the local transport provider.

- XTI defines an event management mechanism that lets transport providers notify applications of significant events. The current event on a transport endpoint is always available through user request. Furthermore, the occurrence of an asynchronous event that requires immediate attention also causes some XTI calls to return t_look() (some event).

- XTI allows multiple processes to share the same transport endpoint. Synchronization calls are defined to allow an application to synchronize with its transport provider. Synchronization among applications is still left to the user application.

1.1.3 XTI Enhancements

Compared to the BSD IPC calls, XTI has enhancements. These include:

- During connection establishment, XTI allows an application to exchange and negotiate connection options, determine the status of a previously-sent connect request, or selectively accept connections from several incoming connections.

- During data transfer, XTI applications can send one transport service data unit (normal or expedited) in multiple portions or receive one transport service data unit (normal or expedited) using multiple issues of the same call.

- During connection release, XTI applications can send user-initiated disconnect requests, identify the cause of a disconnect and retrieve any user data sent with the disconnect, initiate an orderly release, or acknowledge receipt of an orderly release indication.

1.1.4 Event Handling

The transport service interface is inherently asynchronous. Events can occur independently of the actions of the transport user. Signals can also interrupt the blocking call.

XTI defines a set of asynchronous events in which the application would be interested. The transport provider generates these events as a result of either protocol messages received over the network or clearing of flow control conditions within the transport provider. Refer to Chapter 4 for a detailed description of event handling.

1.2 Transport Provider

The transport provider is the transport protocol that provides the services of the transport service interface. Each transport provider supports a set of default quality-of-service parameters. These parameters are negotiable on a per-connection basis for connection-mode transport services and exchanged on a per-datagram basis for connectionless-mode transport services. Refer to Chapter 4 for a description of the transport provider's parameters.

1.3 Transport Endpoints

The file descriptor (transport endpoint) used by XTI is a UNIX file descriptor, which can be manipulated by file system calls such as `fork()`, `exec()`, `read()`, and `write()`.

1.4 Modes of Service

The transport service interface provides two modes of service: connection and connectionless. 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, data stream-oriented interactions.

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

- Involve short-term request/response interactions
- Exhibit a high level of redundancy
- Are dynamically reconfigurable
- Do not require guaranteed, in-sequence delivery of data

1.4.1 Connection-Mode Service

The connection-mode transport service is characterized by five phases:
- Initialization
- Connection establishment
- Data transfer
- Connection release
- Deinitialization

1.4.1.1   Initialization – The initialization phase defines the local operation between a
transport user and transport provider. For example, a user must establish a
communication path to the transport provider, as illustrated in Figure 1-2. Each
communication path between a transport user and transport provider is a unique
endpoint of communication and is called the transport endpoint. The t_open ()
function enables a user to choose a particular transport provider that will supply the
connection-mode services and establish the transport endpoint.

Figure 1-2:   Communication Path Between Transport User and Provider


```
TRANSPORT USER
(Networking application or Session layer protocol)
```

Service   Communication   Events
Request   Path

TRANSPORT SERVICE
INTERFACE

```
TRANSPORT PROVIDER
(TCP or UDP protocol)
```
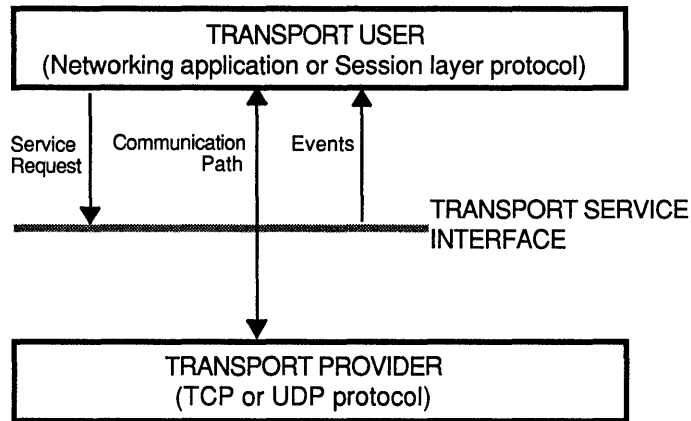
ZK-0098U-R

Another necessary local function for each user is to establish an identity with the
transport provider. Each user is identified by a protocol address. A protocol 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 may be as simple
as a random character string 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 functions are available to support
local initialization. Table 1-1 summarizes all local initialization functions of the
transport service interface.

Table 1-1: Initialization Functions for Connection-Mode

Function	Description
t_alloc()	Allocates transport service interface.
t_bind()	Binds a protocol address to a transport endpoint.
t_error()	Prints a transport service interface error message.
t_free()	Frees structures allocated using t_alloc().
t_getinfo()	Gets protocol-specific service information.
t_getstate()	Gets the current state of the transport endpoint.
t_look()	Returns the current event on a transport endpoint.
t_open()	Establishes a transport endpoint connected to a choosen transport provider.
t_optmgmt()	Negotiates protocol-specific options with the transport provider.
t_sync()	Synchronizes a transport endpoint with the transport provider.

1.4.1.2 Connection Establishment – The connection establishment phase enables two transport users to create a connection (virtual circuit), between them, as illustrated in Figure 1-3.

Figure 1-3: Connection Establishment
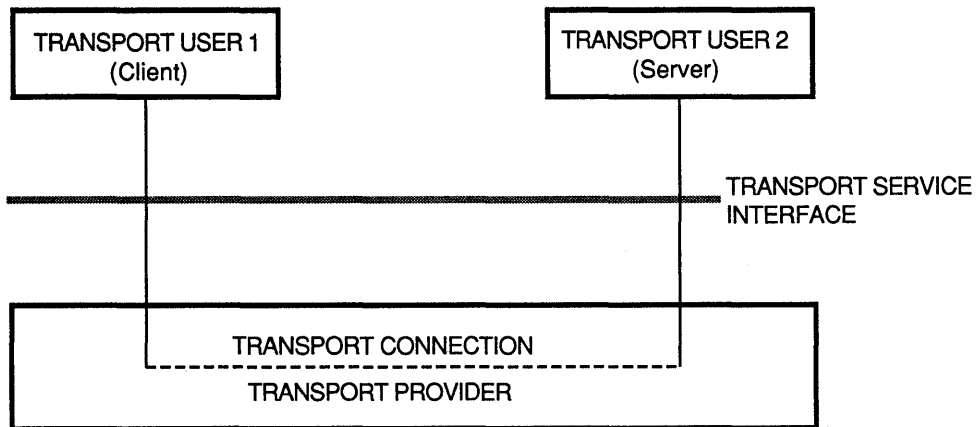
ZK–0100U–R

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 () function initiates the connect request. One argument to

`t_connect()`, the transport address, identifies the server that the client wishes to access. The server is notified of each incoming request using `t_listen()` and may call `t_accept()` to accept the client's request for access to the service. If the request is accepted, the transport connection is established.

Table 1-2 summarizes all functions available for establishing a transport connection.

Table 1-2: Connection Establishment Functions

Function	Description
t_accept()	Accepts a request for a transport connection.
t_connect()	Establishes a connection with the transport user at a specified destination.
t_listen()	Retrieves an indication of a connection request from another transport user.
t_rcvconnect()	Completes a connection establishment if t_connect() was called in asynchronous mode. See Chapter 4.

1.4.1.3 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 the 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. Table 1-3 summarizes the connection mode data transfer functions.

Table 1-3: Data Transfer Functions for Connection-Mode

Function	Description
t_snd()	Sends either normal or expedited data over a transport connection.
t_rcv()	Receives either normal or expedited data on a transport connection.

1.4.1.4 Connection Release – The connection release phase terminates a given transport connection in the connection-mode service. Two sets of calls are used, depending on whether the release is abrupt (abortive) or orderly.

The `t_snddis()` and `t_rcvdis()` functions are used for the abortive release. Because the abortive release does not coordinate between the peer transport providers, data can be lost. The `t_snddis()` call rejects an incoming connection request or ends a connection abruptly, depending on the state of the connection when the call is made. The `t_rcvdis()` call identifies the reason for the abortive release of a connection, where the connection is released by the transport provider or another

transport user.

Orderly release of a transport connections is an optional feature for the TCP protocol. Data from outstanding t_snd() calls are transmitted and retransmitted, as flow control permits, until all t_snd() calls have been serviced.

The t_sndrel() and t_rcvrel() calls are used for the orderly release. The t_sndrel() call can be issued by either transport user to initiate an orderly release of a transport connection. This call indicates to the transport provider that the transport user has no more data to send. The connection remains intact until both users issue the t_sndrel() function and t_rcvrel() function. The t_rcvrel() function is issued when a user is notified of an orderly release request, to inform the transport provider that the user is aware of the remote user's actions.

Table 1-4: Connection Release Functions

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

1.4.1.5 **Deinitialization** – The deinitialization phase provides local management of a transport endpoint. It can involve one or both of the following:

- Disabling a transport endpoint from accepting any further requests

- Informing the user that the transport provider is finished with the transport endpoint

Issuing t_unbind() disables a transport endpoint so that no further request destined for that endpoint will be accepted by the transport provider. In addition, t_unbind() disables event generation and disassociates the endpoint from its protocol address.

Issuing t_close() informs the transport provider that the user is finished with the transport endpoint and frees any local resources associated with that endpoint. Table 1-5 summarizes the deinitialization functions.

Table 1-5: Deinitialization Functions

Function	Description
t_unbind()	No further data or events destined for this transport endpoint will be accepted by the transport provider.
t_close()	The transport provider is informed that the user is finished with the transport endpoint.

1.4.2 Connectionless-Mode Service

The connectionless-mode transport service is characterized by three phases:

- Initialization

- Data transfer

- Deinitialization.

1.4.2.1 Initialization – The initialization phase defines the local operation between a transport user and transport provider. For example, a user must establish a communication path to the transport provider, as illustrated in Figure 1-4. Each communication path between a transport user and transport provider is a unique endpoint of communication, and is called the transport endpoint. The t_open () function enables a user to choose a particular transport provider that will supply the connectionless-mode services and establish the transport endpoint.

Figure 1-4: Connectionless Communication Path

```
┌─────────────────────────────────────────┐
│           TRANSPORT USER                 │
└─────────────────────────────────────────┘
                      ▲
  Transport Endpoint  │
 (Communication Path) │
━━━━━━━━━━━━━━━━━━━━━━━┿━━━━━━━━━━━━━  TRANSPORT SERVICE
                      │              INTERFACE
                      ▼
┌─────────────────────────────────────────┐
│         TRANSPORT PROVIDER               │
└─────────────────────────────────────────┘
```
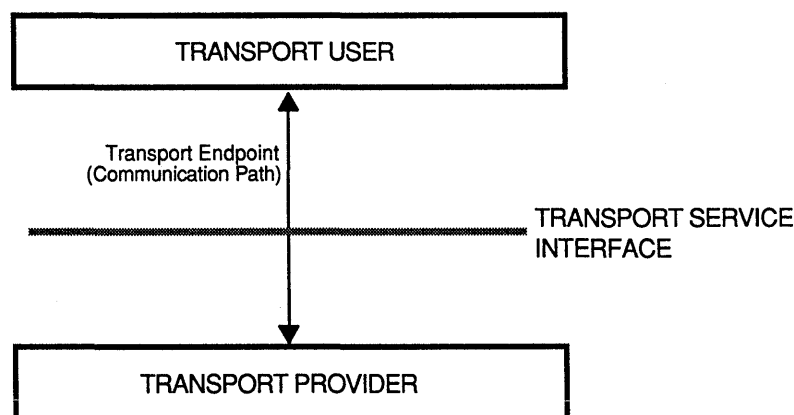
ZK-0099U-R

Another necessary local function for each user is to establish an identity with the transport provider. Each user is identified by a protocol address, which is associated with each transport endpoint, and one user process can manage several transport endpoints. In connectionless-mode service, in addition to the data sent by a user

process, each message contains a protocol address, making it possible to deliver the message to the correct recipient and for the recipient to send a reply. Addresses can be assigned to each transport endpoint by t_bind().

In addition to t_open() and t_bind(), several functions are available to support local initialization. Table 1-6 summarizes all local initialization functions of the transport service interface.

Table 1-6: Initialization Functions for Connectionless-Mode

Function	Description
t_alloc()	Allocates transport service interface.
t_bind()	Binds a protocol address to a transport endpoint.
t_error()	Prints a transport service interface error message.
t_free()	Frees structures allocated using t_alloc().
t_getinfo()	Gets protocol-specific service information.
t_getstate()	Gets the current state of the transport endpoint.
t_look()	Returns the current event on a transport endpoint.
t_open()	Establishes a transport endpoint connected to a chosen transport provider.
t_optmgmt()	Negotiates protocol-specific options with the transport provider.
t_sync()	Synchronizes a transport endpoint with the transport provider.

1.4.2.2 **Data Transfer** – 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 functions, t_sndudata() and t_rcvudata() support this message-based data transfer facility. Table 1-7 summarizes all functions associated with connectionless-mode data transfer.

Table 1-7: Data Transfer Functions for Connectionless-Mode

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

1.4.2.3 **Deinitialization** – The deinitialization phase provides local management of a transport endpoint. It may involve one or both of the following:

- Disabling a transport endpoint from accepting any further requests

- Informing the user that the transport provider is finished with the transport endpoint

Issuing t_unbind() disables a transport endpoint such that no further request destined for the given endpoint will be accepted by the transport provider. In addition, t_unbind() disables event generation and disassociates the endpoint from its protocol address.

Issuing t_close() informs the transport provider that the user is finished with the transport endpoint and frees any local resources associated with that endpoint. Table 1-8 summarizes the deinitialization functions.

Table 1-8: Deinitialization Functions for Connectionless-Mode

Function	Description
t_unbind()	No further data or events destined for this transport endpoint will be accepted by the transport provider.
t_close()	The transport provider is informed that the user is finished with the transport endpoint.

1.5 State Transitions

The transport service interface has two components:

- The library functions that provide the transport services to users

- The state transition rules that define the sequence in which the transport functions may be involved

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

Before writing software programs using the transport service interface, the user needs to understand all the possible state transitions.

Connection-Mode Service **2**

This chapter describes the connection-mode service of the transport service interface. As described in Section 1.4.1.2, the connection-mode service can be illustrated using a client-server paradigm.

2.1 Connection-Mode Programming Examples

The important concepts of connection-mode are described in this chapter with two programming examples: client and server. The client example illustrates how a client establishes a connection to a server and then communicates with the server. The other example illustrates the server's side of the interaction. The two examples discussed in this chapter are presented in their entirety in Appendix D.

2.2 Connection-Mode Initialization

Before the client and server (transport users) can establish a transport connection, each must first establish a communication path to the transport provider. A transport endpoint specifies a communication path between a transport user and a specific transport provider. A local file descriptor identifies a specific transport provider. To activate a transport endpoint, a protocol address must be associated with an endpoint.

The t_open() function is used to create a transport endpoint and returns protocol-specific information associated with that endpoint. A file descriptor is returned as the local identifier of the transport endpoint.

A successful t_open() returns a file descriptor and the default characteristics of the underlying transport protocol are returned in the *info* parameter. This information differs across transport providers. Refer to Chapter 4 for a description of what information is returned by which transport provider. 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 number of user data that can be passed between users during the abortive release of a connection

servtype

> Type of service supported by the transport provider

One of the following service types is returned:

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 of the services can be associated with the transport provider identified by t_open().

Note

> Some characteristics returned by t_open() can change after an endpoint has been opened. This occurs if the characteristics are associated with negotiated options, described later in this section.

After a user establishes a transport endpoint with the chosen transport provider, a protocol address must be associated with a given transport, thereby activating the endpoint. This association is done with t_bind(), which binds a protocol address to the transport provider. In addition, for servers, this association directs the transport provider to begin accepting connect indications, if desired.

Depending upon the transport provider, t_bind() can allow more than one transport endpoint to be bound to the same protocol address but disallows more than one protocol address to be bound to the same transport endpoint. If the application requests the binding of more than one transport endpoint to the same protocol address, only one transport endpoint can be used to listen for connect indications associated with that protocol address.

An optional facility, t_optmgmt(), is available during the local initialization phase. The t_optmgmt() function 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.

2.2.1 The Client

Example 2-1 illustrates the steps necessary to initialize the client. A discussion of the client initialize phase follows this example segment.

Example 2-1: Initialize Phase of the Client (Connection Mode)

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <netinet/in.h>
#include <stdio.h>
#include <ctype.h>
#include <errno.h>
#include <signal.h>
#include <setjmp.h>
#include <netdb.h>
#include <xti.h>
#include <fcntl.h>

extern int errno;
int net;
struct t_info t_open_info;    /* transport char. from transport */
struct t_info t_getinfo_info;
struct tcp_options tcp_opts;
struct t_optmgmt t_optm_req;
struct t_optmgmt t_optm_ret;
struct sockaddr_in sin;
struct     servent *sp;
char *hostname;
struct hostent *host;
#define MAXDSIZE 512
char snd_buf[MAXDSIZE];
char rcv_buf[MAXDSIZE];
int n;
int status;
struct t_call t_conn_sndcall;
struct t_call t_conn_rcvcall;
struct t_call t_rcvconn_call;

struct t_discon discon;
int t_rcv_flags;

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

  char destin[255];

  if ((net = t_open("tcp", O_RDWR|O_NONBLOCK, &t_open_info)) < 0) {    1
    t_error("t_open failed");    4
    exit(t_errno);
  }
  status = t_getinfo(net, &t_getinfo_info);    2


  /*
   * t_bind - bind an address to a transport endpoint
   *
   */
  if (t_bind(net, 0, 0) < 0) {    3
    t_error("iexample: t_bind error");    4
```

Example 2-1:   (continued)

```
    exit(1);
}

t_optm_req.opt.len = 0;
t_optm_req.flags = T_DEFAULT;
t_optm_ret.opt.maxlen = sizeof(struct tcp_options);
t_optm_ret.opt.buf = (char *) &tcp_opts;

status = t_optmgmt(net, &t_optm_req, &t_optm_ret);       5
if (status < 0) {
  t_error("iexample: t_optmgmt error");
  exit(1);
}

printf("host :");
scanf("%s",destin);

host = gethostbyname(destin);

if (host) {
  sin.sin_family = host->h_addrtype;
  bcopy(host->h_addr, (caddr_t)&sin.sin_addr, host->h_length);
  hostname = host->h_name;
}
```

1   The first argument to t_open(), *tcp*, identifies the transport provider as tcp.
In this example, the transport protocol is identified by name (tcp). It is opened
for both reading and writing, as specified by the O_RDWR open. The
O_RDWR flag is ORed with the O_NONBLOCK flag, which specifies
nonblocking operation (asynchronous mode). The asynchronous mode means
that if the requested operation t_open() cannot be completed, the
t_open() call returns -1 immediately and *t_errno* is set to a specific value.
The third argument, &*t_open_info*, returns various default characteristics of the
underlying transport protocol by setting fields in the t_open_info structure.
This argument, *t_open_info*, points to the t_open_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 expedited transport service data unit (ETSDU()) */

long *discon*
        /* max amount of data allowed on t_snddis() and t_rcvdis()
        functions */

long *sertype*
        /*service type supported by thje transport provider */

Refer to the t_open() reference pages for a description of the members of the
t_open_info structure.

As mentioned before, the third argument of the t_open() call can be used to return to the user the service characteristics of the transport provider. This information is useful when writing protocol-independent software, which is discussed in Appendix B. If the user did not need to know the transport characteristics, NULL would be specified for the third argument in the t_open call.

2 After opening the transport service, the t_getinfo() call gets protocol-specific service information, which is redundant to what was done with the third argument of the t_open() call. (The t_getinfo() call was added for illustrative purposes only.) Another alternative would have been to NULL the third argument of the t_open() call and use the t_getinfo() to obtain the protocol-specific service information.

The return value of the t_open() call is a file descriptor obtained by opening the transport protocol file. This file descriptor is an identifier that is used by all subsequent transport service interface calls.

3 After creating the transport endpoint, the client calls t_bind() to assign an address to it. The first argument (*net*) 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.

To access a server, clients use the address associated with the server's transport endpoint. Typically, the client does not care about its own address because no other process tries to access it. This is illustrated in the example, where the second and third arguments to t_bind() are set to NULL. A NULL second argument means that the transport provider assigns an appropriate address to be bound; in other words, the address chosen for the user. A NULL third argument indicates that the user does not care what address is assigned to the endpoint.

4 If either t_open() or t_bind() fail, the program calls t_error() to send an appropriate error message to stderr. If any transport service interface routine fails, the global integer *t_errno* is assigned an appropriate transport error value. A set of such error values has been defined (in <xti.h>) for the transport service interface, and *t_errno* prints an error message corresponding to the value in *t_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.

5 The example also illustrates the use of the optional facility, t_optmgmt(), which enables a user to negotiate the values of protocol options with the transport provider. Each transport protocol defines its own set of negotiable protocol options, which can include such information as quality-of-service parameters. Because t_optmgmt() is protocol-specific, only applications written for a specific protocol environment are expected to use this facility.

2.2.2 The Server

The server in this example must perform local initialization steps similarly to the client before communications can begin. The server must establish a transport endpoint through which it listens for connect indications. The necessary initialization steps are shown in Example 2-2. A discussion of the server initialization phase follows this example segment.

Example 2-2: Initialize Phase for the Server (Connection Mode)

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <sys/file.h>
#include <netinet/in.h>
#include <stdio.h>
#include <signal.h>
#include <errno.h>
#include <sgtty.h>
#include <netdb.h>
#include <syslog.h>
#include <xti.h>

int   net,net1,n,n1;
extern      int errno;

main(argc, argv)
      char *argv[];
{
      int fromlen;
      struct sockaddr_in from;

      int status;

      status =  get_income();
      if (status != 0)
        exit(1);
      else {
        sleep(10);
        exit(0);
          }
}

      .
      .
      .
      .


int
get_income()
{
   struct sockaddr_in sname;
   struct servent *sp;
   int i;
   int child;

   struct t_call t_list_call;
   struct t_call *t_list_ptr;
   struct t_call t_snddis_call;
   struct t_bind t_bind_addr_req;
   struct t_bind t_bind_addr_req1;
   struct t_bind t_bind_addr_ret;
   struct t_info t_open_info;   /* transport char. from transport */
   int t_status;

   /*
    * Call t_open - establish a transport endpoint
    *
    */

      if ((net = t_open("tcp", O_RDWR, &t_open_info)) < 0) {      1
```

Example 2-2:   (continued)

```
    t_error("rexample: t_open error");
    exit(1);
}

/*
 * t_bind - bind an address to a transport endpoint
 *
 */

sname.sin_port = 200;          /* load port # */
sname.sin_family = AF_INET;
sname.sin_addr.s_addr = 0;


t_bind_addr_req.addr.len = sizeof (struct sockaddr_in);
t_bind_addr_req.addr.buf = (char *) &sname;
t_bind_addr_req.qlen = 1;
t_bind_addr_ret.addr.maxlen = sizeof (struct sockaddr_in);
t_bind_addr_ret.addr.buf = (char *) &sname;


if ((t_bind(net, &t_bind_addr_req, &t_bind_addr_ret)) < 0) {        2
    t_error("rexample: t_bind error");
    exit(1);
}


t_list_ptr = (struct t_call *) t_alloc(net, T_CALL_STR, T_ADDR);   3
bcopy(&sname, t_list_ptr->addr.buf, t_list_ptr->addr.maxlen);

t_status = t_listen(net, t_list_ptr);

if (t_status < 0) {
    if (t_errno != TNODATA) {
        t_error("rexample: t_listen error");
        t_unbind(net);
        t_close(net);
        exit(1);
    }
}

printf("Have a incomming connection with sequence # %d\n",
       t_list_ptr->sequence);
printf("attempting to accept sequence # %d\n",
         t_list_ptr->sequence);

net1 = get_endpoint();
if (t_status = t_accept(net,net1,t_list_ptr) < 0) {
    t_error("rexample: t_accept error");
    if (t_errno == TLOOK) {
        printf("event %x came in\n",t_look(net1));
    }
    exit(1);
}

fcntl(net1,F_SETOWN, getpid());
child = fork();

if (child == 0) {
    t_unbind(net);
    t_close(net);
    t_sync(net1);
    doit(net1, t_list_ptr->sequence);
```

Example 2-2: (continued)

```
   }
   else
      {
        printf("Forking Child process =%d for fd = %d seq=%d\n",
             child,net1, t_list_ptr->sequence);
        t_unbind(net1);
        t_close(net1);
        t_free(t_list_ptr, T_CALL_STR);
      }
   return(0);
}

int
get_endpoint()
{
   struct sockaddr_in sname;
   struct servent *sp;
   int tmp_net;

   struct t_call t_list_call;
   struct t_bind t_bind_addr_req;
   struct t_bind t_bind_addr_req1;
   struct t_bind t_bind_addr_ret;
   struct t_info t_open_info;    /* transport char. from transport */
   int t_status;


/*
 * Call t_open - establish a transport endpoint
 *
 */

   if ((tmp_net = t_open("tcp", O_RDWR, &t_open_info)) < 0) {
     t_error("rexample: t_open error");
     exit(1);
   }

   /*
    * t_bind - bind an address to a transport endpoint
    *
    */

   sname.sin_port = 0;
   sname.sin_family = AF_INET;
   sname.sin_addr.s_addr = 0;


   t_bind_addr_req.addr.len = sizeof (struct sockaddr_in);
   t_bind_addr_req.addr.buf = (char *) &sname;
   t_bind_addr_req.qlen = 0;
   t_bind_addr_ret.addr.maxlen = sizeof (struct sockaddr_in);
   t_bind_addr_ret.addr.buf = (char *) &sname;


   if ((t_bind(tmp_net, &t_bind_addr_req, &t_bind_addr_ret)) < 0) {
     t_error("rexample: t_bind error");
     exit(1);
   }
   return(tmp_net);
}
```

1   Like the client, the first step is to call t_open() to establish a transport
    endpoint with the desired transport provider. Refer to the

`get_incomeroutine` in the example for this discussion. This endpoint, *net,*
is used to listen for connection requests from the clients.

2	Next, the server must bind its address, which is well-known to the clients, to
the endpoint. Each client uses this address to access the server. The second
argument to `t_bind()`,*&t_bind_addr_req*, `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;
}
```

The members have the following meanings:

addr

Address to be bound

qlen

Maximum outstanding connect indications that can arrive at this endpoint

Note

All transport service interface structure and constant definitions are
located in `<xti.h>`.

A `netbuf` structure specifies the address, which consists of the following members:

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

These members have the following meaning:

buf

Points to a buffer containing data which identifies a transport address.

len

Specifies the bytes of data in the buffer.

maxlen

Indicates the maximum bytes the buffer can hold (set only to return data to the
user by the transport service interface routine).

The structure of addresses varies among each protocol implementation under the
transport service interface. The `netbuf` structure should be able to support any
variations.

The *qlen* value 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
but which has not been accepted or rejected. In the example, *qlen* (value of 1) is
greater than 0, which means the transport endpoint can be used to listen for connect
indications. The `t_bind()` call directs the transport provider to immediately begin
queueing connect indications destined for the bound address. Furthermore, the *qlen*
value specifies the maximum outstanding connect indications the server can process.

The server must respond to each connect request, either accepting or rejecting the request for connection.

3 The t_alloc() call is called to allocate memory for the needed t_bind structure to hold the correct address. The t_alloc() function takes three arguments: *fd* (*net*, *struct_type*), (*T_CALL_STR*), and *fields* (*T_ADDR*). The first argument, *net*, which is a file descriptor, references a transport endpoint. It is used to access the characteristics of the transport provider. The second argument, *struct_type*, identifies the appropriate transport service interface structure to be allocated. The third argument, *fields*, specifies which netbuf buffers should be allocated for that structure. The size of this buffer is determined from the transport provider characteristics that define the maximum address size. The t_alloc() call sets the *maxlen* field of this netbuf structure to the size of the newly allocated buffer.

In this example, because *qlen* is set to 1, the server processes connect indications one at a time. The address information is assigned to the newly allocated t_bind structure. The t_bind structure is used to pass information to t_bind() in the second argument and also is 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. Should the transport provider not be able to bind the requested address (for example, it can already be bound), another appropriate address would be chosen.

The server checks the bound address to ensure that it is the one previously advertised to clients. Otherwise, the clients are unable to reach the server.

If t_bind() is successful, the transport provider begins queueing connect indications. The next phase of communication, connection establishment, is entered.

2.3 Connection Establishment

The connection establishment procedures emphasize the difference between clients and servers. The transport service interface imposes a different set of procedures in this phase for each type of transport user. The client uses t_connect() to initiate the connection establishment procedure by requesting 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 the client's request by calling t_accept() to establish the connection, or calling t_snddis() to reject the client's request. The server notifies the client of the decision to accept or reject the connection when t_connect() completes.

The transport service 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 connect establishment.

The second optional service supported by the transport service interface during connection establishment is the negotiation of protocol options. The client can specify protocol options that it would like the transport provider or the remote user to use. The transport service interface supports both local and remote option negotiation.

As discussed earilier, option negotiation is inherently a protocol-specific function. Use of this facility is discouraged if protocol-independent software is a goal (Refer to Appendix B).

2.3.1 The Client

Continuing with the connection-mode example, the steps needed by the client to establish a connection are shown in Example 2-3. The example segment is followed by a discussion of the steps.

Example 2-3: Connection Phase for the Client (Connection Mode)

```
printf("host :");
scanf("%s",destin);

host = gethostbyname(destin);

if (host) {
   sin.sin_family = host->h_addrtype;
   bcopy(host->h_addr, (caddr_t)&sin.sin_addr, host->h_length);
   hostname = host->h_name;
}

sin.sin_port = 200; /* try to connect to port 200 */
t_conn_sndcall.addr.len = sizeof (struct sockaddr_in);
t_conn_sndcall.addr.buf = (char *) &sin;
t_conn_sndcall.opt.len = 0;
t_conn_sndcall.udata.len = 0;
t_conn_rcvcall.addr.maxlen = sizeof (struct sockaddr_in);
t_conn_rcvcall.addr.buf = (char *) &sin;
t_conn_rcvcall.opt.maxlen = sizeof(struct tcp_options);
t_conn_rcvcall.opt.buf = (char *) &tcp_opts;
t_conn_rcvcall.udata.maxlen = 0;
t_rcvconn_call.addr.maxlen = sizeof (struct sockaddr_in);
t_rcvconn_call.addr.buf = (char *) &sin;
t_rcvconn_call.opt.maxlen = sizeof(struct tcp_options);
t_rcvconn_call.opt.buf = (char *) &tcp_opts;
t_rcvconn_call.udata.maxlen = 0;
t_rcvconn_call.udata.buf = 0;
if ((t_connect(net, &t_conn_sndcall, &t_conn_rcvcall)) < 0) {       [1]
   if (t_errno == TNODATA) {
     while (1) {
     status = t_rcvconnect(net, &t_rcvconn_call);       [2]

     if (status < 0) {
       if (t_errno == TLOOK) {
         printf("Event %x came in\n",t_look(net));
         (void) t_unbind(net);
         (void) t_close(net);
         exit(1);
       }
       if (t_errno != TNODATA) {
         t_error("iexample: t_rcvconnect()");
         (void) t_unbind(net);
         (void) t_close(net);
         exit(1);
       }
     }
     else
       break;
     }
   } else {
```

Example 2-3: (continued)

```
    t_error("iexample: t_connect()");
    (void) t_unbind(net);
    (void) t_close(net);
    exit(1);
}
}
```

1. The t_connect() call establishes the connection with the server. The first argument, *net*, identifies the transport provider through which the connection is established. The second argument, *t_conn_sndcall*, identifies the destination server by containing the address of a t_call structure, which has the following members:

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

The members have the following meanings:

addr
   Specifies the protocol address of the destination transport user.

opt
   Presents any protocol-specific information that may be needed by the transport provider.

udata
   Points to optional user data that can be passed to the destination transport user during connection establishment.

sequence
   Has no meaning for this function.

It should be noted that the *t_conn_sndcall.opt.len* argument in this example is set to zero. This argument defines the options, which are specific to the underlying protocol, that are passed to the transport provider. By setting this argument to zero, the user has chosen to use the default options.

The *t_conn_sndcall.udata.len* argument has also been set to zero in our example. This argument enables the caller to pass user data to the destination transport; but, by specifying a value of zero, no data is sent to the destination transport user.

The third argument (*t_conn_rcvcall*) can be used to return information about the newly established connection to the user, and can retrieve any user data sent by the server in its response to the connect request. The *t_conn_rcvcall* argument points to a t_call structure. The members of the t_call structure have the following meanings:

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 that can be returned during connection establishment.

sequence

      Has no meaining for this function.

On return, the *addr*, *opt*, and *udata* fields of *t_conn_rcvcall* are updated to reflect values associated with connection. Thus, the *maxlen* field of each argument must be set before issuing t_connect() to indicate the maximum size of the buffer for each. However, if *t_conn_rcvcall* had been set to NULL, than no information is returned to the user from the t_connect().

2   The t_rcvconnect() call confirms the connection to the server (asynchronous mode only). The first argument, *net*, identifies the local transport endpoint where communication has been established. The second argument, *t_rcvconn_call*, points to a t_call structure that contains information about the newly established connection.

    In our example, the t_rcvconnect() call is operating in asynchronous mode because the O_NONBLOCK flag was specified in the t_open() call. This means that t_rcvconnect() is reduced to a poll for an existing connect confirmation. If there is no connect confirmation, t_rcvconnect() fails and returns immediately, without waiting for the connection to be established. The t_rcvconnect() call must be reissued at a later time to complete the connection establishment phase and retrieve the information returned to the call.

As shown in the state tables of Appendix A, it is possible in some states to receive one of several asynchronous events. The t_look() routine enables a user to determine what event has occured 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 is a disconnect indication. The client exits if its request is rejected.

2.3.2 The Server

Continuing with the server example, when the client calls t_connect(), a connect indication is generated on the server's listening endpoint. For each client, the server accepts the connect request and spawns a server process to manage the connection. Example 2-4 shows the required steps by the server to establish a connection and it is followed by a discussion of the steps.

Example 2-4: Connection Phase for the Server (Connection Mode)

```
t_list_ptr = (struct t_call *) t_alloc(net, T_CALL_STR, T_ADDR);      2
bcopy(&sname, t_list_ptr->addr.buf, t_list_ptr->addr.maxlen);

t_status = t_listen(net, t_list_ptr);

if (t_status < 0) {
  if (t_errno != TNODATA) {
    t_error("rexample: t_listen error");
    t_unbind(net);
    t_close(net);
    exit(1);
  }
}

printf("Have an incomming connection with sequence # %d\n",
    t_list_ptr->sequence);
printf("attempting to accept sequence # %d\n",
```

Example 2-4:   (continued)

```
            t_list_ptr->sequence);

    net1 = get_endpoint();
    if (t_status = t_accept(net,net1,t_list_ptr) < 0) {        1
      t_error("rexample: t_accept error");
      if (t_errno == TLOOK) {
        printf("event %x came in\n",t_look(net1));
      }
      exit(1);
    }

    fcntl(net1,F_SETOWN, getpid());
    child = fork();

    if (child == 0) {
      t_unbind(net);
      t_close(net);
      t_sync(net1);        3
      doit(net1, t_list_ptr->sequence);
    }
    else
      {
        printf("Forking Child process =%d for fd = %d seq=%d\n",
            child,net1, t_list_ptr->sequence);
        t_unbind(net1);
        t_close(net1);
        t_free(t_list_ptr, T_CALL_STR);
      }
    return(0);
}

int
get_endpoint()
{
    struct sockaddr_in sname;
    struct servent *sp;
    int tmp_net;

    struct t_call t_list_call;
    struct t_bind t_bind_addr_req;
    struct t_bind t_bind_addr_req1;
    struct t_bind t_bind_addr_ret;
    struct t_info t_open_info;    /* transport char. from transport */
    int t_status;

/*
 * Call t_open - establish a transport endpoint
 *
 */

    if ((tmp_net = t_open("tcp", O_RDWR, &t_open_info)) < 0) {
      t_error("rexample: t_open error");
      exit(1);
    }

    /*
     * t_bind - bind an address to a transport endpoint
     *
     */

    sname.sin_port = 0;
    sname.sin_family = AF_INET;
```

Example 2-4: (continued)

```
    sname.sin_addr.s_addr = 0;


    t_bind_addr_req.addr.len = sizeof (struct sockaddr_in);
    t_bind_addr_req.addr.buf = (char *) &sname;
    t_bind_addr_req.qlen = 0;
    t_bind_addr_ret.addr.maxlen = sizeof (struct sockaddr_in);
    t_bind_addr_ret.addr.buf = (char *) &sname;


    if ((t_bind(tmp_net, &t_bind_addr_req, &t_bind_addr_ret)) < 0) {
      t_error("rexample: t_bind error");
      exit(1);
    }
    return(tmp_net);
}
```

1. The server loops to process each connect indication. First, the server calls t_listen() to retrieve the next connect indication. When a connect indication arrives, the server calls t_accept() to accept the connect request. The first argument, *net*, of t_accept() identifies the local transport endpoint where the connect indication arrived. The second argument, *net1*, is used for the local transport endpoint that establishes the connection. Because the connection is accepted on an alternate endpoint, the server can continue to listen for connect indications on the endpoint that was bound for listening. If the call is accepted without error, a process is spawned to manage the connection.

As mentioned before, a different transport endpoint, *net1*, is used for a connection than the transport endpoint, *net*, that is used to receive the connection indication. Before t_accept() can be issued, the endpoint, *net1*, must be bound to a protocol address and must be in the T_IDLE state. Refer to the get_endpoint() function in the example for the procedure on binding the protocol address.

The third argument, *t_list_ptr*, points to a t_call structure that contains information required by the transport provider to complete the connection. The members of the t_call structure have the following meanings:

addr
Specifies the address of the caller.

opt
Indicates any protocol-specific parameters associated with the connection.

udata
Points to any data to be returned to the call.

Sequence
Is the value returned by t_listen() that uniquely associates the response with a previously received connect indication.

2. The t_alloc() function is called so that the server can allocate a t_call structure to be used by t_listen(). The first argument, *net*, refers to the transport endpoint that is used to allocate the new structure. The second argument (*T_CALL_STR*) specifies that the allocated structure that is of type t_call and third argument, *T_ADDR*, specifies which buffers are to be allocated. The t_alloc() call must allocate a buffer large enough to store the address of the caller. The buffer size is determined from the *addr*

characteristics returned by t_open(). The *maxlen* field of each netbuf structure is set by t_alloc() to the size of the newly allocated buffer.

③ In the example, the t_sync() function is called to synchronize the internal tables. This function converts an uninitialized file descriptor to an initialized transport endpoint by updating the necessary library data structures.

2.4 Data Transfer

Once the connection has been established, the transport server interface does not differentiate between the client and the server. Either the client or server can begin transferring data over the connection using t_snd() or t_rcv(). Not only can either user send or receive data, but either can also release the connection when appropriate. The transport service interface guarantees reliable, sequenced delivery of data over an existing connection.

Using the TCP protocol, the transport service interface supports the exchange of both normal and expedited data over a transport connection. Expedited data is typically associated with information of an urgent nature. The urgent nature is often indicated by one byte in the data stream. Most TCP applications are expected to discard all data up to the urgent data when the urgent signal is received. It should be noted that the exact semantics of expedited data are subject to the interpretation of the transport provider.

The TCP transport provider allows the user to specify an urgent condition at any point in the normal data stream. Several such indications can be combined, with only the last one shown to the destination. There is no limit to the number of urgent indications that can be sent. However, the user must send at least one data octet with each urgent indication. Current TCP implementation supports sending up to the maximum segment size of urgent data, but retrieval of only one byte of urgent data. If several urgent data are received, only the outstanding urgent data is reported.

Note

The user must set the T_MORE flag (t_snd()) to send multiple units over a transport connection, whereas the T_MORE flag is automatically set to receive (t_rcv()) a message in multiple units. The TCP transport provider ignores the T_MORE flag.

2.4.1 The Client

Example 2-5 shows how the client can transfer data to or from the server. A discussion of client data transfer follows this example segment.

Example 2-5: Data Transfer for the Client (Connection Mode)

```
printf("calling t_snd with %d bytes of regular data\n",sizeof(snd_buf));
n = t_snd(net, &snd_buf[0],sizeof(snd_buf) , 0);     ①

if (n < 0) {
  if (t_errno == TLOOK) {
    printf("Generated a %X TLOOK error\n",t_look(net));
    (void) t_unbind(net);
    (void) t_close(net);
    exit(1);
```

Example 2-5: (continued)

```
    }
    t_error("iexample: t_snd error");
    (void) t_unbind(net);
    (void) t_close(net);
    exit(1);
}
printf("t_snd sent %d bytes\n",n);

while (1) {
    n = t_rcv(net, rcv_buf, sizeof(rcv_buf), &t_rcv_flags);      ②


    if (n < 0) {
        if (t_errno != TNODATA) {
        t_error("iexample: t_rcv error");
        (void) t_unbind(net);
        (void) t_close(net);
        exit(1);
        }
        else {
        t_error("iexample: NO data available");
        }
    }
    if (n > 0) break;
}

printf("t_rcv received %d bytes\n",n);

if (t_rcv_flags & T_EXPEDITED)
    printf("data is expedited\n");
else
    printf("data is normal\n");

n = t_sndrel(net, (struct t_call *) 0);

if (n < 0) {
    t_error("iexample: error in t_sndrel:");
    t_unbind(net);
    t_close(net);
    exit(1);
}

}
```

①   The client calls t_snd() to send data to the server. The first argument, *net*, identifies the local transport endpoint over which the data is to be sent. The second argument, *&snd_buf[0]*, points to the user data to be sent, while the third argument, *sizeof(snd_buf)*, specifies the number of bytes to be sent. The fourth argument is used for optional flags. In the example, the argument *0* means no flags are set. The optional flags could have been either *T_EXPEDITED* or *T_MORE*. The *T_EXPEDITED* flag specifies the data to be expedited, while a *T_MORE* flag is ignored by the TCP transport provider.

②   The client continuously calls t_rcv() to process incoming data. Because t_rcv() is operating in the asynchronous mode in the example, if there is no data, t_rcv() fails. The first argument *net* identifies the local transport endpoint through which data arrives. The second argument, *rcv_buf*, points to the buffer where the user data is placed. The third argument, *sizeof(rcv_buf)*, specifies the size of the receive buffer in bytes.

2.4.2 The Server

Example 2-6 shows how the server can transfer data to and from the client. The server data transfer is discussed following this example segment.

Example 2-6: Data Transfer for Server (Connection Mode)

```
doit(f, seq)
     int f,seq;
{
  int t_rcv_flags;
  struct hostent *hp;
  char rcv_buf[512];
  char snd_buf[512];
  int n;


  while (1) {
    n = t_rcv(f,rcv_buf, sizeof(rcv_buf) ,&t_rcv_flags);     1

    if (n < 0) {
      if (t_errno != TNODATA) {
      t_error("rexample: t_rcv error");
      t_unbind(f);
      t_close(f);
      exit(1);
       }
       else {
      t_error("rexample: NO data available");
       }
    }
    if (n > 0) break;
  }

  printf("t_rcv received %d bytes\n",n);

  if (t_rcv_flags & T_EXPEDITED)
    printf("data is expedited\n");
  else
    printf("data is normal\n");

  printf("calling t_snd with %d bytes of regular data\n",sizeof(snd_buf));
  n = t_snd(f, &snd_buf[0],sizeof(snd_buf) , 0);

  if (n < 0) {
    if (t_errno == TLOOK) {
      printf("Generated a %X TLOOK error\n",t_look(f));
      (void) t_unbind(f);
      (void) t_close(f);
      exit(1);
    }
    t_error("rexample: t_snd error");
    (void) t_unbind(f);
    (void) t_close(f);
    exit(1);
  }
  printf("t_snd sent %d bytes\n",n);
```

As mentioned before, when the connection has been established, the transport service interface does not differentiate between the client and the server. As the following description shows, the server description is very similar to the client description.

[1] The server calls t_rcv() to receive data or expedited data over the connection. The first argument, *f*, of t_rcv() identifies the local transport endpoint through which data arrives. The second argument, *rcv_buf*, points to the buffer where the user data is placed, while the third argument, *sizeofd(rcv_buf*, specifies the size of the receive buffer. The fourth argument, *&t_rcv_flags*, points to the optional flags. The example checks for expedited data; if there is expedited data, the message "data is expedited" is printed.

2.5 Connection Release

At any point during data transfer, either user can release the transport connection and end the data exchange between the two users. The transport service interface supports two kinds of connection release:

- Abortive release

- Orderly release

The abortive release breaks a connection immediately and can result in the loss of any data that has not yet reached the destination user. To generate an abortive release, either user calls t_snddis(). In addition, the transport provider can abort a connection if a problem occurs below the transport service interface. A user can use t_snddis() 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 the connection was aborted, and 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 should not be interpreted by protocol-independent software.

The orderly release gracefully terminates a connection and guarantees that no data is lost. Orderly release is an optional facility that is supported by the TCP transport provider.

2.5.1 The Client

If the server releases the connection by issuing t_sndrel(), t_rcv() fails and sets t_errno() to TLOOK. The client then processes the connection release as shown in Example 2-7.

Example 2-7: Connection Release for the Client (Connection Mode)

```
n = t_sndrel(net, (struct t_call *) 0);

if (n < 0) {
  t_error("iexample: error in t_sndrel:");
  t_unbind(net);
  t_close(net);
  exit(1);
}

while (1) {
```

Example 2-7: (continued)

```
      n = t_rcvrel(net);      1

      if (n < 0) {
         if (t_errno != TLOOK && t_errno != TNOREL) {
         t_error("iexample: error in t_rcvrel:");
         t_unbind(net);
         t_close(net);
         exit(1);
          }
          else {
         if (t_errno == TNOREL)
            t_error("iexample: NO T_ORDREL available");
         else {
            t_error("iexample: TLOOK event");
            t_unbind(net);
            t_close(net);
            exit(1);
         }
          }
      }
      if (n == 0) break;
   }
   t_unbind(net);      2
   t_close(net);       3
   exit(0);
}
```

1   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 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.

2   Deinitialization of a transport endpoint provides local management only, it does not send information over the network. Issuing t_unbind() disables a transport endpoint so that no further request destined for the given endpoint is accepted by the transport provider. In addition, t_unbind() disables event generation and disassociates the endpoint from its protocol address.

3   Issuing t_close() informs the transport provider that the user is finished with the transport endpoint and frees any local resources associated with that endpoint.

2.5.2 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 shown in Example 2-8.

Example 2-8: Connection Release for the Server (Connection Mode)

```
while (1) {
  n = t_rcvrel(f);

  if (n < 0) {
    if (t_errno != TLOOK && t_errno != TNOREL) {
    t_error("rexample: error in t_rcvrel:");
    t_unbind(f);
    t_close(f);
    exit(1);
     }
     else {
    if (t_errno == TLOOK) {
      t_error("TLOOK error");
      t_unbind(f);
      t_close(f);
      exit(1);
    }
    t_error("rexample: NO T_ORDREL available");
     }
  }
  if (n == 0) break;
}

n = t_sndrel(f, (struct t_call *) 0);

if (n < 0) {
  t_error("rexample: error in t_sndrel:");
  t_unbind(f);
  t_close(f);
  exit(1);
}

t_unbind(f);
t_close(f);
exit(0);
}
```

2.6 Deinitialization

Refer to the previous client and server connection-release example segment for an example of deinitialization.

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

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

The example in this chapter appears in its entirety in Appendix E.

3.1 Initialization

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

In Example 3-1, the definitions and local management calls needed by the transaction server are shown and a description follows the example.

Example 3-1: Initialization Phase for the Transaction Server (Connectionless Mode)

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <sys/file.h>
#include <netinet/in.h>
#include <stdio.h>
#include <signal.h>

#include <errno.h>
#include <sgtty.h>
#include <netdb.h>
#include <syslog.h>
#include <xti.h>


struct sockaddr_in sname;
int    net,ncc;
extern      int errno;
extern  void do_setup();
struct t_unitdata unitdata;

main(argc, argv)
      char *argv[];
{
      do_setup();
      doit(net);
```

Example 3-1: (continued)

```
}


doit(f)
     int f;
{
  int t_rcv_flags;
  struct hostent *hp;
  char rcv_buf[5120];
  struct sockaddr sname1;

  unitdata.addr.maxlen = sizeof(sname1);
  unitdata.addr.buf = (char *) &sname1;
  unitdata.opt.maxlen = 0;
  unitdata.opt.buf = 0;
  unitdata.udata.maxlen = sizeof(rcv_buf);
  unitdata.udata.buf = &rcv_buf[0];
  .
  .
  .


void
  do_setup()
{
  struct t_call t_list_call;
  struct t_bind t_bind_addr_req;
  struct t_bind t_bind_addr_req1;
  struct t_bind t_bind_addr_ret;
  struct t_info t_open_info;   /* transport char. from transport */
  int t_status;


/*
 * Call t_open - establish a transport endpoint
 *
 */

  if ((net = t_open("udp", O_RDWR, &t_open_info)) < 0) {      1
    t_error("rexamless: t_open error");
    exit(1);
  }


  /*
   * t_bind - bind an address to a transport endpoint
   *
   */

  sname.sin_port = 200;
  sname.sin_family = AF_INET;

  t_bind_addr_req.addr.len = sizeof (struct sockaddr_in);
  t_bind_addr_req.addr.buf = (char *) &sname;
  t_bind_addr_req.qlen = 1;
  t_bind_addr_ret.addr.maxlen = sizeof (struct sockaddr_in);
  t_bind_addr_ret.addr.buf = (char *) &sname;

  if ((t_bind(net, &t_bind_addr_req, &t_bind_addr_ret)) < 0) {      2
    t_error("rexamless: t_bind error");
    exit(1);
  }
}
```

[1]	The connectionless-mode initialization is similar to the connection-mode initialization. The server establishes a transport endpoint with the desired transport provider, using t_open(). In the above example segment, the first argument, *udp*, of t_open() identifies the UDP transport provider. The second argument, *O_RDWR*, identifies the open flag as being READ and WRITE operation. The third argument, *&t_open_info*, points to a location where the returned characteristics of the underlying transport protocol are placed. Refer to the t_open() reference pages for a description of the returned characteristics.

[2]	Like the connection-mode server, the connectionless-mode server also binds a transport address to the endpoint, so that potential clients can identify and access the server. The transport address is bound to the endpoint by using a t_bind() call. The first argument, *net*, identifies the transport endpoint which is associated with a protocol address. Both the second argument, *&t_bind_addr_req*, and third argument, *&t_bind_addr_ret*, point to t_bind structures. The second argument contains the address that is requested to be bound with the transport endpoint. On return, the third argument contains the address that was actually bound to the transport endpoint. This returned address can be different from the address specified in the second argument.

Unlike the connection-mode server, the *qlen* field of the t_bind structure has no meaning for connectionless-mode service, because all users are capable of receiving datagrams once they have bound an address. It should be noted that the transport service interface does define a client-server relationship between two users in the connection-mode service; however, no such relationship exists in the connectionless-mode service. It is this example, not the transport service interface, that defines one user as a server and another as a client.

Once the endpoint is bound, the transport user can send or receive data units through the transport endpoint.

3.2 Data Transfer

After a user has bound a protocol 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 service interface enables a user to specify protocol options that should be associated with the transfer of the data unit. 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.

Example 3-2 shows the steps for the server to receive data. A description of the data transfer follows this example segment.

Example 3-2: Data Transfer for Transaction Server (Connectionless Mode)

```
doit(f)
     int f;
{
  int t_rcv_flags;
  struct hostent *hp;
  char rcv_buf[5120];
  struct sockaddr sname1;

  unitdata.addr.maxlen = sizeof(sname1);
  unitdata.addr.buf = (char *) &sname1;
  unitdata.opt.maxlen = 0;
  unitdata.opt.buf = 0;
  unitdata.udata.maxlen = sizeof(rcv_buf);
  unitdata.udata.buf = &rcv_buf[0];

  ncc = t_rcvudata(f,&unitdata,&t_rcv_flags);      1

  if (ncc == 0)
    printf("received %d octets\n",unitdata.udata.len);
  else
    printf("ncc = %d, errno =%d\n",ncc,errno);
  (void) t_close(f);      2
  exit(0);
}
```

1 In the example, t_rcvudata is called to receive a data unit. The first argument, *f*, of t_rcvudata identifies the local transport endpoint through which data will be received. The second argument, *&unitdata*, points to a t_unitdata structure that contains the following members:

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

On return from the call, the members have the following meanings:

addr
> Specifies the protocol address of the sending unit.

opt
> Identifies protocol-specific options that were associated with this data unit.

udata
> Specifies the user data that was received.

The third argument, *&t_rcv_flags* is set on return to indicate that the complete data unit was not received. In other words, the buffer defined in the *udata* field of *&unitdata* is not large enough to hold the current data unit. The buffer is filled and T_MORE is set in *&t_rcv_flags* on return, to indicate that another *t_rcv_udata* should be issued to retreive the rest of the data unit. Subsequent t_rcvudata calls return zero for the length of the address and for options, until the full data unit has been received.

3.3 Deinitialization

Deinitialization of a transport endpoint provides local management only. It does not send information over the network. Issuing t_unbind() disables a transport endpoint so that no further request destined for the given endpoint is accepted by the transport provider. In addition, t_unbind() disables event generation and disassociates the endpoint from its protocol address.

2 Issuing t_close() informs the transport provider that the user is finished with the transport endpoint and frees any local resources associated with that endpoint.

Refer to Example 3-2 for an example of deinitialization.

Advanced Topics 4

This chapter contains important concepts of the transport service interface that have not been discussed in the previous chapters. It describes:

- The characteristics associated with a transport endpoint that can be changed after an endpoint is opened

- How memory resources can be managed

- Choosing a mode of execution for an application

- Reporting events to an application

- Using the two levels of error reporting

4.1 Management of Local Transport Characteristics

As was discussed in previous chapters, the t_open () call returns the default provider characteristics associated with a transport endpoint. However, some characteristics can change after an endpoint has been opened. The characteristics that can change are associated with negotiated options. An example of a negotiated option is the support of an expedited data transfer. The t_getinfo () call can be used to retrieve the current characteristics of a transport endpoint.

The t_optmgmt () call can be used to negotiate the values of protocol options with the transport provider. Each transport protocol defines its own set of negotiable protocol options. See Section 4.1.1 for a description of the transport-protocol characteristics. Because of the protocol-specific nature of options, only applications written for a particular protocol environment are expected to use the t_optmgmt () call.

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

The XTI library provides information on both the default characteristics of the underlying transport protocol and the quality of service supported by the transport provider.

4.1.1 Transport-Protocol Characteristics

Table 4-1 lists the characteristics of the transport protocol, which is supported by the underlying transport provider. The characteristics are returned in the t_info structure by both t_open () and t_getinfo () calls.

Table 4-1: Transport Provider Characteristics

Parameters	Before Call	After Call (TCP)	After Call (UDP)
info->addr	/	x	x
info->options	/	x	-2
info->tsdu	/	0	x
info->etsdu	/	-1	-2
info->connect	/	-2	-2
info->discon	/	-2	-2
info->servtype	/	T_COTS_ORD	T_CLTS

/
    The parameter value is meaningless

x
    Value determined by the transport protocol

0
    The transport provider does not support the concept, although the function is supported in another form

-1
    No limit on the value supported

-2
    Not allowed by the transport protocol

4.1.2 Quality of Service

The following are quality of service (types of service) supported by the different transport providers.

4.1.2.1 Types of Service Supported by TCP — The TCP transport provider supports the types of services listed in Table 4-2. The type of service is returned in the tcp_options structure in the *opt* fields of parameters of the t_listen(), t_accept(), t_connect, t_rcvconnect(), and t_optmgmt() calls.

Table 4-2: Types of Service

Parameter	Service Type	Description
precedence	TCP_ROUTINE	Routine precedence, defined in xti.h
timeout(ms)	TCP_LINGERTIME converted to ms	Maximum linger time (2 minutes), defined in tcp_timer.h
max_seg_size	TCP_MSS	Default maximum segment size for TCP, defined in tcp.h
secoptions	--------	--------
security	T_UNUSED	Not used
compartment	T_UNUSED	Not used
handling	T_UNUSED	Not used
tcc	T_UNUSED	Not used

4.1.2.2 Types of Service Supported by UDP – The UDP transport provider does not support Quality of Service options, because the association of types of service with each datagram is not supported by UDP.

4.2 Management of Memory Resources

The t_alloc() and t_free() functions are used to manage the memory resources for XTI applications. The t_alloc() function dynamically allocates storage for the specified library data structure. The structure type has to be one of the following:

- T_BIND_STR
- T_CALL_STR
- T_OPTMGMT_STR
- T_DIS_STR
- T_UNITDATA_STR
- T_INFO_STR

The t_free() function is used to free memory previously allocated by t_alloc(). If memory has been allocated for buffers referenced by the structure, the t_free() call also frees the referenced buffers first, before the structure itself is freed. Also, the t_free() call frees memory allocated by malloc(). If the *ptr* argument in the t_alloc() call or any of the *buf* pointers points to a block of memory that was not previously allocated by t_alloc(), t_free() does not return with any warning.

4.3 Modes of Execution

The XTI library offers both synchronous and asynchronous modes of execution. The effect is local only to the application process. By default, all XTI calls are synchronous.

In the synchronous mode, an application normally blocks until completion. For example, an application making a synchronous t_rcv() call blocks until data from over the network can be retrieved.

In the asynchronous mode, an application can use the nonblocking I/O feature. If the requested operation cannot be completed, the XTI call returns immediately with -1, and *t_errno* is set to a specific value. For example, an application making an asynchronous t_rcv() call returns immediately if no data is available. The application can then periodically poll for the required event by means of the t_look() call. The reissued XTI call can be successful only after the event has occurred.

The ay011chronous mode is specified through the O_NONBLOCK flag, which can be set in either a t_open() call or a fcntl() call.

4.4 Event Handling

The XTI defines a set of events that must be reported to XTI applications. These events are generated (written) by the transport provider and consumed (read) by XTI applications. Two means are specified for reporting these events to the application:

- A request to t_look() call

- An exception (in the form of a [TLOOK] error return) during some XTI calls

The TLOOK error serves a special purpose in the transport service interface. It notifies the user that an event has occurred. As such, TLOOK does not indicate an error with a transport service interface routine, but the normal processing of that routine will not be performed because of the pending event.

The t_look() call provides a means to peek (without consuming) the events, except for the T_GODATA and T_GOEXDATA events that are consumed, that have been generated by the transport provider. The order of event reporting by t_look() is systems dependent.

Nine asynchronous events are defined in the transport service interfaces for both connection and connectionless mode services. The events defined are as follows:

T_LISTEN
 A request for a connection (connect indication) has arrived at the transport endpoint.

T_CONNECT
 A connect confirmation of a previously sent connect request has arrived at the transport endpoint. A connect confirmation is generated when a server accepts a connect request.

T_DATA
 User data has arrived at the transport endpoint.

T_EXDATA
 Expedited user has arrived at the transport endpoint.

T_DISCONNECT

A notification that the connection was aborted or that the server rejected a connect request. This is known as the disconnect indication.

T_ORDREL

A request for the orderly release of a connection has arrived at the transport endpoint. This is known as the orderly release indication.

T_UDERR

The notification of an error in a previously sent datagram has arrived at the transport endpoint. This is known as unit data error indication.

T_GODATA

An indication that flow control restrictions on normal data have been removed.

T_GOEXDATA

An indication that flow control restrictions on expedited data have been removed.

As shown in the state tables of Appendix C, it is possible in some states to receive one of several asynchronous events. The t_look() routine 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 is a disconnect indication. The client exits if its request is rejected.

The t_look() function is the only XTI call that reports events. It provides a means for applications to poll for occurrence of events at a transport endpoint. Any of the above events can be reported in t_look(). Because it is a local management function only, no information is sent over the network.

You can use the t_look() function with XTI calls operating in the synchronous or asynchronous mode. You can issue it to find out what happened at a transport endpoint, before issuing the appropriate XTI call. Upon immediate return from an asynchronous XTI call, t_look() can also be used to poll for the appropriate event before reissuing the asynchronous XTI call.

Although t_look() facilitates event-driven applications, it does not invoke the application automatically when a specific event occurs.

4.5 Error Reporting

There are two levels at which errors are defined:

- Library level

- System level

System level errors are errors resulting from the operating system routines that are invoked by the XTI library implementation. These errors result in having the XTI library setting t_errno() to [TSYSERR] and the external variable *errno* containing the value of the system error.

Library level errors are errors resulting from invalid input parameters or the function being called out of state. An external integer, *t_errno*, defined in <xti.h>, reflects the type of error. The errors reported are caused by:

- Input parameters that are illegal or out-of-bounds

- The function being invoked in the wrong sequence

- Lack of permission to execute the operation required by the function

- Events occurring while the function is executing in the asynchronous mode

The *t_errno* function is used to print out a message describing the last error encountered during a call to a transport library function. This call provides local management functions only, because no information is sent over the network.

State Transitions A

A.1 States and Events in XTI

The tables in this appendix describe the possible states of the transport provider as seen by the transport user, the incoming and outgoing events that may occur on any connection, and identify the allowable sequence of function calls. Given a current state and event, the transition of the next state is shown, as well as any actions that must be taken by the transport user.

Note

The t_error() function and the support functions, t_getstate(), t_getinfo(), t_alloc(), t_free(), t_look(), and t_sync() are excluded from the state tables, because they do not affect the state of the interface. Each of these functions may be issued from any state of the interface except the uninitialized state.

A.1.1 Transport Service Interface States

Table A-1 lists all possible states of the transport provider as seen by the transport user. The transport service interface manages a transport endpoint by using, at most eight states. The service type may be connection-mode (T_COTS), connection-mode with orderly release (T_COTS_ORD), or connectionless-mode (T_CLTS).

Table A-1: Transport Service Interface States

State	Description	Service Type
T_UNINIT	Uninitialized - initial and final state of the interface	T_COTS T_CLTS T_COTS_ORD
T_UNBND	Unbound	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 active user	T_COTS T_COTS_ORD
T_INCON	Incoming connection pending for passive user	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

A.1.2 Outgoing Events

The outgoing events listed in Table A-2 correspond to the successful return of the user-level transport functions, where these functions send a response to the transport provider.

As shown in Table A-2, some events (for example, acceptX are distinguished by the context in which they occur. The context is based on the values shown in Table A-3.

Table A-2: Outgoing Events

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_snddis	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

Table A-3 lists the values that affect the events listed in Table A-2.

Table A-3: Context Values for Table A-2

Value	Description
ocnt	Count of outstanding connect indications (connect indications passed to the user but not accepted or rejected by the user), only meaningful for the listening transport endpoint
fd	File descriptor of the current transport endpoint
resfd	File descriptor of the transport endpoint where a connection will be accepted

A.1.3 Incoming Events

Table A-4 lists incoming events, except for `pass_conn`, that correspond to the successful return of the specified user-level transport functions, where these functions retrieve data or event information from the transport provider. The `pass_conn` event is not associated directly with the return of a function on a given transport endpoint.

The `pass_conn` event occurs when a user transfers a connection to another transport endpoint. This event occurs on the endpoint that is being passed the connection, despite the fact that no function is issued on that endpoint. The `pass_conn` event is included in the state tables to describe what happens when a user accepts a connection on another transport endpoint.

Notice in Table A-4 that the `rcvdisX` 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 current transport endpoint.

Table A-4: Incoming Events

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
rcvudat	Successful return of t_rcvudata	T_CLTS
rcvuderr	Successful return t_rcvuderr	T_CLTS
pass_conn	Receive a passed connection	T_COTS T_COTS_ORD

A.1.4 Transport User Actions

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 follows:

[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.1.5 State Tables

Tables A-5 describes the possible next states, given the current state and event. 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 specfied in Table A-7. The transport user must take the specific actions in the order specified in the state table.

Table A-5: Common Local Management State Table

Event	T_UNINIT State	T_UNBND State	T_IDLE State
opened	T_UNBND		
bind		T_IDLE[1]	
optmgmt			T_UNBND
unbind			T_UNBND
closed		T_UNINIT	T_UNITIT

Table A-6 describes the possible next states, given the current state and event. 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 specfied in Table A-7. The transport user must take the specific actions in the order specified in the state table.

Table A-6: Connectionless-Mode State Table

Event	T_IDLE State
sndudata	T_IDLE
rcvudata	T_IDLE
rcvuderr	T_IDLE

Along with the next state, each box may include an action list as specfied in Table A-7. The transport user must take the specific actions in the order specified in the state table.

Table A-7: Connection-Mode State Table

Event	T_IDLE	T_OUTCON	T_INCON	T_DATAXFER	T_OUTREL	T_INREL
connect1	T_DATAXFER					
connect2		T_DATAXFER				
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				TDATAXFER		T_INREL
snddis1		T_IDLE	T_IDLE[3}	T_IDLE	T_IDLE	T_IDLE
snddis2			T_IDLE[3]			
rcvdis1		T_IDLE		T_IDLE	T_IDLE	T_IDLE
rcvdis2			T_IDLE[3]			
rcvdis3			T_INCON[3]			
sndrel				T_INREL	T_IDLE	
rcvrel				T_INREL	T_IDLE	
pass_conn	T_DATAXFER					
closed	T_UNINIT	T_UNINIT	T_UNINIT	T_UNINIT	T_UNINIT	T_UNINIT

A.1.6 Events and TLOOK Error Indication

Table A-8 lists the asynchronous events that cause an XTI call to return with a
[TLOOK] error.

Table A-8: Asynchronous Events That Returns a [TLOOK] Error

XTI Call	Asynchronous Events	Comment
t_accept:	T_DISCONNECT, T_LISTEN	
t_connect:	T_DISCONNECT, T_LISTEN	Occurs only when t_connect is on an endpoint that has been bound with a *qlen* = 0 and for which a connect indication is pending.
t_listen:	T_DISCONNECT	This event indicates a disconnect has occurred on an outstanding connect indication.
t_rcv:	T_DISCONNECT, T_ORDREL	
t_rcvconnect:	T_DISCONNECT	
t_rcvrel:	T_DISCONNECT	
t_rcvudata:	T_UDERR	
t_snd:	T_DISCONNECT, T_ORDREL	
t_sndudata:	T_UDERR	
t_unbind:	T_LISTEN	
t_sndrel:T_DISCONNECT		

When a [TLOOK] error has been received on a transport endpoint by means of an XTI function, subsequent calls to that and other XTI functions to which the same [TLOOK] error applies, continue to return [TLOOK] until the event is consumed. An event causing the [TLOOK] error can be determined by calling t_look (), and can then be consumed by calling the corresponding consuming XTI function.

Guidelines for Writing Protocol-Independent Software B

Protocol-independent applications are applications that can run over several transport providers without significant changes.

B.1 Amount of Required Changes

The number of changes required depends upon the following factors:

- Extent of transport services required by the application
- Functional compatibility of the transport providers
- Availability of optional XTI functions for examination and negotiation of transport options

Each transport provider should provide most, if not all, of the transport services required by the application. Deficiencies in this area may require enhancements in the application.

Transport providers that are functionally equivalent often have similar transport characteristics. Thus, default characteristics set by the underlying transport protocols may be sufficient for application portability. On the other hand, if the default characteristics between the transport providers differ greatly, the user may enhance the application or negotiate protocol options with the providers. Optional XTI functions such as t_optmgmt() can be used for this purpose.

B.2 General Rules

In order to maximize portability of XTI applications between different kinds of machines and to support protocol independence, you should follow these general rules:

- An application should make use only of these functions and mechanisms described as being mandatory features of XTI. This assumes that the default transport services offered are adequate for application support.

- 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.

- The transport provider identifier should not be hard-coded into the application. While software may be written for a particular class of service (for example, connectionless-mode service), it should not be written to depend on any attribute of the underlying protocol.

- The protocol-specific service limits returned on the t_open() and t_getinfo() functions must not be exceeded. It is the responsibility of the user to access these limits and then adhere to the limits throughout the communication process.

- The user program should not look at or change options that are specific to the underlying protocol. The t_optmgmt() function enables a user to access default protocol options from the transport provider, which can then be blindly passed as an argument on the appropriate connection establishment function. Optionally, the user can choose not to pass options as an argument on connect establishment functions.

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

- Protocol-specific addressing issues should be hidden from the user program. Similarly, the user must have some way of accessing a destination address in an invisible manner, such as through a name server.

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

- Optional orderly release facility of the connection-mode service, for example, t_sndrel() and t_rcvrel(), should not be used by programs targetted for multiple protocol environments. This facility is not supported by all connection-based transport protocols.

Migrating from Socket-Based Software to XTI-Based Software   C

This appendix contains information on migrating from socket-based software to XTI-based software. Table C-1 lists an example of the call sequences issued by an active TCP user. Table C-2 lists an example of the call sequences issued by a passive TCP user which communicates with the active TCP user in Table C-1. Table C-3 lists an example of the call sequences issued by a UDP user.

Table C-1 lists an example of the call sequences issued by an active user.

Table C-1: Active TCP User

Socket Level Calls	XTI Calls
s=socket (af, type, protocol)	fd = t_open (name, oflag, info) name that corresponds to <af, type, protocol> is provided in <xti.h> oflag = O_RDWR
bind (s, sockname, namelen)	t_bind (fd, req, ret) req->addr.len = (unsigned int) namelen req->addr.buf = (char *) sockname req->qlen = (unsigned) 0
ret->addr.maxlen = (unsigned int)	struct sockaddr_in) ret->addr.buf = &<local socket>
connect (s, name, namelen)	t_connect (fd, sndcall, rcvcall) sndcall->addr.len = (unsigned int) namelen sndcall->addr.buf = (char *) name sndcall->opt.len = (unsigned int) sizeof (struct tcp_options) sndcall->opt.buf = &<tcp options> sndcall->udata.len = 0
nc = snd (s, msg, len, sflags)	cc = t_snd(fd, msg, len, tflags) tflags = T_EXPEDITED if sflags is set to MSG_OOB
close (s)	t_close (fd)

Table C-2 lists an example of the call sequences issued by a passive TCP user that communicates with the active TCP user in Table C-1.

Table C-2: Passive TCP User

Socket Level Calls	XTI Calls
s = socket (af, type, protocol)	fd = t_open (name, oflag, info) Name which corresponds to <af, type, protocol> is provided in <xti.h> oflag = O_RDWR
bind (s, sockname, nameln)	t_bind (fd, req, ret) req->addr.len = (unsigned int) nameln req->addr.buf = (char *) sockname req->qlen = (unsigned) backlog where backlog is input to listen ret->addr.maxlen = (unsigned int)(struct sockaddr_in) ret->addr.buf = addr.buf = &<local socket>
setsockopt (s, IPPROTO_TCP), TCP_ACCEPTMODE, &acc_mode, sizeof(acc_mode)) int acc_mode = ACC_DEFER	
listen (s, backlog)	t_listen (fd, call) call->addr.maxlen = (unsigned int) (struct sockaddr_in) call->addr.buf = &<remote socket> call->opt.maxlen = (unsigned int) sizeof(struct tcp_options) call->opt.buf =&<remote options> call->udata.maxlen = 0
ns = accept (s, addr, addrlen)	
setsockopt (ns, IPPROTO_TCP TCP_CONACCEPT, 0, 0)	t_accept (fd, resfd, call) call->addr.len = (unsigned int) (struct sockaddr_in) call->addr.buf = &<remote socket> call->opt.len = (unsigned int) sizeof(struct tcp_options) call->opt.buf = &<tcp options> call->udata.len = 0 call->sequence = <sequence number returned in t_listen>
cc = recv (ns, buf, len, flags)	nc = t_rcv (resfd, buf, len, rflags)
close (ns)	t_close (resfd)

Note

If resfd != fd, resfd must be obtained by means of a t_open() call and the t_bind() call must be issued with *qlen* = 0. On output, *rflags*, if the type of data received matches that given by flags in the recv call.

Table C-3 lists an example of the call sequences issued by a UDP user.

Table C-3: UDP User

Socket Level Calls	XTI Calls
s = socket (af, type protocol)	fd = t_open (name, oflag, info) Name that corresponds to <af, type, protocol> is provided in <xti.h> oflag = O_RDWR
bind (s, sockname, namelen)	t_bind (fd, req, ret) req->addr.len = (unsigned int) namelen req->addr.buf = (char *) sockname req->qlen = (unsigned) 0 ret->addr.maxlen = (unsigned int) (struct sockaddr_in) ret->addr.buf = &local socket>
cc = sendto (s, msg, len,	t_sndudata (fd,unitdata) flags, to, tolen) unitdata->addr.len = (unsigned int) tolen unitdata->addr.buf = to unitdata->opt.len = 0 unitdata->udata.len = len unitdata->udata.buf = msg
cc = recvfrom (s, buf, len, flags) from, fromlen)	t_rcvudata (fd, unitdata, flags, unitdata->addr.buf = from unitdata->opt.maxlen = 0 unitdata->udata.maxlen = (unsigned int) len unitdata->udata.buf = buf
close (s)	t_close (fd)

Connection-Mode Programming Examples   D

This appendix contains the entire connection-mode client and server code from which
the examples used in Chapter 2 were taken.

D.1   Connection-Mode Client Programming Example

Example D-1 shows how the client establishes a transport connection with the server
and then exchanges data with the server. The connection is released using the orderly
release facility of the transport service interface.

Example D-1: Connection-Mode Client Code

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <netinet/in.h>
#include <stdio.h>
#include <ctype.h>
#include <errno.h>
#include <signal.h>
#include <setjmp.h>
#include <netdb.h>
#include <xti.h>
#include <fcntl.h>

extern int errno;
int net;
struct t_info t_open_info;     /* transport char. from transport */
struct t_info t_getinfo_info;
struct tcp_options tcp_opts;
struct t_optmgmt t_optm_req;
struct t_optmgmt t_optm_ret;
struct sockaddr_in sin;
struct     servent *sp;
char *hostname;
struct hostent *host;
#define MAXDSIZE 512
char snd_buf[MAXDSIZE];
char rcv_buf[MAXDSIZE];
int n;
int status;
struct t_call t_conn_sndcall;
struct t_call t_conn_rcvcall;
struct t_call t_rcvconn_call;

struct t_discon discon;
int t_rcv_flags;

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

Example D-1: (continued)

```
{

    char destin[255];

    if ((net = t_open("tcp", O_RDWR|O_NONBLOCK, &t_open_info)) < 0) {
      t_error("t_open failed");
      exit(t_errno);
    }

    status = t_getinfo(net, &t_getinfo_info);


    /*
     * t_bind - bind an address to a transport endpoint
     *
     */

    if (t_bind(net, 0, 0) < 0) {
      t_error("iexample: t_bind error");
      exit(1);
    }

    t_optm_req.opt.len = 0;
    t_optm_req.flags = T_DEFAULT;
    t_optm_ret.opt.maxlen = sizeof(struct tcp_options);
    t_optm_ret.opt.buf = (char *) &tcp_opts;

    status = t_optmgmt(net, &t_optm_req, &t_optm_ret);
    if (status < 0) {
      t_error("iexample: t_optmgmt error");
      exit(1);
    }
    t_optm_req.opt.len = 0;
    t_optm_req.flags = T_DEFAULT;
    t_optm_ret.opt.maxlen = sizeof(struct tcp_options);
    t_optm_ret.opt.buf = (char *) &tcp_opts;

    status = t_optmgmt(net, &t_optm_req, &t_optm_ret);
    if (status < 0) {
      t_error("iexample: t_optmgmt error");
      exit(1);
    }

    printf("host :");
    scanf("%s",destin);

    host = gethostbyname(destin);

    if (host) {
      sin.sin_family = host->h_addrtype;
      bcopy(host->h_addr, (caddr_t)&sin.sin_addr, host->h_length);
      hostname = host->h_name;
    }

    sin.sin_port = 200; /* try to connect to port 200 */
    t_conn_sndcall.addr.len = sizeof (struct sockaddr_in);
    t_conn_sndcall.addr.buf = (char *) &sin;
    t_conn_sndcall.opt.len = 0;
    t_conn_sndcall.udata.len = 0;
    t_conn_rcvcall.addr.maxlen = sizeof (struct sockaddr_in);
    t_conn_rcvcall.addr.buf = (char *) &sin;
    t_conn_rcvcall.opt.maxlen = sizeof(struct tcp_options);
    t_conn_rcvcall.opt.buf = (char *) &tcp_opts;
```

Example D-1: (continued)

```
t_conn_rcvcall.udata.maxlen = 0;
t_rcvconn_call.addr.maxlen = sizeof (struct sockaddr_in);
t_rcvconn_call.addr.buf = (char *) &sin;
t_rcvconn_call.opt.maxlen = sizeof(struct tcp_options);
t_rcvconn_call.opt.buf = (char *) &tcp_opts;
t_rcvconn_call.udata.maxlen = 0;
t_rcvconn_call.udata.buf = 0;

if ((t_connect(net, &t_conn_sndcall, &t_conn_rcvcall)) < 0) {
  if (t_errno == TNODATA) {
    while (1) {
    status = t_rcvconnect(net, &t_rcvconn_call);

    if (status < 0) {
      if (t_errno == TLOOK) {
        printf("Event %x came in\n",t_look(net));
        (void) t_unbind(net);
        (void) t_close(net);
        exit(1);
      }
      if (t_errno != TNODATA) {
        t_error("iexample: t_rcvconnect()");
        (void) t_unbind(net);
        (void) t_close(net);
        exit(1);
      }
    }
    else
      break;
    }
  } else {
    t_error("iexample: t_connect()");
    (void) t_unbind(net);
    (void) t_close(net);
    exit(1);
  }
}

printf("calling t_snd with %d bytes of regular data\n",sizeof(snd_buf));
n = t_snd(net, &snd_buf[0],sizeof(snd_buf) , 0);

if (n < 0) {
  if (t_errno == TLOOK) {
    printf("Generated a %X TLOOK error\n",t_look(net));
    (void) t_unbind(net);
    (void) t_close(net);
    exit(1);
  }
  t_error("iexample: t_snd error");
  (void) t_unbind(net);
  (void) t_close(net);
  exit(1);
}
printf("t_snd sent %d bytes\n",n);

while (1) {
  n = t_rcv(net, rcv_buf, sizeof(rcv_buf), &t_rcv_flags);

  if (n < 0) {
    if (t_errno != TNODATA) {
    t_error("iexample: t_rcv error");
    (void) t_unbind(net);
    (void) t_close(net);
```

Example D-1: (continued)

```
    exit(1);
     }
     else {
     t_error("iexample: NO data available");
     }
   }
   if (n > 0) break;
}

printf("t_rcv received %d bytes\n",n);

if (t_rcv_flags & T_EXPEDITED)
  printf("data is expedited\n");
else
  printf("data is normal\n");

n = t_sndrel(net, (struct t_call *) 0);

if (n < 0) {
  t_error("iexample: error in t_sndrel:");
  t_unbind(net);
  t_close(net);
  exit(1);
}

while (1) {
  n = t_rcvrel(net);

  if (n < 0) {
    if (t_errno != TLOOK && t_errno != TNOREL) {
    t_error("iexample: error in t_rcvrel:");
    t_unbind(net);
    t_close(net);
    exit(1);
     }
     else {
     if (t_errno == TNOREL)
       t_error("iexample: NO T_ORDREL available");
     else {
       t_error("iexample: TLOOK event");
       t_unbind(net);
       t_close(net);
       exit(1);
    }
     }
   }
   if (n == 0) break;
  }
  t_unbind(net);
  t_close(net);
  exit(0);
}
```

D.2 Connection-Mode Server Programming Example

Example D-2 shows how the server establishes a transport connection with a client and then exchanges data with the client on the other side of the connection. The connection is released using the orderly release facility of the transport service interface.

Example D-2: Connection-Mode Server Code

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <sys/file.h>
#include <netinet/in.h>
#include <stdio.h>
#include <signal.h>
#include <errno.h>
#include <sgtty.h>
#include <netdb.h>
#include <syslog.h>
#include <xti.h>

int    net,net1,n,n1;
extern     int errno;

main(argc, argv)
     char *argv[];
{
     int fromlen;
     struct sockaddr_in from;

     int status;

     status =  get_income();
     if (status != 0)
       exit(1);
     else {
       sleep(10);
       exit(0);
         }
}


doit(f, seq)
     int f,seq;
{
  int t_rcv_flags;
  struct hostent *hp;
  char rcv_buf[512];
  char snd_buf[512];
  int n;


  while (1) {
    n = t_rcv(f,rcv_buf, sizeof(rcv_buf) ,&t_rcv_flags);

    if (n < 0) {
      if (t_errno != TNODATA) {
      t_error("rexample: t_rcv error");
      t_unbind(f);
      t_close(f);
      exit(1);
       }
       else {
      t_error("rexample: NO data available");
       }
    }
    if (n > 0) break;
  }

  printf("t_rcv received %d bytes\n",n);
```

Example D-2:   (continued)

```
if (t_rcv_flags & T_EXPEDITED)
  printf("data is expedited\n");
else
  printf("data is normal\n");

printf("calling t_snd with %d bytes of regular data\n",sizeof(snd_buf));
n = t_snd(f, &snd_buf[0],sizeof(snd_buf) , 0);

if (n < 0) {
  if (t_errno == TLOOK) {
    printf("Generated a %X TLOOK error\n",t_look(f));
    (void) t_unbind(f);
    (void) t_close(f);
    exit(1);
  }
  t_error("rexample: t_snd error");
  (void) t_unbind(f);
  (void) t_close(f);
  exit(1);
}
printf("t_snd sent %d bytes\n",n);

while (1) {
  n = t_rcvrel(f);

  if (n < 0) {
    if (t_errno != TLOOK && t_errno != TNOREL) {
   t_error("rexample: error in t_rcvrel:");
   t_unbind(f);
   t_close(f);
   exit(1);
    }
    else {
   if (t_errno == TLOOK) {
     t_error("TLOOK error");
     t_unbind(f);
     t_close(f);
     exit(1);
   }
   t_error("rexample: NO T_ORDREL available");
    }
  }
  if (n == 0) break;
}

n = t_sndrel(f, (struct t_call *) 0);

if (n < 0) {
  t_error("rexample: error in t_sndrel:");
  t_unbind(f);
  t_close(f);
  exit(1);
}

t_unbind(f);
t_close(f);
exit(0);
}


int
get_income()
```

Example D-2: (continued)

```
{
    struct sockaddr_in sname;
    struct servent *sp;
    int i;
    int child;

    struct t_call t_list_call;
    struct t_call *t_list_ptr;
    struct t_call t_snddis_call;
    struct t_bind t_bind_addr_req;
    struct t_bind t_bind_addr_req1;
    struct t_bind t_bind_addr_ret;
    struct t_info t_open_info;    /* transport char. from transport */
    int t_status;

    /*
     * Call t_open - establish a transport endpoint
     *
     */

    if ((net = t_open("tcp", O_RDWR, &t_open_info)) < 0) {
        t_error("rexample: t_open error");
        exit(1);
    }

    /*
     * t_bind - bind an address to a transport endpoint
     *
     */

    sname.sin_port = 200;            /* load port # */
    sname.sin_family = AF_INET;
    sname.sin_addr.s_addr = 0;


    t_bind_addr_req.addr.len = sizeof (struct sockaddr_in);
    t_bind_addr_req.addr.buf = (char *) &sname;
    t_bind_addr_req.qlen = 1;
    t_bind_addr_ret.addr.maxlen = sizeof (struct sockaddr_in);
    t_bind_addr_ret.addr.buf = (char *) &sname;


    if ((t_bind(net, &t_bind_addr_req, &t_bind_addr_ret)) < 0) {
        t_error("rexample: t_bind error");
        exit(1);
    }


    t_list_ptr = (struct t_call *) t_alloc(net, T_CALL_STR, T_ADDR);
    bcopy(&sname, t_list_ptr->addr.buf, t_list_ptr->addr.maxlen);

    t_status = t_listen(net, t_list_ptr);

    if (t_status < 0) {
        if (t_errno != TNODATA) {
            t_error("rexample: t_listen error");
            t_unbind(net);
            t_close(net);
            exit(1);
        }
    }

    printf("Have a incomming connection with sequence # %d\n",
```

Example D-2:   (continued)

```
        t_list_ptr->sequence);
    printf("attempting to accept sequence # %d\n",
            t_list_ptr->sequence);

    net1 = get_endpoint();
    if (t_status = t_accept(net,net1,t_list_ptr) < 0) {
      t_error("rexample: t_accept error");
      if (t_errno == TLOOK) {
        printf("event %x came in\n",t_look(net1));
      }
      exit(1);
    }

    fcntl(net1,F_SETOWN, getpid());
    child = fork();

    if (child == 0) {
      t_unbind(net);
      t_close(net);
      t_sync(net1);
      doit(net1, t_list_ptr->sequence);
    }
    else
      {
        printf("Forking Child process =%d for fd = %d seq=%d\n",
            child,net1, t_list_ptr->sequence);
        t_unbind(net1);
        t_close(net1);
        t_free(t_list_ptr, T_CALL_STR);
      }
    return(0);
}

int
get_endpoint()
{
  struct sockaddr_in sname;
  struct servent *sp;
  int tmp_net;

  struct t_call t_list_call;
  struct t_bind t_bind_addr_req;
  struct t_bind t_bind_addr_req1;
  struct t_bind t_bind_addr_ret;
  struct t_info t_open_info;    /* transport char. from transport */
  int t_status;


/*
 * Call t_open - establish a transport endpoint
 *
 */

  if ((tmp_net = t_open("tcp", O_RDWR, &t_open_info)) < 0) {
    t_error("rexample: t_open error");
    exit(1);
  }

  /*
   * t_bind - bind an address to a transport endpoint
   *
   */
```

Example D-2: (continued)

```
sname.sin_port = 0;
sname.sin_family = AF_INET;
sname.sin_addr.s_addr = 0;


t_bind_addr_req.addr.len = sizeof (struct sockaddr_in);
t_bind_addr_req.addr.buf = (char *) &sname;
t_bind_addr_req.qlen = 0;
t_bind_addr_ret.addr.maxlen = sizeof (struct sockaddr_in);
t_bind_addr_ret.addr.buf = (char *) &sname;


if ((t_bind(tmp_net, &t_bind_addr_req, &t_bind_addr_ret)) < 0) {
  t_error("rexample: t_bind error");
  exit(1);
}
return(tmp_net);
}
```

Connectionless-Mode Programming Examples E

This appendix contains the entire connectionless-mode server code from which the examples used in Chapter 3 were taken. It also contains a connectionless-mode client code example.

E.1 Connectionless-Mode Server Programming Example

Example E-1 shows how the server waits for incoming datagram queries and then processes each query.

As was mentioned in Chapter 3, the client-server relationship between two users does not exist in the connectionless-mode service. It is only within context of the example that the term is used because the transport service interface does not support this relationship.

Example E-1: Connectionless-Mode Server Code

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <sys/file.h>
#include <netinet/in.h>
#include <stdio.h>
#include <signal.h>
#include <errno.h>
#include <sgtty.h>
#include <netdb.h>
#include <syslog.h>
#include <xti.h>

struct sockaddr_in sname;
int    net,ncc;
extern     int errno;
extern  void do_setup();
struct t_unitdata unitdata;

main(argc, argv)
      char *argv[];
{
      do_setup();
      doit(net);
}


doit(f)
      int f;
{
   int t_rcv_flags;
   struct hostent *hp;
   char rcv_buf[5120];
```

Example E-1: (continued)

```
  struct sockaddr sname1;

  unitdata.addr.maxlen = sizeof(sname1);
  unitdata.addr.buf = (char *) &sname1;
  unitdata.opt.maxlen = 0;
  unitdata.opt.buf = 0;
  unitdata.udata.maxlen = sizeof(rcv_buf);
  unitdata.udata.buf = &rcv_buf[0];

  ncc = t_rcvudata(f,&unitdata,&t_rcv_flags);

  if (ncc == 0)
    printf("received %d octets\n",unitdata.udata.len);
  else
    printf("ncc = %d, errno =%d\n",ncc,errno);
  (void) t_close(f);
  exit(0);
}

void
  do_setup()
{
  struct t_call t_list_call;
  struct t_bind t_bind_addr_req;
  struct t_bind t_bind_addr_req1;
  struct t_bind t_bind_addr_ret;
  struct t_info t_open_info;   /* transport char. from transport */
  int t_status;


/*
 * Call t_open - establish a transport endpoint
 *
 */

  if ((net = t_open("udp", O_RDWR, &t_open_info)) < 0) {
    t_error("rexamless: t_open error");
    exit(1);
  }


  /*
   * t_bind - bind an address to a transport endpoint
   *
   */

  sname.sin_port = 200;
  sname.sin_family = AF_INET;

  t_bind_addr_req.addr.len = sizeof (struct sockaddr_in);
  t_bind_addr_req.addr.buf = (char *) &sname;
  t_bind_addr_req.qlen = 1;
  t_bind_addr_ret.addr.maxlen = sizeof (struct sockaddr_in);
  t_bind_addr_ret.addr.buf = (char *) &sname;


  if ((t_bind(net, &t_bind_addr_req, &t_bind_addr_ret)) < 0) {
    t_error("rexamless: t_bind error");
    exit(1);
  }
}
```

E.2 Connectionless-Mode Client Programming Example

Example E-2 represents the client-side (user) that would communicate with the server-side (user) as represented by the code under the previous section: Connectionless-Mode Server. This code is not found in Chapter 3.

Example E-2: Connectionless_mode Client Code

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <netinet/in.h>
#include <stdio.h>
#include <ctype.h>
#include <errno.h>
#include <signal.h>
#include <setjmp.h>
#include <netdb.h>
#include <xti.h>
#include <fcntl.h>

int  net;
extern    int errno;
struct sockaddr_in sin;
char *hostname;
char hnamebuf[32];
struct t_call t_conn_sndcall;
struct t_call t_conn_rcvcall;
struct t_info t_open_info;   /* transport char. from transport */
struct t_unitdata unitdata;
int t_rcv_flags;
char snd_buf[6000];
char rcv_buf[6000];
struct hostent *host;
int scc,n;

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

  host = gethostbyname("nil");

  if (host) {
    sin.sin_family = host->h_addrtype;
    bcopy(host->h_addr, (caddr_t)&sin.sin_addr, host->h_length);
    hostname = host->h_name;
  }

  sin.sin_port = 0;            /* don't set port till time to do connect */


/*
 * Call t_open - establish a transport endpoint
 *
 */

  if ((net = t_open("udp", O_RDWR, &t_open_info)) < 0) {
    t_error("iexamless: t_open error");
    return(1);
  }
```

Example E-2: (continued)

```
/*
 * t_bind - bind an address to a transport endpoint
 *
 */

  if ((t_bind(net, 0, 0)) < 0) {
    t_error("iexamless: t_bind error");
    exit(1);
  }

  sin.sin_port = 200;
  unitdata.addr.len = sizeof(sin);
  unitdata.addr.buf = (char *) &sin;
  unitdata.opt.len = 0;
  unitdata.udata.len = sizeof(snd_buf);
  unitdata.udata.buf = snd_buf;
  unitdata.opt.len = 0;

  n = t_sndudata(net, &unitdata);

  if (n < 0) {
    if (t_errno != TNODATA) {
      t_error("iexamless: t_sndudata error");
      (void) t_close(net);
      exit(1);
    }
  }

  t_close(net);
  exit(0);
}
```

Glossary

abortive release
> A connection termination that breaks a connection immediately and may result in the loss of any data that has not reached the destination user.

asynchronous mode
> The mode of execution in which transport service interface routines do not block while waiting for specific asynchronous events to occur, but instead return immediately if the event is not pending.

client
> The transport user in connection-mode that initiates the establishment of a transport connection to a another transport user (server).

connection establishment
> The phase in connection-mode that enables two transport users to create a transport connection (virtual circuit) between them.

connection-mode
> A circuit-oriented mode of transfer that 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.

connectionless-mode
> A message-oriented mode that supports data transfer in self-contained units with no logical relationships required among multiple units.

connection release
> The phase in connection-mode that terminates a previously established connection and ends the data exchange between two transport users.

datagram
> A unit of data transferred between two transport users during the connectionless-mode.

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

ETSDU
> An acronym for Expedited Transport Service Data Unit. ETSDU is the maximum expedited data message size that may be sent over a transport connection.

expedited data

Data that are considered urgent. The transport protocol that provides the transport service defines the specific semantics for the expedited data.

initialization

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.

orderly release

A procedure in connection-mode to gracefully terminate a transport connection with no loss of data.

peer user

The user with whom a given user is communicating above the transport service interface.

protocol address

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

server

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

service request

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

synchronous mode

The mode of execution in which an application normally blocks until completion. For example, an application making a synchronous `t_rcv()` call will block until data from over the network can be retrieved.

T_CLTS

An acronym for Transport ConnectionLess Transport Service. T_CLTS means that the transport provider supports connectionless-mode service.

T_COTS

An acronym for Transport Connection Oriented Transport Service. T_COTS means that the transport provider supports connection-mode service but does not provide the optional orderly release facility.

transport addresss

See protocol address definition.

transport connection

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

transport endpoint

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

transport service interface
> A set of transport-independent C library functions that support the services of a transport interface. These functions conform to the X/Open Transport Interface Specifications.

transport provider
> The transport protocol that provides the services of the transport service interface.

transport service data unit
> The amount of user data whose identity is preserved from one end of a transport connection to the other.

transport user
> The user-level application or protocol that accesses the services of the transport service interface.

TSDU
> An acronym for Transport Service Data Unit. TSDU is the maximum message size that may be transmitted in either connection-mode or connectionless-mode.

virtual circuit
> A transport connection established in connection mode.

Index

A

aborting
connection, 2–19
address
client, 2–5
applications
migrating, C–1
portability, B–1
portability rules, B–1
protocol independent, B–1
asynchronous mode
description of, 2–4, 2–13
events, 2–13

B

binding
address, 2–9
address to endpoint, 2–5
transport address, 3–3
binding address
required state, 2–15
bound address of, 2–10
buffers
allocating, 2–10, 2–15
flags, 3–4
maximum size of, 2–13
netbuf, 2–10
size of, 2–15, 2–17, 2–19
user data, 2–17

C

calling functions
legal sequence, 1–10
state tables, 1–10
client
addresss of, 2–5
communication path
establishing, 1–4, 2–1
connect indication
processing, 2–15
connect indications
listening, 2–15
listening for, 2–2, 2–9
maximum number of, 2–9
outstanding, 2–9
queueing, 2–9
connection
aborting, 2–19
accepting or rejecting, 2–10
establishing, 1–4, 2–15
establishment, 1–5
initiating, 2–10
multiple units, 2–16
orderly release, 2–19
release, 2–19
requirement for, 2–15
connection release
abortive, 1–6
orderly, 1–7
connectionless-mode
communication path, 1–8
data transfer, 1–9
description, 1–3
initialization functions, 1–9

S

server

accepting request, 1–6

description, 1–5

identity, 1–5

notify request, 1–6

service

advertising, 1–5

synchronizing

transport endpoint, 2–16

T

t_accept(), 2–10, 2–15

t_alloc(), 2–10, 2–15, 2–16, 4–3

t_bind(), 1–4, 1–9, 2–2, 2–5, 2–9, 2–10, 3–1, 3–3

t_call structure, 2–12, 2–15

t_close(), 1–7, 1–10, 2–20, 3–5

t_connect(), 1–5, 2–10, 2–12, 2–13

tcp_options structure, 4–2

t_errno(), 4–5

t_error(), 2–5

t_free(), 4–3

t_getinfo(), 2–5, 4–1

t_info structure, 4–1

t_listen(), 1–6, 2–10, 2–15

t_look(), 4–4

t_open(), 1–4, 1–8, 2–1, 2–4, 2–5, 2–8, 2–16, 3–1, 3–3, 4–1

t_open_info structure, 2–4

t_optmgmt(), 2–2, 2–5, 4–1

transport address

actual, 3–3

binding, 3–3

transport endpoint

assigning address, 2–5

assigning an address, 1–4

associated address, 1–4, 2–2, 2–5

binding address, 2–9, 2–15

binding to, 2–2

closing, 2–20

description, 1–3

disable, 3–5

transport endpoint (cont.)

disabling, 1–7, 2–20

establishing, 1–4, 1–8, 2–8, 3–3

freeing, 1–7

identifying, 2–1, 2–13, 2–17, 2–19

identity, 2–1

manipulating, 1–3

number of bound addresses to, 2–2

synchronizing, 2–16

used for connection, 2–15

transport protocol

characteristics of, 2–1, 2–4, 3–3

transport provider

accepting connection, 2–2

address, 1–8

address structure, 1–4

characteristics, 4–1

characteristics of, 2–2, 4–1

default characteristics, B–1

description of, 1–1

establishing communication path, 2–1

establishing connection, 1–4

functions of, A–2

identifying, 2–4, 2–12, 3–3

identity, 1–4, 1–8, 3–1

passing data to, 2–12

protocol options, 4–1

quality of service, 1–3

returning information, 2–1, 2–12

service request of, 1–1

service types, 2–2

state tables, A–5, A–6

states, A–1

supported protocols, 1–1

urgent condition, 2–16

transport service interface

BSD IPC enhancements, 1–2

characteristics, 1–2

components, 1–10

consists of, 1–1

event handling, 1–2

transport user

actions, A–5

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-234-1998 using a 1200- or 2400-baud modem from anywhere in the USA, Canada, or Puerto Rico. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital Subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	————	Local Digital subsidiary or approved distributor
Internal*	————	SSB Order Processing - WMO/E15 or Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

* For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? _____

_____

_____

What do you like best about this manual? _____

_____

_____

What do you like least about this manual? _____

_____

_____

Please list errors you have found in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

_____

What version of the software described by this manual are you using? _____

Name/Title _____ Dept. _____

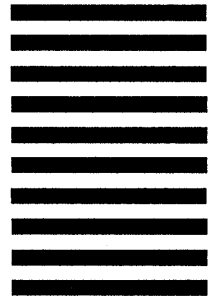Company _____ Date _____

Mailing Address _____

_____ Email _____ Phone _____

digital ™

BUSINESS REPLY MAIL
FIRST–CLASS MAIL PERMIT NO. 33 MAYNARD MA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3–2/Z04
110 SPIT BROOK ROAD
NASHUA NH 03062–9987

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? _____

_____

_____

What do you like best about this manual? _____

_____

_____

What do you like least about this manual? _____

_____

_____

Please list errors you have found in this manual:

Page   Description

_____   _____

_____   _____

_____   _____

_____   _____

_____   _____

Additional comments or suggestions to improve this manual:

_____

_____

_____

_____

What version of the software described by this manual are you using? _____

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Email _____ Phone _____

d i g i t a l ™

BUSINESS REPLY MAIL
FIRST–CLASS MAIL PERMIT NO. 33 MAYNARD MA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3–2/Z04
110 SPIT BROOK ROAD
NASHUA NH 03062–9987