

*SunLink<sup>®</sup> OSI 8.1  
TLI Programmer's Reference*

 *SunSoft*  
A Sun Microsystems, Inc. Business  
2550 Garcia Avenue  
Mountain View, CA 94043  
U.S.A.

Part No.: 801-7170-12  
Revision A, March 1995

© 1995 Sun Microsystems, Inc.  
2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A.

All rights reserved. This product and related documentation are protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or related documentation may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Portions of this product may be derived from the UNIX<sup>®</sup> and Berkeley 4.3 BSD systems, licensed from UNIX System Laboratories, Inc., a wholly owned subsidiary of Novell, Inc., and the University of California, respectively. Third-party font software in this product is protected by copyright and licensed from Sun's font suppliers.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the United States Government is subject to the restrictions set forth in DFARS 252.227-7013 (c)(1)(ii) and FAR 52.227-19.

The product described in this manual may be protected by one or more U.S. patents, foreign patents, or pending applications.

#### TRADEMARKS

Sun, the Sun logo, Sun Microsystems, Solaris and SunLink are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and certain other countries. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd. OPEN LOOK is a registered trademark of Novell, Inc. PostScript and Display PostScript are trademarks of Adobe Systems, Inc. All other product names mentioned herein are the trademarks of their respective owners.

All SPARC trademarks, including the SCD Compliant Logo, are trademarks or registered trademarks of SPARC International, Inc. SPARCstation, SPARCserver, SPARCengine, SPARCstorage, SPARCware, SPARCcenter, SPARCclassic, SPARCcluster, SPARCdesign, SPARC811, SPARCprinter, UltraSPARC, microSPARC, SPARCworks, and SPARCcompiler are licensed exclusively to Sun Microsystems, Inc. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK<sup>®</sup> and Sun<sup>™</sup> Graphical User Interfaces were developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

X Window System is a product of the Massachusetts Institute of Technology.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.



# Contents

---

## *Part 1 —SunLink OSI 8.1 TLI Guide*

<b>1. TLI Overview.</b>	<b>1</b>
TLI Architecture.	2
Devices to Open.	3
Modes of Service	4
Connection Mode Service	4
Connectionless Mode Service	5
Blocking and Non-Blocking Interactions	5
State Transitions.	6
Error Handling.	6
<b>2. Modes of Service.</b>	<b>7</b>
Connection Mode Services	8
Local Management Phase	9
Connection Establishment Phase.	11
Client Connection Request	12

---

Server Responsibilities.....	12
Data Transfer .....	15
Connection Release .....	16
Connectionless Mode .....	17
Local Management Phase .....	18
Data Transfer Phase.....	18
Error Handling.....	19
Memory Management.....	19
<b>3. Addressing.....</b>	<b>21</b>
Address Library Functions.....	22
Setting Addresses Explicitly.....	23
Transport Layer Addressing.....	23
RFC1006 Addressing.....	24
CLNS Addressing .....	25
<b>4. Non-Blocking Mode.....</b>	<b>27</b>
Setting Non-Blocking Mode .....	28
Non-Blocking Functions .....	28
t_open .....	29
t_connect and t_rcvconnect.....	29
t_listen.....	29
t_rcv and t_rcvudata .....	30
t_snd and t_sndudata .....	30
Using poll and select.....	30
<b>5. State Transition Tables.....</b>	<b>31</b>

---

TLI States. . . . .	32
Outgoing Events . . . . .	32
Incoming Events. . . . .	34
Transport User Actions . . . . .	34
State Tables . . . . .	35
<b>6. Compiling &amp; Protocol Independence . . . . .</b>	<b>37</b>
Compiling and Linking Programs . . . . .	37
Include Files . . . . .	38
Procedure for Compiling and Linking . . . . .	38
Protocol Independency Rules. . . . .	39
<b>7. Programming Examples . . . . .</b>	<b>41</b>
Argument List Example . . . . .	44
Address Mapping Example . . . . .	47
Connection Mode Client Example . . . . .	50
Connection Mode Server Example . . . . .	56
Connectionless Mode Client Example. . . . .	66
Connectionless Mode Server Example . . . . .	71
Non-Blocking Mode Client Example. . . . .	77
OSI Library Use Examples . . . . .	84
TLI over CLNS Example . . . . .	89
TLI over RFC1006 Examples. . . . .	94
Makefile Example . . . . .	112

*Part 2 —Function Call Reference*

---

<b>8. Using TLI Functions with SunLink OSI .....</b>	<b>117</b>
Summary of Call Order.....	118
Where to Find Descriptions .....	120
The <i>netbuf</i> Structure.....	121
Function Reference .....	122
t_accept.....	122
t_alloc.....	126
t_bind.....	129
t_close.....	133
t_connect.....	135
t_error.....	140
t_free.....	142
t_getinfo.....	144
t_getstate.....	148
t_listen.....	150
t_look.....	153
t_open.....	155
t_optmgmt.....	158
t_rcv.....	163
t_rcvconnect .....	166
t_rcvdis.....	169
t_rcvudata.....	172
t_rcvuderr.....	175

---

t_snd.....	178
t_snddis.....	181
t_sndudata.....	184
t_sync.....	187
t_unbind.....	189
<b>9. Address Manipulation Functions.....</b>	<b>191</b>
getmyclnsnsap—use the local CLNS NSAP.....	192
getmyconsnsap—use the local CONS NSAP.....	193
getnamebynsap—convert an NSAP into a hostname.....	194
getnsapbyname—convert a hostname into an NSAP.....	195
getttselbyname—convert a service name into a TSEL.....	196
nsap2net—write the NSAP to the <i>netbuf</i> structure.....	197
tsap2net—write the TSAP to the <i>netbuf</i> structure.....	198





## *Figures*

---

Figure 1-1	TLI/Stack Architecture.....	2
Figure 2-1	Transport Endpoint.....	9
Figure 2-2	Transport Connection.....	11
Figure 2-3	Listening and Responding Transport Endpoints.....	13
Figure 8-1	Call Order Summary.....	118



## *Tables*

---

Table 1-1	Devices to open . . . . .	3
Table 2-1	Functions for local management phase . . . . .	10
Table 2-2	Functions for establishing transport connections. . . . .	12
Table 2-3	Asynchronous endpoint events. . . . .	14
Table 2-4	Connection-mode data transfer functions . . . . .	15
Table 2-5	Connectionless mode local management functions . . . . .	18
Table 2-6	Connectionless mode data transfer functions. . . . .	18
Table 3-1	Address Library Functions. . . . .	22
Table 5-1	States Describing TLI State Transitions . . . . .	32
Table 5-2	Outgoing Events . . . . .	33
Table 5-3	Incoming Events . . . . .	34
Table 5-4	Initialization and De-initialization . . . . .	35
Table 5-5	Data Transfer in Connectionless Mode . . . . .	36
Table 5-6	Connection/Release/Data Transfer in Connection Mode . . .	36
Table 6-1	SunLink OSI Libraries. . . . .	37
Table 8-1	Call Reference. . . . .	120

---

Table 8-2 .....	155
-----------------	-----

## *Preface*

---

The *SunLink OSI 8.1 TLI Programmer's Reference* explains how to access and use the Transport Library Interface (TLI) application development facilities.

### *Who Should Use This Book*

This book is written for the system programmer developing programs that interface directly with the transport services provided by SunLink OSI 8.1.

You should have a thorough understanding of OSI programming and networking principles, and terminology. Code examples are provided to illustrate the use of the TLI function calls. These samples do not constitute a complete application, although they do work. You can find these samples on the product CD-ROM.

---

## *How This Book Is Organized*

The *SunLink OSI 8.1 TLI Programmer's Reference* is organized as follows:

### ***Part 1 - SunLink OSI 8.1 TLI Guide***

*Chapter 1, "TLI Overview"* introduces the Transport Level Interface, what it does, and how it relates to the SunLink OSI 8.1 architecture. It explains the different modes of connection, gives an overview of the order of functions calls for each mode.

*Chapter 2, "Modes of Service"* describes connection mode and connectionless mode SunLink OSI 8.1 specifics. It also describes the format of the protocol address and how to allocate memory.

*Chapter 3, "Addressing"* describes the correct address formats to use when working with TLI.

*Chapter 4, "Non-Blocking Mode"* describes the special way in which you use the TLI function calls for programs using non-blocking, or asynchronous mode.

*Chapter 5, "State Transition Tables"* describes the states, events and legal sequence of states.

*Chapter 6, "Compiling & Protocol Independence"* lists a set of rules to follow for developing protocol independent programs and explains the requirements for linking and compiling programs.

*Chapter 7, "Programming Examples"* provides some example segments of programs for the various TLI modes. These are supplied on the product CD-ROM in `/opt/SUNWconn/osinet/example/tli`.

### ***Part 2 - Reference***

*Chapter 8, "Using TLI Functions with SunLink OSI"* contains reference pages of the TLI functions, their respective parameters and options, return values and error codes.

*Chapter 9, "Address Manipulation Functions"* is a reference to the address manipulation functions provided as part of the `osi.h` library.

---

## *Related Documentation*

The other documents in the SunLink OSI 8.1 set are:

- *Installing and Licensing SunLink 8.1*—describes the procedure for installing and licensing SunLink OSI 8.1. It also describes the components and modules that make up the SunLink OSI 8.1 package. You should use this book to ensure that the software is installed correctly before proceeding to the configuration procedure explained in the *SunLink OSI 8.1 Communication Platform Administrator's Guide*.
- *SunLink OSI 8.1 Communication Platform Administrator's Guide*—describes how to configure the SunLink OSI 8.1 software for your network.
- *SunLink OSI 8.1 APLI Programmer's Reference*—describes the ACSE Presentation Library Interface (APLI), describing how to develop applications that access the ACSE and presentation layer services directly. It describes its functions and how to compile and link application programs.

---

## *Standards Reference*

The SunLink OSI 8.1 Transport Level Interface conforms to the OSI reference model, and is specifically based on the following standards:

- ISO/IEC-8072 *Transport Service Definition*  
(CCITT X.214)
- ISO/IEC-8073 *Connection-Oriented Transport Protocol Specification*  
(CCITT X.224)
- ISO/IEC-8073 (ADD 2) *Class 4 Operation Over Connectionless Network Service*  
(CCITT X.224)
- OSI/IEC- 8602 *Connectionless Transport Protocol (CLTP)*



---

## Typographic Conventions

The following table describes the conventions for fonts and symbols used in this book.

Table P-1 Typographic Conventions

Typeface or Symbol	Meaning	Example
Typewriter	The names of commands, files, and directories; computer output	Edit your <code>.login</code> file. Use <code>ls -a</code> to list all files. system% You have mail.
<b>boldface</b>	User input; what you type	<div>system% <b>su</b> Password:</div>
<i>italic</i>	Command-line placeholder: replace with an actual name or value	To delete a file, type <code>rm filename</code> .
	Book titles, new words or terms requiring emphasis	See Chapter 6 in the <i>User's Guide</i> . These are called <i>class</i> options. You <i>must</i> be root to do this.
Code samples are placed in boxes and may display the following output:		
%	UNIX C shell prompt	system%
\$	UNIX Bourne and Korn shell prompt	system\$
#	Superuser prompt, all shells	system#



## *Part 1 — SunLink OSI 8.1 TLI Guide*

---



# TLI Overview

---

1 

<i>TLI Architecture</i>	<i>page 2</i>
<i>Modes of Service</i>	<i>page 4</i>
<i>State Transitions</i>	<i>page 6</i>
<i>Error Handling</i>	<i>page 6</i>

To use TLI, you require a thorough knowledge of the OSI architecture and how it is implemented in SunLink OSI 8.1. More details about the architecture of SunLink OSI 8.1 are given in *SunLink OSI 8.1 Communication Platform Administrator's Guide*.

General information about TLI can be found in the online manual pages. Note though that the online manual pages are not specific to the SunLink OSI 8.1 TLI.

## *TLI Architecture*

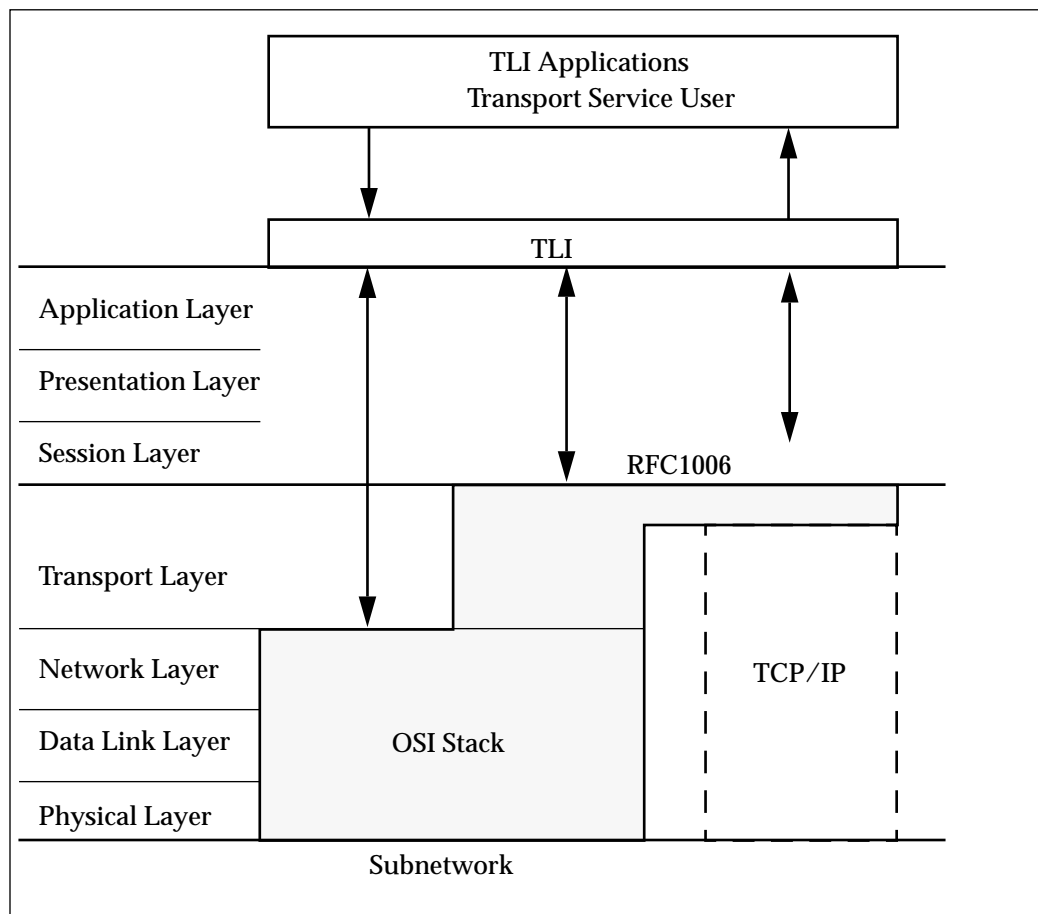


Figure 1-1 TLI/Stack Architecture

As shown in Figure 1-1, the SunLink OSI 8.1 Transport Layer Interface (TLI) can be used in the following situations:

- To develop applications above the transport layer of the OSI stack.
- To develop applications above RFC1006, an extension to the the OSI stack that allows applications to be used above TCP/IP.

- To develop applications over CLNS. This is a means of writing TLI programs over layer 3, which allows for the writing of proprietary transport layers.

SunLink OSI acts as the *transport service provider* for applications using TLI.

It is also possible to write applications that use TLI which are independent of the transport provider (see “Protocol Independency Rules” on page 39 for details).

*Transport service users* are the application programs or session layer protocols which use TLI to access SunLink OSI 8.1 transport services. The transport user issues service requests to the transport provider, which calls functions provided by TLI. These TLI functions are contained in libraries that are installed with the SunLink OSI 8.1 software.

## *Devices to Open*

The device you open depends on what you are interfacing to, as shown in Table 1-1.

*Table 1-1* Devices to open

Interfacing to	Device to open
RFC1006	/dev/otk6
Connection Mode	/dev/otpi
CLNS	/dev/clnp
Connectionless Mode	/dev/oclt

## *Modes of Service*

SunLink OSI implements both the Connection-Oriented Transport Protocol (ISO 8073) and the Connectionless Transport Protocol (ISO 8602) and supports both connection mode and connectionless mode services. The transport protocol classes which can be used in connection mode are TP4 over CLNP and TP 0, 2, 3, and 4 over CONS.

TLI provides two modes of service:

- *Connection mode*
- *Connectionless mode*

### *Connection Mode Service*

Connection mode requires a connection to be established before data may be sent, that is, it is circuit-oriented. The connection should be closed once the data transfer is complete. This mode of service should be used for applications that require datastream-oriented interactions.

Connection mode transport service has four phases of operation:

- *Local management* establishes a channel of communication from the initiator or transport user to the transport provider. This channel of communication is known as a transport endpoint. The transport endpoint is then bound to a TSAP address and identified with a unique file descriptor (*fd*). At this stage you can also determine the characteristics of the transport service, such as the transport class to be used.
- *Connection establishment* between peer transport users. The connection initiator is known as the *client*, and the connection responder is known as the *server*. Since the server listens for incoming calls initiated by a client, the connection establishment phase is treated differently for servers and clients.
- *Data transfer* enables the transfer of data in both directions between the client and server. For connection mode, all data transferred is guaranteed to be delivered.
- *Connection release* directs the transport provider to release the connection. This is known as an abortive release.



## *Connectionless Mode Service*

Connectionless mode transfers data in self-contained units where there is no relationship between the units, that is, it is message-oriented. This service requires peer users to be able to determine the characteristics of the data. All the information required to deliver a message (such as the destination address) is presented to the transport provider, with the data to be transmitted, within the service request. Connectionless transports do not necessarily maintain the message sequence. The reconstruction of a sequence of messages, or the recovery of lost data is the responsibility of the transport user.

Connectionless mode transport service has two phases:

- *Local management* establishes a channel of communication from the initiator or transport user to the transport provider. This channel of communication is known as a transport endpoint. The transport endpoint is then bound to a TSAP address and identified with a unique file descriptor (*fd*).
- *Data transfer* phase lets a user transfer data units (known as datagrams) to the specified peer user. Each data unit is self contained and must be accompanied by the transport address of the destination user.

## *Blocking and Non-Blocking Interactions*

Connection mode and connectionless services can be configured for blocking (synchronous) or non-blocking (asynchronous) mode operation.

- *Blocking mode* returns the results of a function calls when it has completed its operations. While waiting for the results of a call, the user cannot perform other tasks. For example, if a `t_connect` function is called, this function will not return until the connection with the remote system has been set up. This may result in a delay of several seconds during which the transport user program cannot perform any other function.
- *Non-blocking mode* returns as soon as the call is sent to the local transport provider, without any guarantee that the associated operations have been completed. This means that the transport user program can perform other tasks while waiting for an operation to complete. It is the responsibility of the transport user program to poll the library in order to see when the operation is complete.

These two modes are provided through the same interface, with blocking mode being the default mode of execution.

In general, implementing programs using blocking mode is simpler. However, in cases where the transport user program must perform multiple tasks, where more than one connection needs to be managed simultaneously, or where time is an issue, non-blocking mode should be used. The specific use of function calls for non-blocking mode is discussed in more detail in Chapter 4, “Non-Blocking Mode”.

## *State Transitions*

The state transitions describe the possible states of the transport provider as seen by the transport user. They describe the incoming and outgoing events that occur on a connection, and determine the allowable sequence of function calls.

You should understand the state transitions before writing TLI programs. See Chapter 5, “State Transition Tables” for full details of the legal sequence of calls.

## *Error Handling*

There are two levels of errors for TLI:

- *Library level errors* are returned from function calls. A failed function call returns a value of -1 and sets `t_errno` to a value that defines the reason for the failure. This value is not cleared and is only replaced when another error occurs. If a transport error occurs the transport state may change.
- *Operating system service routine level errors* are generated by each function call when an operating system routine fails or a general error occurs. These are specified by setting the TLI-specific error, `t_errno` to `TSYSERR`. This indicates that the `errno` system error has been set to indicate the exact error. The errors described by `errno` are listed in `/usr/include/sys/errno.h`. A protocol error, for example can generate a system error. If the error is severe, it can cause the file descriptor (*fd*) and the transport endpoint to be unusable. The transport endpoint identified by that file descriptor must be closed by all users, after which it can be re-opened and initialized.

## *Modes of Service*

---

2 

<i>Connection Mode Services</i>	<i>page 8</i>
<i>Connectionless Mode</i>	<i>page 17</i>
<i>Memory Management</i>	<i>page 19</i>

This chapter describes more specific information regarding the characteristics of connection mode and connectionless mode. The details of the parameters and options for the functions are provided in Chapter 8, “Using TLI Functions with SunLink OSI.”

## Connection Mode Services

Connection mode services are used for circuit-oriented connections. That is, a connection must be established before the data can be transferred, and it must be released after the data transfer is complete. It takes place in the following four phases:

1. *Local management* to establish a connection, or endpoint, between the transport user and the transport provider. The `t_open` function performs this task, and identifies the transport endpoint with a file descriptor. The `t_bind` function is then used to associate a TSAP address with the endpoint. Protocol options can be negotiated with the `t_optmgmt` function.
2. *Connection establishment* to initiate a connection with a remote system. The initiator, or *client*, requests a connection and the responder or *server* listens for these connection requests. The server accepts or rejects the connection request.
3. *Data transfer* to transfer data over the established connection. Both client and server can send and receive data.
4. *Connection release* to close the connection and disassociate the transport endpoint.

The initiating system (client) and responding system (server) perform different operations during the connection establishment phase. During the other phases, systems at either end of the connection use the same functions.

The transport endpoint can be disconnected and reallocated at any time during the process. Any associated transport connections will be terminated.

These phases are described in detail in the following sections.

## Local Management Phase

The local management phase defines the local interactions which take place between a transport user and a transport provider. A user must establish a unique channel of communication, or *transport endpoint*, with the transport provider. Each transport endpoint is identified by a unique file descriptor. This is illustrated in Figure 2-1.

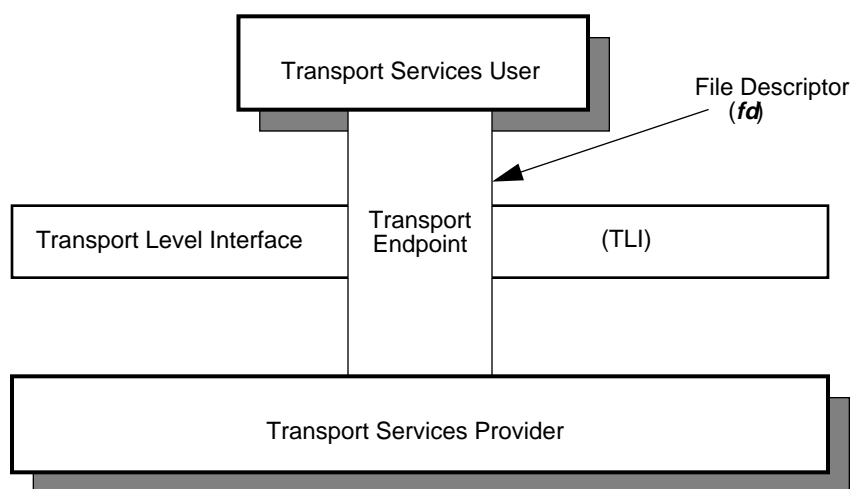


Figure 2-1 Transport Endpoint

Each user must establish its identity with the transport provider. A TSAP address is associated with each transport endpoint and is used in connection requests. See Chapter 3, “Addressing” for a description of how this TSAP is formed. A user process can manage several transport endpoints.

Once the connection endpoint and its identity have been set up, the default characteristics of the SunLink OSI transport service can be modified. For example, the transport user can select a specific transport class or Interface Data Unit (IDU) size.

The specific functions used for the local management phase are shown in the following table:

*Table 2-1* Functions for local management phase

Function	Description
t_open	Establishes a transport endpoint connected to a chosen transport provider. For the SunLink OSI 8.1 connection mode service, specify the /dev/otpi device.
t_bind	Binds a TSAP address to a transport endpoint.
t_optmgt	Negotiates protocol-specific options with the transport provider. It is used to modify the characteristics of the transport service for connection mode.

Other functions can also be used at this stage but are not specific to this phase. For example, t\_getinfo, t\_getstate, t\_sync, t\_alloc, t\_free, t\_error, and t\_look.

## Connection Establishment Phase

The connection establishment phase allows two users to create a connection between them, as illustrated in Figure 2-2.

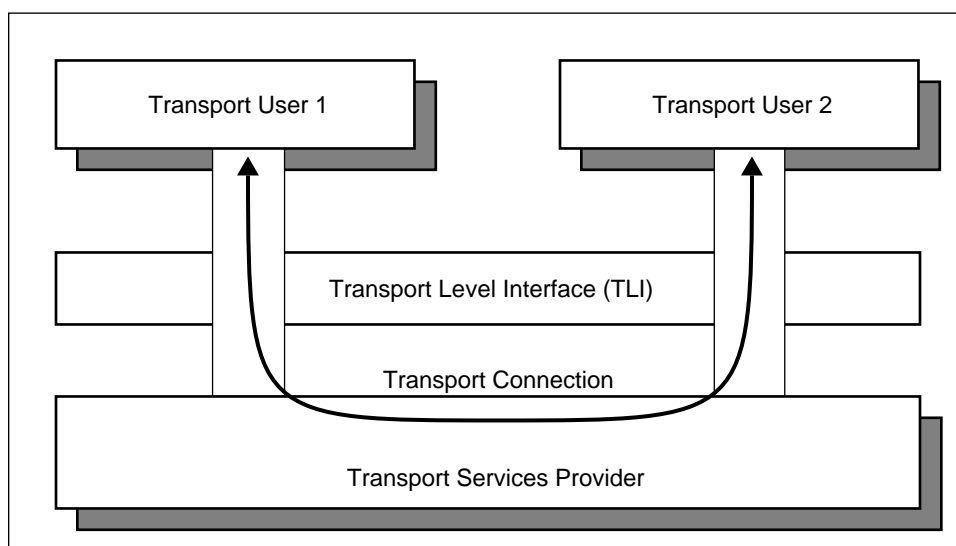


Figure 2-2 Transport Connection

During this phase, transport users are classed as clients (initiators of the connection), or servers (who respond to incoming connections). Client programs simply initiate the connection. Server programs need to listen for incoming connections and decide whether to accept them or not. Therefore, connection establishment is handled differently for each case, and different TLI functions are required for clients and servers.

The following table summarizes the functions used for establishing a transport connection.

*Table 2-2* Functions for establishing transport connections

Function	Description
<code>t_connect</code>	Establishes a connection with the transport user at a specified destination for a client.
<code>t_listen</code>	Listens for connect request from another transport user for a server.
<code>t_accept</code>	Accepts a request for a transport connection for a server.

For connections made in non-blocking mode, `t_rcvconnect` is used to complete the connection establishment in response to `t_connect`.

The connection establishment requirements for the client and server are described below.

## *Client Connection Request*

Once the transport endpoint has been opened and associated with a TSAP address, a client needs to request a connection to a server. The client uses `t_connect` to specify the address of the server with which it is establishing a connection. In blocking mode, the client waits for the server to accept the request, or rejects the connection.

The client can use the `t_connect` function to send user data with the `t_connect` function. The data is set in the user data field on the transport connect request PDU. The amount of data allowed to be sent in this way is specified by the `t_info` structure returned by the `t_open` function. This is protocol-dependent.

## *Server Responsibilities*

The responsibility of the server in the connection establishment phase is to listen at the transport endpoint for incoming connection requests. The `t_listen` function blocks until an incoming connection request is received. The `t_listen` function returns information contained in a `t_call` structure which identifies the client requesting the connection and any protocol options



and user data sent with the connection request. The server accepts the connection request with the `t_accept` function or rejects it with the `t_snddis` function.

Generally, server programs use a single transport endpoint to listen for incoming calls and a different transport endpoint to actually handle the call. The `t_accept` function can accept the connection request and specify a different transport endpoint to handle the call. This can simplify the design of a server which manages multiple connections, and ensures that incoming calls are handled consistently. Once the listening transport endpoint has "handed off" the call to a different transport endpoint, it can continue to listen for further incoming calls. If only one call will ever be handled by the server, the same transport endpoint can be used for both listening and handling the call.

The transport endpoint used to handle the call must be opened using `t_open` and an address bound to it using `t_bind`, before it can be used by the `t_accept` function.

A `t_call` structure is also passed in the `t_accept` call. This contains the calling address, any transport options to be negotiated, any user data that the user wishes to return in the Connect Confirm PDU, and the sequence number of the call. The calling address and the sequence number together allow the transport provider to identify which of the calls in the listen queue is being accepted. These values are those that were returned by the `t_listen` function.

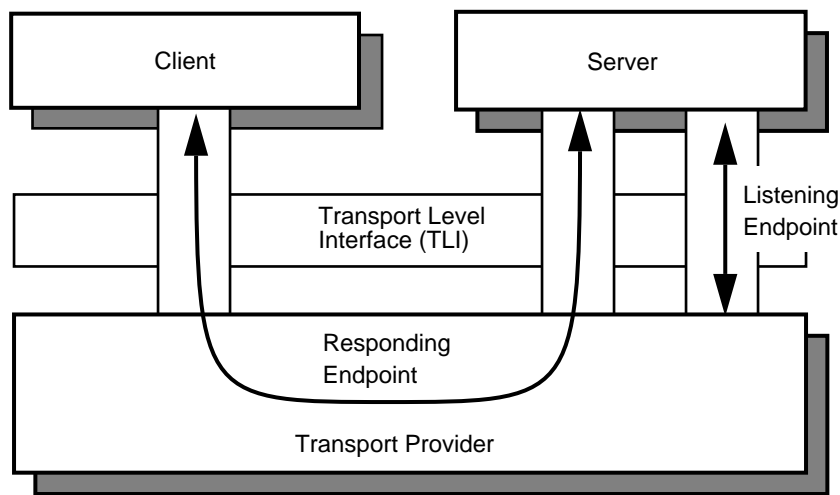


Figure 2-3 Listening and Responding Transport Endpoints

There are two reasons for the failure of a `t_accept` function:

- An local error has been detected by the TLI library or the transport provider. The program exits, closing all file descriptors, and resulting in the deallocation of all resources and the closing of all connections.
- An incoming event has been received on the connection. The only event which is valid in the current state is the reception of a Disconnect Request from the remote system. The global variable `t_errno` will have a value of `TLOOK`, indicating that an asynchronous event has been detected.

The `TLOOK` error has special significance. `TLOOK` is set if a TLI routine is interrupted by an unexpected asynchronous transport event on the endpoint. `TLOOK` does not report an error with a TLI routine, but the normal processing of the routine is not done because of the pending event. The events defined by TLI are listed below:

*Table 2-3 Asynchronous endpoint events*

Name	Description
<code>T_LISTEN</code>	Connection indication arrived at the transport endpoint.
<code>T_CONNECT</code>	Confirmation of a previous connect indication arrived (generated when a server accepts a connect request).
<code>T_DATA</code>	User data indication has arrived.
<code>T_EXDATA</code>	Expedited data indication has arrived.
<code>T_DISCONNECT</code>	Notice of an aborted connection or of a rejected connect request arrived.
<code>T_ORDREL</code>	A request for orderly release of a connection arrived.
<code>T_UDERR</code>	Notice of an error in a previous datagram arrived.

The state tables in Chapter 5, “State Transition Tables” show which events can happen in each state. `t_look` lets a user determine what event has occurred if a `TLOOK` error is returned.

## Data Transfer

Once the connection is established, there is no distinction in the functions used for clients and servers. Data is transferred over the connection using `t_snd` and `t_rcv` functions. Both client and server can send or receive data or close the connection. The functions used for connection mode data transfer are:

Table 2-4 Connection-mode data transfer functions

Function	Description
<code>t_snd</code>	Sends data or expedited data over a connection.
<code>t_rcv</code>	Receives data or expedited data over a connection.

Two types of data can be transferred:

- Normal data
- Expedited data

Expedited data mode is used for urgent data and is indicated by setting the `T_EXPEDITED` data flag in `t_snd` or `t_rcv`. The support of expedited data depends on the transport class being used. Transport class 0 does not support expedited data.

Data transfer in connection mode is in byte streams divided into Transport Service Data Units (TSDU). To indicate that there is more information to transfer in a given TSDU, the `T_MORE` flag is set in each `t_snd` function. This flag indicates that the current `t_snd` and the following one are a logical unit. The final call will not have its `T_MORE` flag set. SunLink OSI 8.1 preserves TSDU message boundaries over the transport connection. Maximum message size is determined by the transport protocol. Use `t_getinfo` to determine the maximum message size allowed.

The `T_MORE` flag implies nothing about how the data is packaged below TLI or how the data is delivered to the remote user. Each transport protocol, and each implementation of a protocol, may package and deliver the data differently.

For example, if a user sends a complete message in a single call to `t_snd`, there is no guarantee that the transport provider delivers the data in a single unit to the receiving user. Similarly, a message transmitted in two units may be delivered in a single unit to the remote transport user. The message boundaries

are only preserved by setting the value of `T_MORE` for `t_snd` and testing it after `t_rcv`. This guarantees that the receiver sees a message with the same contents and message boundaries as was sent.

If too much data is sent at any one time to the transport provider, flow control may be exercised. This depends on the transport class in use. In such conditions the `t_snd` function will block until the flow control restriction is removed.

### *Connection Release*

SunLink OSI 8.1 supports only the abortive release procedure specified by the TLI library. The connection release uses the following functions:

Function	Description
<code>t_snddis</code>	Sends a user initiated disconnect request.
<code>t_rcvdis</code>	Retrieves information about the cause, and any user data when a disconnect request received.

Where the abortive release procedure is used any previously sent data which has not already been received by the remote transport user is discarded by the transport provider. This means that the transport users must ensure that all data has been transferred before disconnecting, for example, by sending a unique pattern to indicate the last of the data. When the client receives this, it can initiate the connection release.

Either user can call `t_snddis` to perform an abortive connection release at any time. The transport provider can abort a connection if a problem occurs below TLI. An abortive release breaks the connection immediately and discards any data that has not been delivered to the destination user.

The ability to send data when aborting a connection using `t_snddis` is supported by SunLink OSI. This corresponds to sending data in the user data field of the Disconnect Request PDU.

When a transport user is notified of the aborted connection, `t_rcvdis` should be called to receive the disconnect request. The function returns a reason code that identifies why the connection was aborted, and returns any data that may

have accompanied the disconnect request (if the abort was initiated by the remote user). The reason code is specific to the underlying transport protocol and should not be interpreted by protocol-independent software.

---

**Note** – The `t_snddis` function can also be used to reject an incoming connection request. In this case the `t_call` structure which is the second argument of this function must be supplied and filled in to allow the connection being rejected to be identified. Otherwise this argument is optional.

---

## *Connectionless Mode*

The connectionless mode service is used for message-oriented connections. It is appropriate for short-term request and response interactions, usually where small amounts of data are being transferred, and where the order of data is not critical. Data is transferred in self-contained units with no logical relationship between the different units.

When using the connectionless mode service there is no concept of client or server at the transport or TLI level, since there is no connection initiator or responder. However, it does require an established association between peer users to determine the characteristics of the transport connection.

Use connectionless mode service for applications that:

- Have short-term request/response interactions
- Do not require sequential delivery of data
- Can take steps to recover the data in any lost messages

Connectionless mode takes place in two phases:

- Local management phase
- Data transfer phase

## Local Management Phase

The initiation of the endpoint is similar to that described for connection mode. `t_open` is used to set up the transport endpoint, but this time it specifies the SunLink OSI 8.1 connectionless mode service. Similarly, `t_bind` is used to bind to the address. The functions used for connectionless mode local management are shown in Table 2-5.

*Table 2-5* Connectionless mode local management functions

Function	Description
<code>t_open</code>	Establishes a transport endpoint connected to a chosen transport provider. For the SunLink OSI 8.1 connectionless mode service, specify the <code>/dev/oclt</code> device.
<code>t_bind</code>	Binds a TSAP address to a transport endpoint.

There are no protocol options to be set for connectionless transport so the `t_optmgmt` function does not need to be used. However, you can change the size of the IDU using the `t_optmgmt` function.

## Data Transfer Phase

The data transfer phase allows a user to transfer data units (known as datagrams) to the specified peer user. Each data unit must be accompanied by the transport address of the destination user. The `t_sndudata` function sends and the `t_rcvudata` function receives messages. Table 2-6 lists the functions required for connectionless mode data transfer.

*Table 2-6* Connectionless mode data transfer functions

Function	Description
<code>t_sndudata</code>	Sends a message to another user of the transport.
<code>t_rcvudata</code>	Receives a message sent by another user of the transport.
<code>t_rcvuderr</code>	Retrieves error information associated with a previously sent message.

Once a user has bound an address to the transport endpoint, datagrams may be sent or received using that endpoint. Each outgoing message carries the address of the destination user. In SunLink OSI connectionless mode you cannot specify protocol options for the transfer of data.

Each datagram used to transfer data using `t_sndudata` and `t_rcvudata` has an associated `t_unitdata` structure containing information about the source address or destination address, protocol options and the data itself. Use `t_alloc` to allocate the structure and buffers that will contain this information. The `T_MORE` flag may be used to indicate that data spread over more than one buffer should be grouped together.

Expedited data is not supported by SunLink OSI for connectionless mode.

## *Error Handling*

If the transport provider cannot process a datagram sent by `t_sndudata`, it returns a unit data error event, `T_UDERR`, to the user. This event includes the destination address and options of the datagram, and a protocol-specific error value that identifies the error. Datagram errors are protocol-specific.

---

**Note** – A unit data error event does not always indicate success or failure in delivering the datagram to the specified destination. Remember, the connectionless mode service does not guarantee reliable delivery of data.

---

## *Memory Management*

The structures containing call information require memory to be allocated before they can be used. The `t_alloc` function allocates memory for specified structures and also for the buffers referenced by the structure, if required. Using `t_alloc` to allocate structures helps to ensure the compatibility of user programs with future releases of the transport interface.

For each field specified, `t_alloc` can allocate memory for the buffer associated with that field. The length of each buffer allocated is based on the same size information that is returned to the user on `t_open` and `t_getinfo`. The file descriptor identifies the transport endpoint through which the newly allocated structure will be passed, so that the appropriate size information can be accessed. If the size value associated with any specified field is -1 or -2, `t_alloc` is unable to determine the size of the buffer to allocate and will fail.

Set the `T_ALL` field to allocate all relevant fields (both buffers and structures) of the given structure.

Use `t_free` to free memory previously allocated by `t_alloc`. This function can free memory for the specified structure, and also memory for buffers referenced by the structure.


Undefined results will occur if `t_free` tries to deallocate a block of memory that was not previously allocated by `t_alloc`.

See `t_alloc` on page 126 and `t_free` on page 142 for full details of the parameters for these functions.



## Addressing

---

3 

<i>Address Library Functions</i>	<i>page 22</i>
<i>Setting Addresses Explicitly</i>	<i>page 23</i>

There are two ways to handle addressing when using SunLink OSI 8.1 TLI. You can either make use of the address library functions supplied, or you can set addresses explicitly.

## Address Library Functions

The User Library provides a mechanism for setting local and remote addresses using the hostname of the machine. This avoids the need for users of your applications to deal with NSAPs and TSAPs. The library uses the file `/etc/net/oclt/host` for hostname to NSAP conversion, and the file `/etc/net/oclt/services` for hostname to TSEL conversion. In order to use it, you must:

- Add the directory `/opt/SUNWconn/osinet/lib` to your `LD_LIBRARY_PATH` environment variable.
- Compile setting the option `-I/opt/SUNWconn/osinet/include`.

The functions used are listed in Table 3-1.

Table 3-1 Address Library Functions

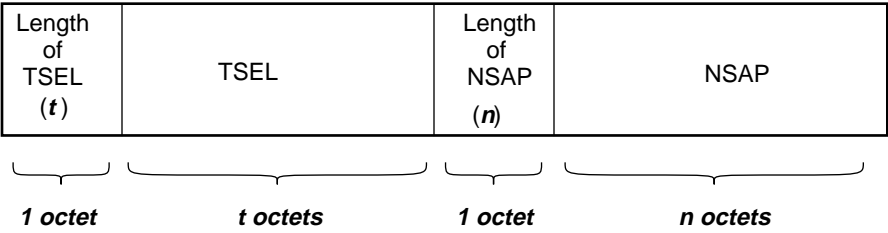
Function	Description
<code>getnsapbyname</code>	Retrieves the NSAP from the host file.
<code>getnamebynsap</code>	Retrieves the hostname from the host file.
<code>igettselbyn</code>	Retrieves the TSEL from the service file.
<code>getmyclnsnsap</code>	Returns the local CLNS NSAP.
<code>getmyconsnsap</code>	Returns the local CONS NSAP.
<code>tsap2net</code>	Sets the <i>net</i> variable as a formatted buffer composed of the NSAP and the TSEL. This can be used directly by the TLI transport provider.
<code>nsap2net</code>	Sets the <i>net</i> variable as a formatted buffer composed of the NSAP. This can be used directly by the TLI network provider.

# Setting Addresses Explicitly

Addressing varies according to whether you are using TLI above the transport layer of the OSI stack, above RFC1006 or above CLNS.

## Transport Layer Addressing

The TSAP, which identifies the transport provider can be coded in hexadecimal or as a character string. A TSAP is formed from the TSEL plus the NSAP.



The lengths of the TSEL and NSAP are described by a 1 octet field.  
For example, a TSAP described in hexadecimal might look like this:

	Length	TSEL	Length	NSAP
Specified	0x3	0x01 0x03 0x05	0x5	0x4712345678
Transferred	03	010305	05	4712345678

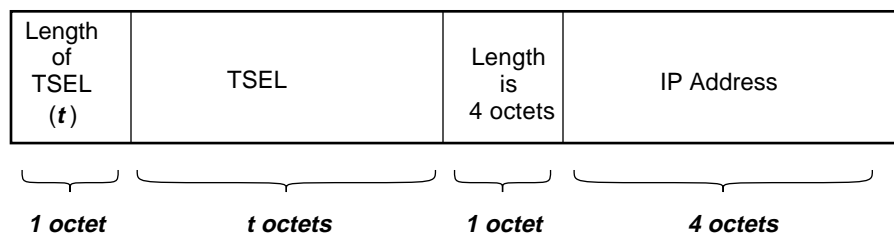
An example of a TSAP described by a character string might look like this:

	Length	TSEL	Length	NSAP
Specified	0x4	"TSEL"	0x5	"NSAP"
Transferred	04	5453454C	04	4E534150

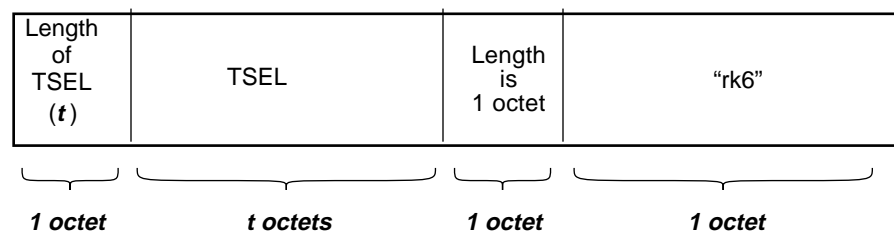
## RFC1006 Addressing

Addressing for RFC1006 applications is formatted in the same way as that for Transport Layer applications, described in "Transport Layer Addressing."

When using the `t_connect` function, the NSAP is replaced by a 4 octet IP address, as shown below:

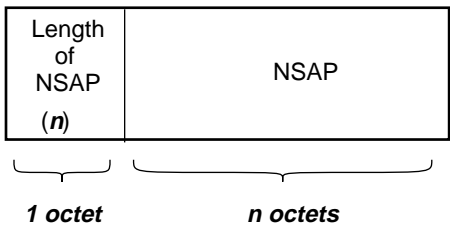


When using the `t_bind` function, the NSAP takes the value `rk6`, as shown below:



CLNS Addressing

If your application is to run over the network layer (CLNS), the address to bind to is any valid NSAP. Specify the NSAP in the `netbuf.buf` field in either hexadecimal format or as a character string. Specify the length of the address in octets in the `netbuf.len` field. You can use multiple NSAPs. You cannot use a NULL address.



The length of the NSAP is described by a 1 octet field.  
An NSAP described in hexadecimal might look like this:

	Length	NSAP
Specified	0x5	0x4712345678
Transferred	05	4712345678

An NSAP described by a character string might look like this

	Length	NSAP
Specified	0x5	"NSAP"
Transferred	04	4E534150



## Non-Blocking Mode

---



<i>Setting Non-Blocking Mode</i>	<i>page 28</i>
<i>Non-Blocking Functions</i>	<i>page 28</i>
<i>Using poll and select</i>	<i>page 30</i>

Many TLI library calls block to wait for an incoming event. In many practical situations this synchronous mode of operation is not acceptable. Where a process is responsible for managing more than one connection, or where time is important, it may not be possible to allow an application to be driven by incoming events. Non-blocking mode allows an application to do local processing while waiting for an asynchronous TLI event, or to manage a number of connections simultaneously.

Since the default mode of operation is blocking mode, you need to select non-blocking mode with the `O_NONBLOCK` flag. This can be set when initiating a transport endpoint, or afterwards, with the `fcntl` operating system service routine. The selection of non-blocking mode is specific to a connection endpoint.

Asynchronous processing of TLI events is available to applications through the combination of asynchronous features and the non-blocking mode of TLI library calls. The `poll` system call and the `I_SETSIG` `ioctl` command can be used to process events asynchronously.

TLI routines that blocks for events can also be run in non-blocking mode. For example, `t_listen` normally blocks for a connect request, but server can periodically poll a transport endpoint for queued connect requests by calling `t_listen` in non-blocking mode.

## Setting Non-Blocking Mode

Non-blocking mode is set in one of the following ways:

- When initiating a connection, by setting the `t_open` *oflag* argument as `O_NDELAY` or `O_NONBLOCK`.
- During a connection using the `fcntl` operating system call. Whenever the `F_SETFL` option is used with the `O_NDELAY` or `O_NONBLOCK` flag, the transport endpoint is set to non-blocking mode. The `fcntl` system call may also be used at any time to reset synchronous mode operation for the endpoint.

The `fcntl` system call is a general system call which can used to change the attributes of any file descriptor. You can find more details of its use in the online manual pages.

## Non-Blocking Functions

Some SunLink OSI 8.1 function calls act differently in blocking and non-blocking mode. It is important that you understand the semantics of `O_NDELAY` or `O_NONBLOCK` for a particular routine before you use it. The functions that operate differently for non-blocking mode are:

- `t_open`
- `t_connect`
- `t_rcvconnect`
- `t_listen`
- `t_rcv`
- `t_rcvudata`
- `t_snd`
- `t_sndudata`

The other TLI functions operate similarly for both modes.



## `t_open`

Set the `O_NDELAY` or `O_NONBLOCK` flag with *oflag* in `t_open` to initiate a connection in non-blocking mode.

## `t_connect` *and* `t_rcvconnect`

In blocking mode, when a client program calls the `t_connect` function, the function blocks until a response is received from the remote system. For OSI transport this corresponds to an exchange of CR (Connect Request) and CC (Connect Confirm) TPDU. The time it takes to set up the connection depends on the speed of the network and whether or not the CR TPDU needs to be retransmitted.

If setting up the connection might take too long, you can initiate the connection process in non-blocking mode. This corresponds to the transmission of the CR TPDU. If no response is received immediately the function returns indicating an error with the value of `t_error` set to `TNODATA`. The transport user program must then poll the file descriptor to see when the response is received.

The `t_rcvconnect` call is used to query the TLI library for the response to a connection request. If the transport endpoint is reset into blocking mode before this function is called, `t_rcvconnect` blocks until the connect confirm is received. If the transport endpoint is in non-blocking mode, the `t_rcvconnect` call indicates an error with `t_errno` set to `TNODATA` if the connect confirm has not been received.

## `t_listen`

In blocking mode, `t_listen` blocks until an incoming connection indication is received. If the transport endpoint is set to non-blocking mode, `t_listen` polls for incoming connections. If there are no incoming connections in the queue, `t_listen` returns an error with the value of `t_errno` set to `TNODATA`.

It is important that `t_listen` is called sufficiently often to ensure that the connection is not terminated by the remote initiating system due to the lack of response or that the queue size is exceeded. The amount of time allowed depends on the Retransmission Time and Retransmission Count on the remote system.

## `t_rcv` *and* `t_rcvudata`

In synchronous mode, `t_rcv` blocks until there is an incoming event from the transport provider. In non-blocking mode, `t_rcv` and `t_rcvudata` poll for incoming events. If there are no incoming events or data, then they return an error with the value of `t_errno` set to `TNODATA`.

## `t_snd` *and* `t_sndudata`

Normally, `t_snd` does not block even in blocking mode. Data is accepted by the transport provider and will be queued internally or sent directly to the remote system. The function returns almost immediately on completion of either operation. However, if flow control is invoked, and the internal queues are full, the `t_snd` function call blocks until the flow control condition is removed.

In non-blocking mode, the `t_snd` call returns immediately, and the following cases are possible:

- If the `T_MORE` flag is set by the `t_snd` call and the number of bytes returned by `t_snd` is less than the number requested in the argument list for `t_snd`, then part of the data has been accepted by the transport provider, and the remainder must be sent in another `t_snd` call.
- If the `t_snd` call returns an error and `t_errno` is set to `TFLOW`, the data was not accepted by the transport provider due to flow control being in operation.

If internal memory or STREAMS resources are exhausted, the `t_snd` call blocks whether or not non-blocking mode is set.

The `t_sndudata` call operates in the same way as `t_snd`. However, if there is flow control, the `t_sndudata` call fails.

## *Using* `poll` *and* `select`

The `poll` and `select` system calls may be used to detect activity on transport endpoint file descriptors in the same way as for any other file descriptors. These calls will often be used whenever multiple connections are being managed by a single process and non-blocking mode is being used.

Check the online manual pages for details of how to use these calls.

## State Transition Tables

5 

<i>TLI States</i>	<i>page 32</i>
<i>Outgoing Events</i>	<i>page 32</i>
<i>Incoming Events</i>	<i>page 34</i>
<i>Transport User Actions</i>	<i>page 34</i>
<i>State Tables</i>	<i>page 35</i>

This chapter describes:

- The possible states of the transport provider as seen by the transport user describing the incoming and outgoing events that occur on a connection.
- The legal sequence of state transitions, given the current state and event.
- Any actions that must be taken by the transport user.

Functions that are not included in the state tables do not affect the state of the interface.

## TLI States

Table 5-1 defines the states that describe the TLI state transitions. SunLink OSI 8.1 supports T\_COTS (connection mode) and T\_CLTS (connectionless mode). The T\_COTS\_ORD service type for connection mode orderly release is not supported by SunLink OSI 8.1.

Table 5-1 States Describing TLI State Transitions

State	Description	Service Type
T_UNIT	Uninitialized – initial and final state of interface	T_COTS T_CLTS
T_UNBND	Initialized but not bound	T_COTS T_CLTS
T_IDLE	No connection established	T_COTS T_CLTS
T_OUTCON	Outgoing connection pending for client	T_COTS
T_INCON	Incoming connection pending for server	T_COTS
T_DATAXFER	Data transfer	T_COTS

## Outgoing Events

The outgoing events described in Table 5-2 indicate the return of the specified transport functions that cause a change of state when a request or response is sent to the transport provider. Some events (such as `accept`) are distinguished by the context in which they occur.

The context is based on the values of one or more of the following variables:

- *ocnt* – count of outstanding connect requests
- *fd* – file descriptor of the current transport endpoint
- *resfd* – file descriptor of the transport endpoint where a connection is accepted

The event `sndrel` does not occur when SunLink OSI is the transport provider.

Table 5-2 Outgoing Events

Event	Description	Service Type
opened	Successful return of <code>t_open</code>	T_COTS T_CLTS
bind	Successful return of <code>t_bind</code>	T_COTS T_CLTS
optmgmt	Successful return of <code>t_optmgmt</code>	T_COTS T_CLTS
unbind	Successful return of <code>t_unbind</code>	T_COTS T_CLTS
closed	Successful return of <code>t_close</code>	T_COTS T_CLTS
connect1	Successful return of <code>t_connect</code> in synchronous mode	T_COTS
connect2	TNODATA error on <code>t_connect</code> in asynchronous mode, or TLOOK error due to a disconnect request arriving on the transport endpoint	T_COTS
accept1	Successful return of <code>t_accept</code> with <code>ocnt == 1, fd == resfd</code>	T_COTS
accept2	Successful return of <code>t_accept</code> with <code>ocnt == 1, fd != resfd</code>	T_COTS
accept3	Successful return of <code>t_accept</code> with <code>ocnt &gt; 1</code>	T_COTS
snd	Successful return of <code>t_snd</code>	T_COTS
snddis1	Successful return of <code>t_snddis</code> with <code>ocnt &lt;= 1</code>	T_COTS
snddis2	Successful return of <code>t_snddis</code> with <code>ocnt &gt; 1</code>	T_COTS
sndudata	Successful return of <code>t_sndudata</code>	T_CLTS

Note that *ocnt* is only meaningful for the listening transport endpoint, *fd*.

## Incoming Events

The incoming events described in Table 5-3 correspond to the successful return of the specified transport functions causing a change of state when the functions retrieve data or event information from the transport provider. Some events (such as `rcvdis`) are distinguished by the value of the count of outstanding connect requests on the endpoint (`ocnt`).

The `pass_conn` event is not directly associated with the return of a function. It occurs when a connection is transferred to another endpoint. The event occurs on the endpoint that is being passed the connection, although no TLI routine is called on the endpoint.

Table 5-3 Incoming Events

Event	Description	Service Type
<code>listen</code>	Successful return of <code>t_listen</code>	T_COTS
<code>rcvconnect</code>	Successful return of <code>t_rcvconnect</code>	T_COTS
<code>rcv</code>	Successful return of <code>t_rcv</code>	T_COTS
<code>rcvdis1</code>	Successful return of <code>t_rcvdis</code> , with <code>ocnt</code> $\leq$ 0	T_COTS
<code>rcvdis2</code>	Successful return of <code>t_rcvdis</code> , with <code>ocnt</code> $==$ 1	T_COTS
<code>rcvdis3</code>	Successful return of <code>t_rcvdis</code> with <code>ocnt</code> $>$ 1	T_COTS
<code>rcvudata</code>	Successful return of <code>t_rcvudata</code>	T_CLTS
<code>rcvuderr</code>	Successful return of <code>t_rcvuderr</code>	T_CLTS
<code>pass_conn</code>	Receive a passed connection	T_COTS

## Transport User Actions

The state transitions listed specify the actions that the transport user must take. These actions are represented in the tables by a number from 1 through 4 where:

1. Set the count of outstanding connect requests to zero.
2. Increment the count of outstanding connect requests.
3. Decrement the count of outstanding connect requests.
4. Pass a connection to another transport endpoint as indicated in `t_accept`.

## State Tables

The following tables describe the possible next state according to the current state (shown in the column header) and the current event (shown by the row label). An empty cell indicates an invalid state/event combination. The user actions that must be performed are shown in brackets.

The following should be noted when using the state tables:

- `t_close` can be called from any state to close an endpoint. A transport address bound to an endpoint is unbound. If the transport connection is active, the connection is aborted.
- If a transport user calls a function out of sequence, the function fails and `t_errno` is set to `TOUTSTATE`. The state does not change.
- The error codes `TLOOK` or `TNODATA` after `t_connect` can result in state changes described in “Connection Establishment Phase” on page 11. The state tables assume the correct use of TLI. Some error codes do not cause a change of state.
- The support functions `t_getinfo`, `t_getstate`, `t_alloc`, `t_free`, `t_sync`, `t_look`, and `t_error` are not listed since they do not affect the state.

The following tables illustrate:

- Initialization/de-initialization
- Data transfer in connectionless mode
- Connection/release/data transfer in connection mode

Table 5-4 Initialization and De-initialization

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

**Table 5-5** Data Transfer in Connectionless Mode

Event/State	T_IDLE
snudata	T_IDLE
rcvdata	T_IDLE
rcvuderr	T_IDLE

**Table 5-6** Connection/Release/Data Transfer in Connection Mode

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



## Compiling & Protocol Independence

6 

<i>Compiling and Linking Programs</i>	<i>page 37</i>
<i>Protocol Independency Rules</i>	<i>page 39</i>

This chapter contains the information you need on compiling and linking, together with some tips to ensure your programs are portable.

### Compiling and Linking Programs

The TLI calls must be linked with the libraries `libnsl.a` and `libnsl.so.1` to support library-specific functions. After installation, the libraries are located in `/usr/lib`. The contents of these libraries are described in the table below:

*Table 6-1* SunLink OSI Libraries

Library	Description
<code>libnsl.a</code>	Provides the services defined by Sun Microsystems and UNIX System Laboratories API.
<code>libnsl.so.1</code>	Provides the package description of the data structures.

## *Include Files*

The TLI application source files are standard files defined by the specification. The include files are as follows:

`/opt/SUNWconn/osinet/include/osi_lib.h` delivered with the SunLink OSI 8.1 software.

`/usr/include/sys/tiuser.h` delivered with the operating system software.

These files provide all the standard constants, macros, and data types specified by the TLI specifications.

## *Procedure for Compiling and Linking*

To compile and link the include files, you must specify the following entries in your Makefile:

```
CFLAGS += -I /opt/SUNWconn/osinet/include
```

```
LDFLAGS += -lnsl
```

## *Protocol Independency Rules*

The set of services provided by TLI can be used by various transport providers, if an application does not use protocol-dependent techniques. The following list of rules advises how you can ensure protocol independence:

- In connection mode service, Transport Service Data Units (TSDU) may not be supported by all transport providers. Make no assumptions about preserving logical data boundaries across a connection.
- Protocol and implementation-specific service limits are returned by the `t_open` and `t_getinfo` calls. Use these limits to allocate buffers to store protocol-specific transport addresses and options.
- Do not send user data with connect requests or disconnect requests (see `t_connect` and `t_snddis`). Not all transport protocols support it.
- The buffers in the `t_call` structure used for `t_listen` must be large enough to hold any data sent by the client during connection establishment. Use the `T_ALL` argument in `t_alloc` to set maximum buffer sizes to store the address, options, and user data for the current transport provider.
- Do not test or change options of any TLI routine. These options are specific to the underlying transport protocol.
- Do not pass options with `t_connect` or `t_sndudata`, since the transport provider can use default values. Servers should use the options returned by `t_listen` to accept a connection.
- Do not specify a protocol address on `t_bind`. Let the transport provider assign an appropriate address to the transport endpoint. A server should retrieve its address for `t_bind` in such a way that it does not need to know the format of the transport provider's name space. For SunLink OSI 8.1, you must specifically assign an address, however, to maintain protocol independence, you can specify `t_bind` with a null address. In this case, SunLink OSI 8.1 automatically assigns a default address for X.25 over CONS.
- Do not make assumptions about formats of transport addresses. Transport addresses should not be constants in a program.
- The reason codes associated with `t_rcvdis` are protocol-dependent. Do not interpret this information.

- The `t_rcvuderr` error codes are protocol-dependent. Do not interpret this information.
- Do not code the names of devices into programs, since the device node identifies a particular transport provider, and is not protocol independent.
- Do not use the orderly release facility for connection mode provided by `t_sndrel` and `t_rcvrel`. This facility is not supported by all connection-based transport protocols.

## Programming Examples

---



<i>Argument List Example</i>	<i>page 44</i>
<i>Address Mapping Example</i>	<i>page 47</i>
<i>Connection Mode Client Example</i>	<i>page 50</i>
<i>Connection Mode Server Example</i>	<i>page 56</i>
<i>Connectionless Mode Client Example</i>	<i>page 66</i>
<i>Connectionless Mode Server Example</i>	<i>page 71</i>
<i>Non-Blocking Mode Client Example</i>	<i>page 77</i>
<i>OSI Library Use Examples</i>	<i>page 84</i>
<i>TLI over CLNS Example</i>	<i>page 89</i>
<i>TLI over RFC1006 Examples</i>	<i>page 94</i>
<i>Makefile Example</i>	<i>page 112</i>

This chapter contains the following programming examples:

- `targs.c` obtains the argument list and copies the relevant local and remote addresses and the name of the file to the appropriate strings. See “Argument List Example” on page 44 for the sample program.
- `taddr.c` maps the TSAP address to the TLI buffer format for the transport address. See “Address Mapping Example” on page 47 for the example program.

- `tclient.c` describes an example of a client in connection mode using blocking mode to set up a connection to a server and waits for a file transfer. When the client is notified that all the data is transferred, the client disconnects. See “Connection Mode Client Example” on page 50 for the example program.
- `tserver.c` describes a server in connection mode using blocking mode to listen for incoming connections. When an incoming connection request is received, a new endpoint is created to accept the connection. The process forks to handle the transfer of data and release the connection while the parent process continues to wait for other connection requests. See “Connection Mode Server Example” on page 56 for the example program.
- `tclient_cltp.c` describes a client in connectionless mode using blocking mode to send a packet to a server containing data. It then waits for the server to respond and displays the response. It repeats this in a continuous loop, sending unitdata requests every five seconds. See “Connectionless Mode Client Example” on page 66 for the example program.
- `tserver_cltp.c` describes the server in connectionless mode using blocking mode to respond to a client. It acts as a transaction server. When it receives a unitdata request, it prints it and responds to the client with a text message. Both the received and the sent messages fit into one buffer, so the `T_MORE` flag is not required. See “Connectionless Mode Server Example” on page 71 for the example program.
- `tclient_async.c` describes a client program using non-blocking mode. It sets up a connection with a server and waits for the server to transfer a file. When the server indicates that all the data is transferred, the client releases the connection. See “Non-Blocking Mode Client Example” on page 77 for the example program.
- `uselib.c` and `tbind.c` show how to use the OSI library with TLI.
- `bind_clns.c` is an example of the use of TLI over CLNS. The main emphasis is on the address format set in the `netbuf` structure. It shows binding the address, stopping to allow for checking the routing table and then unbinding.
- `rfcclient.c` is an example of the use of TLI over RFC1006. The client program opens in non-blocking mode, waits for data to be transferred, then disconnects.

- `rfcserver.c` is also an example of the use of TLI over RFC1006. The server program listens for incoming calls and transfers data to the client.
- `Makefile` is an example `makefile` which could be used with these example programs. See “Makefile Example” on page 112 for the example program.

Note that the code samples described in this chapter are only examples and do not constitute a complete application, although they do compile and run. They are provided to illustrate the way in which the function calls are used in the SunLink OSI 8.1 TLI.

The code examples can be found on the product CD-ROM in:

`/opt/SUNWconn/osinet/example`

## Argument List Example

This sample of code can be found on the product CD-ROM, in:  
 /opt/SUNWconn/osinet/example/tli/targs.c.

```

/*
 * Copyright 1995 Sun Microsystems, Inc. All Rights Reserved
 */
/*****
/* This code is provided only for example purposes only and does not      */
/*   comprise part of the supported product.                               */
*****/

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

#include "osi_lib.h"

/* get_addrs() - interpret the argument list of one of the example programs.
 *
 * This routine accepts the argument list from one of the example products
 * and copies the relevant local and remote addresses and the name of the
 * file to be transferred into the relevant strings.
 *
 * Input -
 *   argc- number of arguments on command line.
 *   argv- list of arguments.
 *
 * Output -
 *   client_nsap- NSAP address of client.  If a NULL argument
 *               is supplied this is not expected on
 *               command line.
 *   client_tsel- TSEL of client.  If a NULL argument is
 *               supplied this is not expected on
 *               command line.
 *   server_nsap- NSAP address of server.
 *   server_tsel- TSEL of server.
 *   filename- name of file to be transferred.  If a NULL
 *             argument is supplied this is not
 *             expected on command line.
 */

```



```
void
get_addrs(
    int argc,
    char **argv,
    char *client_nsap,
    char *client_tsel,
    char *server_nsap,
    char *server_tsel,
    char *filename
)
{
    int argn;

    if (argc < 3 || (client_nsap && argc < 5)) {
        Usage();
        exit(0);
    }

    *argv++;
    argn = 1;

    if (client_nsap) {
        strcpy(client_nsap, *argv++);
        argn++;
    }
    if (client_tsel) {
        strcpy(client_tsel, *argv++);
        argn++;
    }
    strcpy(server_nsap, *argv++);
    strcpy(server_tsel, *argv++);
    argn += 2;

    if (!filename)
        return;
    if (argc > argn && strncmp(*argv++, "-f", 2)) {
        Usage();
        exit(0);
    }
    else if (argc <= argn) {
        return;
    }
}
```

```
strcpy(filename, *argv);  
return;  
}
```

## Address Mapping Example

This sample of code can be found on the product CD-ROM, in:  
 /opt/SUNWconn/osinet/example/tli/taddr.c.

```

/*
 * Copyright 1995 Sun Microsystems, Inc. All Rights Reserved
 */
/*****
/*      This code is provided only for example purposes only and does not      */
/*      comprise part of the supported product.                                */
*****/

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

#include "osi_lib.h"

/* tsap_to_net() - TSAP Mapping to TLI buffer format
 *
 * Format = [Length of TSEL][TSEL][Length of NSAP][NSAP]
 *
 * The function returns the length of the TSAP, or -1 in the error condition.
 *
 * Input -
 * nsap - contains a string representation of an NSAP
 *        (eg: 0x4712345678). Hex NSAPs are prefixed
 *        by 0x, character NSAPs are not prefixed.
 * tsel - contains a string representation of a TSEL
 *        (eg: 0x1234). Hex TSELS are prefixed by
 *        0x, character TSELS are not prefixed.
 * Output -
 * net - contains the encoded TSAP in netbuf format.
 *
 */

#define CHAR2INT(n)((n > 0x39) ? (n - 'a' + 0xa) : (n - 0x30))
#define IS_ODD(n)(((n/2)*2 != n) ? (-1) : (0))

int

```

```

tsap_to_net(
    unsigned char*net,
    unsigned char*nsap,
    unsigned char*tsel
)

{
    int i, j, nstart;

    /*
     * Encode the TSEL.
     * If string begins with 0x assume that address is in hexadecimal.
     * Otherwise assume it is in characters.
     */

    if (tsel[0] == '0' && tsel[1] == 'x') {
        /*
         * Address is in hexadecimal - one byte of string is only a
         * half a byte in the address. Length must be an even number.
         */

        net[0] = strlen(tsel) - 2;
        if (IS_ODD(net[0])) {
            return(-1);
        }
        net[0] = net[0]/2;
        for (i = 1, j = 2; i <= (int) net[0]; i++, j += 2) {
            net[i] = CHAR2INT((int)tsel[j]) << 4;
            net[i] |= CHAR2INT((int)tsel[j+1]) & 0x0f;
        }

    }
    else {
        /*
         * Address is in characters.
         */
        net[0] = strlen(tsel);
        strcpy(&net[1], tsel);
    }

    /*
     * Encode the NSAP.
     */

```

```
nstart = net[0] + 1;
if (nsap[0] == '0' && nsap[1] == 'x') {
    /*
     * Address is in hexadecimal - one byte of string is only a
     * half a byte in the address. Length must be an even number.
     */

    net[nstart] = strlen(nsap) - 2;
    if (IS_ODD(net[nstart])) {
        return(-1);
    }
    net[nstart] = net[nstart]/2;
    for (i = nstart + 1, j = 2; i <=(int) (net[nstart] + nstart);
        i++, j += 2) {
        net[i] = CHAR2INT((int)nsap[j]) << 4;
        net[i] |= CHAR2INT((int)nsap[j+1]) & 0x0f;
    }

}
else {
    /*
     * Address is in characters.
     */
    net[nstart] = strlen(nsap);
    strcpy(&net[nstart+1], nsap);
}

return(net[0] + net[nstart] + 2);
}
```

## Connection Mode Client Example

This sample of code can be found on the product CD-ROM, in:  
 /opt/SUNWconn/osinet/example/tli/tclient.c.

```

/*
 * Copyright 1995 Sun Microsystems, Inc. All Rights Reserved
 */
/*****
/*      This code is provided for example purposes only and does not      */
/*      comprise part of the supported product.                          */
*****/

/*****
/* tclient      */
/*      */
/* This is a client program using blocking mode. The client sets up a      */
/* connection with a suitable server program and waits for the server to  */
/* transfer a file. The contents of this file are displayed on stdout.     */
/* Special flags are used by the server to indicate the end of the transfer. */
/* When these flags are received, the client disconnects.                  */
/*      */
/* This program interoperates with the tserver program.                    */
/*      */
*****/

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

#include "osi_lib.h"

#define DLEN 1024
#define ESC 0x1b
#define EOT 0x04

void
Usage()

{
    printf("Usage: tclient <local_nsap> <local_tsel> <remote_nsap> <remote_tsel>\n");
}

```

```

        return;
    }

main(
    int argc,
    char **argv
)
{
    int fd;
    int nbytes, flags;
    chardata_buf[DLEN];
    char      client_nsap[32], server_nsap[32];
    char      client_tsel[32], server_tsel[32];
    struct t_bind*bind, *bound;
    struct t_call*call;
    struct t_discon *discon;

    /*
     * Parse argument list for addresses, etc.
     */
    get_addrs(argc, argv, client_nsap, client_tsel, server_nsap,
              server_tsel, NULL);

    /*
     * Create the transport endpoint.
     */

    if ((fd = t_open("/dev/otpi", O_RDWR, (struct t_info *) NULL)) == -1) {
        t_error("t_open failed");
        exit(1);
    }

    /*
     * Allocate the t_bind structure to contain the local address.
     */

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

```

```

/*
 * Allocate a t_bind structure to contain the bound address.
 */

if ((bound = (struct t_bind *) t_alloc(fd, T_BIND, T_ALL)) == NULL) {
    t_error("t_alloc of t_bind (bound) structure failed");
    exit(3);
}

/*
 * Map the local address to the netbuf format.
 */

if ((bind->addr.len = tsap_to_net(bind->addr.buf, client_nsap,
                                client_tsel)) < 0) {
    fprintf(stderr, "Error mapping client address\n");
    exit(4);
}

/*
 * Bind to the local address.
 */

bind->qlen = 0;
if (t_bind(fd, bind, bound) < 0) {
    t_error("t_bind failed");
    exit(5);
}

/*
 * Allocate a t_call structure to contain the call details, only
 * allocating a buffer for the address.
 */

if ((call = (struct t_call *) t_alloc(fd, T_CALL, T_ADDR)) == NULL) {
    t_error("t_alloc of t_call structure failed");
    exit(6);
}

/*
 * Map the remote address to the netbuf format.
 */

```



```

    if ((call->addr.len = tsap_to_net(call->addr.buf, server_nsap,
                                     server_tsel)) < 0) {
        fprintf(stderr, "Error mapping server address\n");
        exit(7);
    }

    /*
     * Attempt to set up connection with the server.
     */

    call->opt.len = 0;
    call->udata.len = 0;
    call->sequence = 0;
    if (t_connect(fd, call, (struct t_call *) NULL) == -1 ) {
        t_error("t_connect failed");
        exit(8);
    }

    /*
     * Allocate a t_discon structure in case we get a disconnect.
     */

    if ((discon = (struct t_discon *) t_alloc(fd, T_DIS, T_ALL)) == NULL) {
        t_error("t_alloc of t_discon structure failed");
        exit(9);
    }

    /*
     * Receive data from server.
     */

    while (1) {
        if ((nbytes = t_rcv(fd, data_buf, DLEN, &flags)) < 0) {

            switch (t_errno) {

                case TNODATA:

                    /*
                     * Shouldn't happen in synchronous case
                     */
                    continue;

                case TLOOK:

```

```

        switch (t_look(fd)) {

        case T_DATA:
            continue;
        case T_DISCONNECT:
            if (t_rcvdis(fd, discon) < 0) {
                t_error("t_rcvdis failed");
                exit(10);
            }
            exit(0);
        case T_EXDATA:
            fprintf(stderr,
                "Not supporting expedited data\n");
        case T_LISTEN:
        case T_CONNECT:
        case T_UDERR:
        case T_ORDREL:
        default:
            fprintf(stderr, "Protocol error\n");
            exit(11);
        }

    default:
        t_error("t_rcv failed");
        exit(12);
    }
}

/*
 * Check for end of data.
 */

if (data_buf[0] != ESC) {

    /*
     * Write out data.
     */

    if (fwrite(data_buf, 1, nbytes, stdout) < (size_t) 0) {
        fprintf(stderr, "fwrite failed\n");
        exit(13);
    }
}

```

```
else {
    switch (data_buf[1]) {
    case EOT:

        /*
         * End of transfer - send disconnect.
         */

        if (t_snddis(fd, NULL) < 0) {
            t_error("t_snddis failed");
        }
        t_close(fd);
        exit(0);

    default:

        /*
         * ESC only indicated "end of record".
         */

        if (fwrite(&data_buf[1], 1, nbytes-1, stdout)
            < (size_t) 0) {
            fprintf(stderr, "fwrite failed\n");
            exit(14);
        }
        break;
    }
}
}
```

## Connection Mode Server Example

This sample of code can be found on the product CD-ROM, in:  
/opt/SUNWconn/osinet/example/tli/tserver.c.

```

/*
 * Copyright 1995 Sun Microsystems, Inc. All Rights Reserved
 */
/*****
/*      This code is provided for example purposes only and does not      */
/*      comprise part of the supported product.                          */
*****/

/*****
/* tserver                                                                */
/*                                                                */
/* This is an example of a server program using connection mode TLI. The  */
/* program listens for incoming connections. When notification is received */
/* of an incoming connection a new transport endpoint is created and the  */
/* call is accepted on this new endpoint. Once the call is accepted a    */
/* a process is forked to transfer a file (specified in the command line)  */
/* to the client. The parent process returns to listening for incoming    */
/* calls. Once the child process has transferred the file it waits for a  */
/* disconnect to be received from the client.                            */
/*                                                                */
/* This program can be used in conjunction with the tclient and tclient-async */
/* programs.                                                            */
*****/

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

#include "osi_lib.h"

#define DLEN 1024
#define EOT 0x04
#define ESC 0x1b
#define DISCONNECT -1

void

```

```

Usage()

{
    printf("Usage: tserver <local_nsap> <local_tsel> -f <filename>\n");
    return;
}

/* accept_call() - accepts calls from clients.
 *
 * Unless an internal error occurs only a disconnect received from the remote
 * system can cause the t_accept call to fail.
 *
 * Input -
 * listen_fd - endpoint where server is currently listening.
 * call - t_call structure containing details of incoming call.
 * server_nsap - local nsap for binding.
 * server_tsel - local tsel for binding.
 *
 * Output -
 * function returns value of endpoint on which the call will be
 * handled.
 */

int
accept_call(
    int listen_fd,
    struct t_call*call,
    char *server_nsap,
    char *server_tsel
)
{
    int call_fd;
    struct t_bind*bind, *bound;

    if ((call_fd = t_open("/dev/otpi", O_RDWR,
                          (struct t_info *) NULL)) == -1) {
        t_error("t_open failed");
        exit(9);
    }

    /*
     * Allocate the t_bind structure to contain the local address.
     */

```

```

if ((bind = (struct t_bind *) t_alloc(call_fd, T_BIND,
                                     T_ALL)) == NULL) {
    t_error("t_alloc of t_bind structure failed");
    exit(10);
}

/*
 * Allocate a t_bind structure to contain the bound address.
 */

if ((bound = (struct t_bind *) t_alloc(call_fd, T_BIND,
                                       T_ALL)) == NULL) {
    t_error("t_alloc of t_bind (bound) structure failed");
    exit(11);
}

/*
 * Map the local address to the netbuf format.
 */

if ((bind->addr.len = tsap_to_net(bind->addr.buf, server_nsap,
                                server_tsel)) < 0) {
    fprintf(stderr, "Error mapping server address\n");
    exit(12);
}

/*
 * Bind to the local address.
 */

bind->qlen = 0;
if (t_bind(call_fd, bind, bound) < 0) {
    t_error("t_bind failed");
    exit(13);
}

/*
 * Ensure that the correct address was actually bound.
 */

if (bind->addr.len != bound->addr.len ||
    strcmp(bind->addr.buf, bound->addr.buf, bind->addr.len)) {

```

```
        fprintf(stderr, "t_bind bound to different address\n");
        exit(14);
    }

    /*
     * Attempt to accept the call.
     */
    if (t_accept(listen_fd, call_fd, call) < 0) {

        /*
         * Check to see if an asynchronous event has occurred.
         */

        if (t_errno == TLOOK) {

            /*
             * Probably a disconnect - otherwise a protocol error.
             * In any case terminate connection.
             */

            if (t_close(call_fd) < 0) {
                t_error("t_close failed for call_fd");
                exit(15);
            }

            return(DISCONNECT);
        }
        t_error("t_accept failed");
        exit(16);
    }

    return(call_fd);
}

/* run_server() - fork a process to do data transfer.
 *
 * The function forks a process which transfers a file to the client.
 *
 * Input -
 * listen_fd - endpoint for listening.
 * call_fd - endpoint for data transfer.
 * filename - file to be transferred
 */
```

```
run_server(
    int listen_fd,
    int call_fd,
    char *filename
)

{
    int nbytes, flags;
    FILE*fp; /* Pointer to file to be transferred */
    chardata_buf[DLEN];
    struct t_discon *discon;

    /*
     * Fork off process to handle file transfer for this call.
     */

    switch (fork()) {

    case -1: /* fork() failed */

        perror("fork failed");
        exit(18);

    default: /* Parent */

        /*
         * Close endpoint for handling call since the parent only
         * performs listens.
         */
        if (t_close(call_fd) < 0) {
            t_error("t_close failed for call_fd");
            exit(19);
        }
        return;

    case 0: /* Child */

        /*
         * Close the listening endpoint - the child does not listen.
         */

        if (t_close(listen_fd) < 0) {
            t_error("t_close failed for listen_fd");
```



```
        exit(20);
    }

    /*
     * Allocate a t_discon structure for use later.
     */

    if ((discon = (struct t_discon *) t_alloc(call_fd, T_DIS,
        T_ALL)) == NULL) {
        t_error("t_alloc of t_discon failed");
        exit(21);
    }

    /*
     * Open the file to be transferred, and transfer data.
     */
    if ((fp = fopen(filename, "r")) == NULL) {
        perror("Cannot open file for transfer");
        exit(22);
    }

    while ((nbytes = fread(data_buf, 1, 1024, fp)) > 0) {
        if (t_snd(call_fd, data_buf, nbytes, 0) < 0) {
            t_error("t_snd failed");
            exit(23);
        }
    }

    /*
     * Send end of transmission flags
     */

    data_buf[0] = ESC;
    data_buf[1] = EOT;

    if (t_snd(call_fd, data_buf, 2, 0) < 0) {
        t_error("t_snd failed");
        exit(24);
    }

    /*
     * Wait for disconnect.
     */
```

```

        if (t_rcv(call_fd, data_buf, nbytes, &flags) < 0) {
            if (t_rcvdis(call_fd, discon) < 0) {
                t_error("t_rcvdis failed");
                exit(25);
            }
        }
    else {
        /*
         * What we received is not a disconnect. In this
         * case the server will disconnect since this is an
         * invalid event.
         */

        if (t_snddis(call_fd, NULL) < 0) {
            t_error("t_snddis failed");
            exit(26);
        }
    }

    /*
     * Finished!
     */

    t_close(call_fd);
    exit(0);
}

/* main() - main part of server program.
 *
 * Endpoint is created and an address bound. Server waits for incoming calls.
 * Whenever an incoming call is received, accept_call() is called to accept
 * the incoming call, and then, if the call is accepted successfully,
 * run_server() is called to fork a process to do the data transfer.
 */

main(
    int argc,
    char **argv
)

{

```

```

int listen_fd, call_fd;
int nbytes, flags;
char data_buf[DLEN];
char server_nsap[32], server_tsel[32];
char filename[256];
struct t_bind* bind, *bound;
struct t_call* call;

/*
 * Get addressing information, and the name of the file to be
 * transferred, from the command line.
 */
get_addrs(argc, argv, NULL, NULL, server_nsap,
          server_tsel, filename);

if (strlen(filename) == 0)
    strcpy(filename, "/etc/termcap");

/*
 * Create the transport endpoint.
 */

if ((listen_fd = t_open("/dev/otpi", O_RDWR,
                      (struct t_info *) NULL)) == -1) {
    t_error("t_open failed");
    exit(1);
}

/*
 * Allocate the t_bind structure to contain the local address.
 */

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

/*
 * Allocate a t_bind structure to contain the bound address.
 */

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

```

```

exit(3);
}

/*
 * Map the local address to the netbuf format.
 */

if ((bind->addr.len = tsap_to_net(bind->addr.buf, server_nsap,
                                server_tsel)) < 0) {
    fprintf(stderr, "Error mapping server address\n");
    exit(4);
}

/*
 * Bind to the local address.
 */

bind->qlen = 1;
if (t_bind(listen_fd, bind, bound) < 0) {
    t_error("t_bind failed");
    exit(5);
}

/*
 * Ensure that the correct address was actually bound.
 */

if (bind->addr.len != bound->addr.len ||
    strncmp(bind->addr.buf, bound->addr.buf, bind->addr.len)) {
    fprintf(stderr, "t_bind bound to different address\n");
    exit(6);
}

/*
 * Allocate a t_call structure to contain the call details.
 */

if ((call = (struct t_call *) t_alloc(listen_fd, T_CALL,
                                     T_ALL)) == NULL) {
    t_error("t_alloc of t_call structure failed");
    exit(7);
}

```

```
/*
 * Enter infinite loop to wait for incoming connection indications.
 */

while (1) {

    printf("Waiting for a connection...\n");
    if (t_listen(listen_fd, call) < 0) {
        t_error("t_listen failed");
        exit(8);
    }
    printf("Incoming call received\n");
    if ((call_fd = accept_call(listen_fd, call, server_nsap,
        server_tsel)) != DISCONNECT) {
        run_server(listen_fd, call_fd, filename);
    }

    /*
     * Go back and listen for another call.
     */
}

/*
 * Not reached.
 */
}
```

## Connectionless Mode Client Example

This sample of code can be found on the product CD-ROM, in:  
 /opt/SUNWconn/osinet/example/tli/tclient-cltp.c.

```

/*
 * Copyright 1995 Sun Microsystems, Inc. All Rights Reserved
 */
/*****
/*      This code is provided for example purposes only and does not      */
/*      comprise part of the supported product.                          */
*****/

/*****
/*
/* tclient-cltp
/*
/* This program is a uses the connectionless transport protocol in blocking */
/* mode and acts as a client. The program sends a unitdata request which */
/* contains a string, to some other process acting as a server. It then */
/* waits for a response from the other process, and displays the contents of */
/* the unitdata indication received as a response. Queries are sent every */
/* five seconds in a continuous loop.
/*
/* This program interoperates with tserver-cltp.
/*
*****/

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

#include "osi_lib.h"

#define DLEN 1024

void
Usage()

{

```

```

        printf("Usage: tclient-cltp <local_nsap> <local_tsel> <remote_nsap>
<remote_tsel>\n");
        return;
    }

main(
    int argc,
    char **argv
)
{
    int fd;
    int nbytes, flags;
    chardata_buf[DLEN];
    char          client_nsap[32], server_nsap[32];
    char          client_tsel[32], server_tsel[32];
    struct t_bind*bind, *bound;
    struct t_call*call;
    struct t_unitdata *ud;
    struct t_uderr *uderr;

    /*
     * Parse argument list for addresses, etc.
     */
    get_addrs(argc, argv, client_nsap, client_tsel, server_nsap,
              server_tsel, NULL);

    /*
     * Create the transport endpoint.
     */

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

    /*
     * Allocate the t_bind structure to contain the local address.
     */

    if ((bind = (struct t_bind *) t_alloc(fd, T_BIND, T_ALL)) == NULL) {

```

```

t_error("t_alloc of t_bind structure failed");
exit(2);
}

/*
 * Allocate a t_bind structure to contain the bound address.
 */

if ((bound = (struct t_bind *) t_alloc(fd, T_BIND, T_ALL)) == NULL) {
t_error("t_alloc of t_bind (bound) structure failed");
exit(3);
}

/*
 * Map the local address to the netbuf format.
 */

if ((bind->addr.len = tsap_to_net(bind->addr.buf, client_nsap,
                                client_tsel)) < 0) {
fprintf(stderr, "Error mapping client address\n");
exit(4);
}

/*
 * Bind to the local address.
 */

bind->qlen = 0;
if (t_bind(fd, bind, bound) < 0) {
t_error("t_bind failed");
exit(5);
}

/*
 * Allocate structures to hold the unitdata and any error indications.
 */

if ((ud = (struct t_unitdata *) t_alloc(fd, T_UNITDATA, T_ALL))
    == NULL) {
t_error("t_alloc of t_unitdata failed");
exit(6);
}

if ((uderr = (struct t_uderr *) t_alloc(fd, T_UDERROR, T_ALL))

```



```

        == NULL) {
    t_error("t_alloc of t_uderr failed");
    exit(7);
}

/*
 * Loop sending out queries and waiting for responses.
 */

while (1) {

    /*
     * Send query to transaction server.
     */

    strcpy(ud->udata.buf,
    "This is a query for the transaction server\n");
    ud->udata.len = strlen(ud->udata.buf);

    if ((ud->addr.len = tsap_to_net(ud->addr.buf, server_nsap,
                                   server_tsel)) < 0) {
        fprintf(stderr, "Error mapping server address\n");
        exit(8);
    }

    printf("Sending query....\n");
    if (t_sndudata(fd, ud) < 0) {
        t_error("t_sndudata failed");
        exit(9);
    }

    /*
     * Wait for response.
     */

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

            int event;

            /*
             * Could be an error on a previously
             * sent datagram.

```

```

        */
        switch (event = t_look(fd)) {
        case T_UDERR:
            if (t_rcvuderr(fd, uderr) < 0) {
                t_error("Unrecoverable error");
                exit(10);
            }

                                fprintf(stderr,
                "Bad datagram error = %d\n",
                                uderr->error);

            break;
            default:
                printf("Event %d received\n", event);
                break;
            }
            continue;
        }
        t_error("t_rcvdata failed");
        exit(11);
    }

    printf("Response received: %s\n", ud->udata.buf);

    /*
     * Wait 5 seconds before sending another query.
     */

    sleep(5);

    }
}

```

## Connectionless Mode Server Example

This sample of code can be found on the product CD-ROM, in:  
 /opt/SUNWconn/osinet/example/tli/tserver-cltp.c.

```

/*
 * Copyright 1995 Sun Microsystems, Inc. All Rights Reserved
 */
/*****
/*      This code is provided for example purposes only and does not      */
/*      comprise part of the supported product.                          */
*****/

/*****
/*
/* tserver-cltp
/*
/* This is a server program using the connectionless transport protocol. */
/* The program acts like a transaction server. Whenever a message is      */
/* received the function do_query() is called to perform some action. In  */
/* this case the data in the received unitdata indication is printed out  */
/* and a text message is sent back to the client. It is assumed that both */
/* the received and returned messages fit into one buffer and the T_MORE  */
/* flag is not used.
/*
/* This program interoperates with tclient-cltp.
/*
*****/

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

#include "osi_lib.h"

#define DLEN 1024

void
Usage()

{
    printf("Usage: tserver-cltp <local_nsap> <local_tsel>\n");

```

```

        return;
    }

    /* query() - process a query.
     *
     * This is just a place-holder function for a real application. This function
     * prints out the contents of the query and formats a text message for the
     * response.
     *
     * Input -
     *   ud - unitdata containing the query.
     *
     * Output -
     *   ud - text message.
     */

    void
    query(
        struct t_unitdata *ud
    )
    {
        /*
         * Print out query - assume that it is a valid null terminated
         * string here.
         */
        printf("Query received: %s", ud->udata.buf);

        /*
         * Reply to query.
         */

        strcpy(ud->udata.buf, "Thank you for your query, please call again\n");
        ud->udata.len = strlen(ud->udata.buf);

        return;
    }

    main(
        int argc,
        char **argv
    )

```

```

{
    int fd;
    int nbytes, flags;
    chardata_buf[DLEN];
    char      server_nsap[32], server_tsel[32];
    struct t_bind*bind, *bound;
    struct t_call*call;
    struct t_unitdata *ud;
    struct t_uderr*uderr;

    /*
     * Get addressing information from the command line.
     */
    get_addrs(argc, argv, NULL, NULL, server_nsap,
              server_tsel, NULL);

    /*
     * Create the transport endpoint.
     */

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

    /*
     * Allocate the t_bind structure to contain the local address.
     */

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

    /*
     * Allocate a t_bind structure to contain the bound address.
     */

    if ((bound = (struct t_bind *) t_alloc(fd, T_BIND, T_ALL)) == NULL) {
        t_error("t_alloc of t_bind (bound) structure failed");
        exit(3);
    }
}

```

```

/*
 * Map the local address to the netbuf format.
 */

if ((bind->addr.len = tsap_to_net(bind->addr.buf, server_nsap,
                                server_tsel)) < 0) {
    fprintf(stderr, "Error mapping server address\n");
    exit(4);
}

/*
 * Bind to the local address.
 */

bind->qlen = 0;
if (t_bind(fd, bind, bound) < 0) {
    t_error("t_bind failed");
    exit(5);
}

/*
 * Allocate structures to hold the unitdata and any error indications.
 */

if ((ud = (struct t_unitdata *) t_alloc(fd, T_UNITDATA, T_ALL))
    == NULL) {
    t_error("t_alloc of t_unitdata failed");
    exit(6);
}

if ((uderr = (struct t_uderr *) t_alloc(fd, T_UDERROR, T_ALL))
    == NULL) {
    t_error("t_alloc of t_uderr failed");
    exit(7);
}

/*
 * Loop forever receiving and processing queries.
 */

while (1) {
    printf("Waiting for an incoming query...\n");
    if (t_rcvudata(fd, ud, &flags) < 0) {

```

```

        if (t_errno == TLOOK) {

            int event;

            /*
             * Could be an error on a previously
             * sent datagram.
             */
            switch (event = t_look(fd)) {
            case T_UDERR:
                if (t_rcvuderr(fd, uderr) < 0) {
                    t_error("Unrecoverable error");
                    exit(8);
                }
                fprintf(stderr,
                    "Bad datagram error = %d\n",
                                uderr->error);
                break;
            default:
                printf("Event %d received\n", event);

            break;
            }

            continue;
        }
        t_error("t_rcvdata failed");
        exit(9);
    }

    /*
     * query() does some processing of the request and places the
     * response in ud->udata.
     */
    query(ud);

    /*
     * Send the response. Note that the addr value in the
     * t_unitdata structure has not been changed - this contained
     * the address of the sending user, and is used to send the
     * response back.
     */

    if (t_sndudata(fd, ud) < 0) {

```

```
        t_error("t_sndudata failed");
        exit(11);
    }
}
```



## Non-Blocking Mode Client Example

This sample of code can be found on the product CD-ROM, in:  
 /opt/SUNWconn/osinet/example/tli/tclient-async.c.

```

/*
 * Copyright 1995 Sun Microsystems, Inc. All Rights Reserved
 */
/*****
/*      This code is provided for example purposes only and does not      */
/*      comprise part of the supported product.                          */
*****/

/*****
/* tclient-async
/*
/* This is a client program using non-blocking mode. The client sets up a
/* connection with a suitable server program and waits for the server to
/* transfer a file. The contents of this file are displayed on stdout.
/* Special flags are used by the server to indicate the end of the transfer.
/* When these flags are received, the client disconnects.
/*
/* This program interoperates with the tserver program.
/*
*****/

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

#include "osi_lib.h"

#define DLEN 1024
#define ESC 0x1b
#define EOT 0x04

void
Usage()

{

```

```

        printf("Usage: tclient-async <local_nsap> <local_tsel> <remote_nsap>
<remote_tsel>\n");
        return;
    }

main(
    int argc,
    char **argv
)
{
    int fd;
    int nbytes, flags;
    chardata_buf[DLEN];
    char          client_nsap[32], server_nsap[32];
    char          client_tsel[32], server_tsel[32];
    struct t_bind*bind, *bound;
    struct t_call*call;
    struct t_discon *discon;

    /*
     * Parse argument list for addresses, etc.
     */
    get_addrs(argc, argv, client_nsap, client_tsel, server_nsap,
              server_tsel, NULL);

    /*
     * Create the transport endpoint using non-blocking mode.
     */

    if ((fd = t_open("/dev/otpi", O_RDWR | O_NONBLOCK,
                    (struct t_info *) NULL)) == -1) {
        t_error("t_open failed");
        exit(1);
    }

    /*
     * Allocate the t_bind structure to contain the local address.
     */

    if ((bind = (struct t_bind *) t_alloc(fd, T_BIND, T_ALL)) == NULL) {

```

```

t_error("t_alloc of t_bind structure failed");
exit(2);
}

/*
 * Allocate a t_bind structure to contain the bound address.
 */

if ((bound = (struct t_bind *) t_alloc(fd, T_BIND, T_ALL)) == NULL) {
t_error("t_alloc of t_bind (bound) structure failed");
exit(3);
}

/*
 * Map the local address to the netbuf format.
 */

if ((bind->addr.len = tsap_to_net(bind->addr.buf, client_nsap,
                                client_tsel)) < 0) {
fprintf(stderr, "Error mapping client address\n");
exit(4);
}

/*
 * Bind to the local address.
 */

bind->qlen = 0;
if (t_bind(fd, bind, bound) < 0) {
t_error("t_bind failed");
exit(5);
}

/*
 * Allocate a t_call structure to contain the call details, only
 * allocating a buffer for the address.
 */

if ((call = (struct t_call *) t_alloc(fd, T_CALL, T_ALL)) == NULL) {
t_error("t_alloc of t_call structure failed");
exit(6);
}

/*

```

```

    * Map the remote address to the netbuf format.
    */

    if ((call->addr.len = tsap_to_net(call->addr.buf, server_nsap,
        server_tsel)) < 0) {
        fprintf(stderr, "Error mapping server address\n");
        exit(7);
    }

    /*
     * Attempt to set up connection with the server.
     */

    call->opt.len = 0;
    call->udata.len = 0;
    call->sequence = 0;
    if (t_connect(fd, call, (struct t_call *) NULL) == -1 ) {
        if (t_errno != TNOADATA) {
            t_error("t_connect failed");
            exit(8);
        }
    }

    /*
     * Allocate a t_discon structure in case we get a disconnect.
     */

    if ((discon = (struct t_discon *) t_alloc(fd, T_DIS, T_ALL)) == NULL) {
        t_error("t_alloc of t_discon structure failed");
        exit(9);
    }

    /*
     * Wait for the connection to be established.
     */
    while (t_rcvconnect(fd, call) < 0) {

        /*
         * Check to see what has happened.
         */
        switch (t_errno) {

```

```
case TNODATA:/* No connection yet */
    printf("Waiting for connect response....\n");
    sleep(1);
    continue;
case TLOOK:/* Probably disconnect - exit anyway */
    fprintf(stderr, "Unexpected event %d, exiting...\n",
            t_look(fd));
    t_close(fd);
    exit(0);
default:
    t_error("t_rcvconnect failed");
    exit(10);
}
}

/*
 * Receive data from server.
 */

while (1) {
    if ((nbytes = t_rcv(fd, data_buf, DLEN, &flags)) < 0) {

/*
 * If there is no data to read then check for other incoming
 * asynchronous events.
 */

        switch (t_errno) {

            case TNODATA:
                printf("No data to read...\n");
                sleep(1);
                continue;

            case TLOOK:
                switch (t_look(fd)) {

                    case T_DATA:
                        continue;
                    case T_DISCONNECT:
                        if (t_rcvdis(fd, discon) < 0) {
                            t_error("t_rcvdis failed");
                            exit(10);
                        }
                }
            }
        }
    }
}
```

```

        exit(0);
    case T_EXDATA:
        fprintf(stderr,
            "Not supporting expedited data\n");
    case T_LISTEN:
    case T_CONNECT:
    case T_UDERR:
    case T_ORDREL:
    default:
        fprintf(stderr, "Protocol error\n");
        exit(11);
    }

    default:
        t_error("t_rcv failed");
        exit(12);
    }
}

/*
 * Check for end of data.
 */

if (data_buf[0] != ESC) {

    /*
     * Write out data.
     */

    if (fwrite(data_buf, 1, nbytes, stdout) < (size_t) 0) {
        fprintf(stderr, "fwrite failed\n");
        exit(13);
    }
}
else {
    switch (data_buf[1]) {
    case EOT:

        /*
         * End of transfer - send disconnect.
         */

        if (t_snddis(fd, NULL) < 0) {

```

```
        t_error("t_snddis failed");
    }
    t_close(fd);
    exit(0);

default:

    /*
     * ESC only indicated "end of record".
     */

    if (fwrite(&data_buf[1], 1, nbytes-1, stdout)
        < (size_t) 0) {
        fprintf(stderr, "fwrite failed\n");
        exit(14);
    }
    break;
}
}
}
```

## OSI Library Use Examples

This sample of code can be found on the product CD-ROM, in:  
/opt/SUNWconn/osinet/example/libosi/uselib.c

```

/*
#ident "@(#)uselib.c1.2 24 Jan 1995 SMI"
*
* This program shows how to use the OSI library
* Usage:
* -n <NSAP>
* Gives the hostname associate to this NSAP
* -h <HOSTANME>
* Gives the NSAP associate to this HOSTNAME
* -s <SERVICE NAME>
* Gives the TSEL associate to this SERVICE NAME
* -l
* Gives the CLNS NSAP of my OSI stack
* -o
* Gives the CONS NSAP of my OSI stack
*/
#include <stdio.h>
#include <stdlib.h>

#include <osi.h>

#defineBUFLLEN50

int
main(
    intargc,
    char*argv[]
)
{
    char*s_nsap,
        *s_serv,
        *s_host;
    charnsap[BUFLLEN],
        host[BUFLLEN],
        tsel[BUFLLEN];
    int len,
        get_clns_nsap = 0,
        get_cons_nsap = 0,
        nsap_len = BUFLLEN,

```



```

        host_len = BUFLen,
        tsel_len = BUFLen;
int c;
extern char*optarg;
extern intoptbind;

s_nsap = (char *)NULL;
s_serv = (char *)NULL;
s_host = (char *)NULL;

while ((c = getopt(argc, argv, "n:h:s:lo")) != EOF) {
    switch (c) {
    case 'n':
        s_nsap = optarg;
        break;
    case 'h':
        s_host = optarg;
        break;
    case 's':
        s_serv = optarg;
        break;
    case 'l':
        get_clns_nsap = 1;
        break;
    case 'o':
        get_cons_nsap = 1;
        break;
    default:
        printf("Usage: %s -n <nsap> -h <hostname> -s <tsel> -l -o\n",
            argv[0]);
        break;
    }
}

if (s_nsap != (char *)NULL) {
    printf("Looking for hostname with NSAP = %s\n", s_nsap);
    if ((len = getnamebynsap(s_nsap, host, host_len)) < 0) {
        perror("getnamebynsap");
    }
    else {
        printf("\tFound HOSTNAME = %s with len = %d\n",
            host, len);
    }
}

```

```

if (s_host != (char *)NULL) {
printf("Looking for NSAP with hostname = %s\n", s_host);
if ((len = getnsapbyname(s_host, nsap, nsap_len)) < 0) {
    perror("getnsapbyname");
}
else {
    printf("\tFound NSAP = %s with len = %d\n",
        nsap, len);
}
}
if (s_serv != (char *)NULL) {
printf("Looking for TSEL with service = %s\n", s_serv);
if ((len = gettselbyname(s_serv, tsel, tsel_len)) < 0) {
    perror("gettselbyname");
}
else {
    printf("\tFound TSEL = %s with len = %d\n",
        tsel, len);
}
}

nsap[0] = 0;
if (get_clns_nsap == 1) {
printf("Looking for the CLNS NSAP...\n");
if ((len = getmyclnsnsap(nsap, nsap_len)) < 0) {
    perror("getmyclnsnsap");
}
else {
    printf("CLNS NSAP = %s with len = %d\n",
        nsap, len);
}
}

nsap[0] = 0;
if (get_cons_nsap == 1) {
printf("Looking for the CONS NSAP...\n");
if ((len = getmyconsnsap(nsap, nsap_len)) < 0) {
    perror("getmyconsnsap");
}
else {
    printf("CONS NSAP = %s with len = %d\n",
        nsap, len);
}
}

```

```
    }  
}
```

This sample of code can be found on the product CD-ROM, in:  
`/opt/SUNWconn/osinet/example/libosi/tbind.c`

```
/*  
#ident "@(#)tbind.c1.2 24 Jan 1995 SMI"  
*  
* This program shows how to use the OSI library with TLI  
*/  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <fcntl.h>  
#include <tiuser.h>  
  
#include <osi.h>  
  
int  
main(  
    int argc,  
    char *argv[]  
)  
{  
    char nsap[BUFSIZ],  
        clns_nsap[BUFSIZ],  
        tsel[BUFSIZ],  
        clns_buf[BUFSIZ],  
        otpi_buf[BUFSIZ];  
    int nsap_len,  
        clns_nsap_len,  
        tsel_len;  
    int fd_otpi, fd_clns;  
    struct t_bindb_otpi, b_clns;  
  
    /* Get the my NSAP to bind on for otpi  
    */  
    if ((nsap_len = getmyclnsnsap(nsap, BUFSIZ)) < 0) {  
        perror("getmyclnsnsap");  
        exit(1);  
    }
```

```

}

/* Since we cannot use the NSAP of the stack because it
 * identify the OSI transport, we should take a new one.
 * Be sure that the entry for this NSAP exist on
 * /etc/net/oclt/hosts
 */
if ((clns_nsap_len = getnsapbyname("clns", clns_nsap, BUFSIZ)) < 0) {
perror("getnsapbyname");
exit(1);
}

/* Get a TSEL for the bind on transport
 * A corresponding entry should exist on /etc/net/oclt/services
 */

if ((tsel_len = gettselbyname("test", tsel, BUFSIZ)) < 0) {
perror("gettselbyname");
exit(1);
}

/* Open the both file descriptors
 */

if ((fd_clns = t_open("/dev/clns",
                     O_RDWR, (struct t_info *)NULL)) == -1) {
t_error("t_open /dev/clns");
exit(1);
}
if ((fd_otpi = t_open("/dev/otpi",
                      O_RDWR, (struct t_info *)NULL)) < 0) {
t_error("t_open /dev/otpi");
exit(1);
}

/* Format the netbuf structure
 */
printf("Set the bind address for clns with nsap = 0x%s\n",
       clns_nsap);

b_clns.addr.len = nsap2net(clns_buf, clns_nsap, BUFSIZ);

if (b_clns.addr.len < 0) {
perror("nsap2net");
}

```

```
exit(1);
}

b_clns.addr.buf = clns_buf;
b_clns.qlen      = 0;

printf("Set the bind address for with nsap = 0x%s and TSEL = 0x%s\n",
      nsap, tsel);

b_otpi.addr.len = tsap2net(otpi_buf, nsap, tsel, BUFSIZ);

if (b_otpi.addr.len < 0) {
    perror("tsap2net");
    exit(1);
}

b_otpi.addr.buf = otpi_buf;
b_otpi.qlen      = 0;

/* Send the t_bind request
 */

if (t_bind(fd_clns, &b_clns, (struct t_bind *)NULL) < 0) {
    t_error("t_bind clns");
}

if (t_bind(fd_otpi, &b_otpi, (struct t_bind *)NULL) < 0) {
    t_error("t_bind otpi");
}

t_close(fd_clns);
t_close(fd_otpi);
}
```

## ***TLI over CLNS Example***

This sample of code can be found on the product CD-ROM, in:

/opt/SUNWconn/osinet/example/clns/bind\_clns.c

```
#ident "@(#)bind_clns.c 1.3 95/01/31 SMI"
/*
 * This example program shows how to use TLI over CLNS.
 * The main emphasis is on the address format to set in the
 * netbuf structure.
 *
 * Copyright 1995 Sun Microsystems, Inc. All Rights Reserved
 */

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

#define CHAR2INT(n)((n > 0x39)?(n - 'a' + 0xa):(n - 0x30))
#define MAX_LEN40

int
main(
    int argc,
    char*argv[]
)
{
    int fd,
        len;
    charch,
        *nsap;
    struct t_bind*bind;

    /* Just one argument to get the new NSAP to bind on
     * We cannot bind with a null NSAP
     */
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <NSAP to bind on>\n", argv[0]);
        fprintf(stderr, "\\tNSAP format is an ASCII string ");
        fprintf(stderr, "\\t or 0x follow by an hexadecimal number\\n");
        exit(1);
    }

    nsap = argv[1];

    /* Check the len of the address
     */

```

```

if (nsap[0] == '0' && nsap[1] == 'x') {
if (strlen(&nsap[2]) > MAX_LEN / 2) {
    fprintf(stderr, "NSAP too long\n");
    exit(1);
}
}
else {
if (strlen(nsap) > MAX_LEN) {
    fprintf(stderr, "NSAP too long\n");
    exit(1);
}
}

/* First step is to open the provider
*/
if ((fd = t_open("/dev/clns", O_RDWR, (struct t_info *) NULL)) < 0) {
t_error("open /dev/clns");
exit(1);
}

/* Allocate the bind structure
*/
if ((bind = (struct t_bind *)t_alloc(fd, T_BIND, T_ADDR)) == NULL) {
t_error("t_alloc T_BIND");
t_close(fd);
exit(1);
}

/* Set the netbuf associate to the bind structure
*/
if (nsap[0] == '0' && nsap[1] == 'x') {
char*buf;
int i, j, n, len;

buf = (char *)malloc(MAX_LEN * sizeof (char));

/* One byte of the NSAP is only
* half a byte of the network buffer
*/
i = 2; /* skip the '0x' */
j = 0;
len = 0;
do {
    n = CHAR2INT((int)nsap[i]);

```

```

        buf[j] = (unsigned char) n << 4;
        i++;
        if (nsap[i] == 0) {
            fprintf(stderr,
                "Incorrect length of the NSAP\n");
            t_close(fd);
            exit(1);
        }
        n = CHAR2INT((int)nsap[i]);
        buf[j] = (unsigned char) buf[j] + n;
        i++, j++, len++;
    } while (nsap[i] != 0);

    /* Set netbuf
    */
    bind->addr.len = len;
    bind->addr.buf = buf;
    }
    else {
        /* For string format we can directly use the received argument
        */
        bind->addr.len = strlen(nsap);
        bind->addr.buf = nsap;
    }

    /* We have to bind now.
    * Remember that the NSAP we bind on should be a new one
    * and it will be published in the ESIS table.
    */
    printf("Binding on CLNS ... ");
    if (t_bind(fd, bind, (struct t_bind *)NULL) < 0) {
        t_error("\nt_bind");
        t_close(fd);
        exit(1);
    }
    printf("done.\n");

    /* We are now bound on this nsap
    * You can now check that your new NSAP will appear in
    * the ESIS entry for this machine in the others hosts routing tables.
    * Also, all t_rcvudata() or t_sndudata() can be done
    * at this point in a standard way.
    */
    printf("If you want to check the ESIS table about this new NSAP\n");

```



```
printf("Then type return to continue\n");

ch = getchar();

/* Unbind does not need to be called if we use t_close()
 * but it works anyway.
 */
printf("Unbinding ... ");
if (t_unbind(fd) < 0) {
    t_error("\nt_unbind");
}
else {
    printf("done.\n");
}

t_close(fd);
}
```

## TLI over RFC1006 Examples

This sample of code can be found on the product CD-ROM, in:  
 /opt/SUNWconn/osinet/example/rfc1006/rfcclient.c

```

/*****
/*      This code is provided for example purposes only and does not      */
/*      comprise part of the supported product.                          */
*****/

/*****
/* rfcclient                                                                */
/*                                                                */
/* This is a client program provided to demonstrate the operation of the TLI */
/* interface when used over RFC1006. It can be used in conjunction with the */
/* rfcserver program also supplied.                                         */
/*                                                                */
/* This program opens a TLI connection in non-blocking mode, waits for a set */
/* amount of data to be transferred, and then disconnects the connection.  */
/*                                                                */
/* The user invokes the program using the command                          */
/*                                                                */
/* hostname% rfcclient remotehostname <tsel>                             */
/*                                                                */
/* where tsel is the Transport Selector on which the server process is      */
/* listening. In this example it is assumed to be a character string.      */
/*                                                                */
*****/

#include <sys/tiuser.h>
#include <stropts.h>
#include <fcntl.h>
#include <stdio.h>
#include <signal.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define MAXDATA1000000
#define BUFSIZE1024
#define MAXADDRLLEN 42

/*

```

```

    * The TSEL values in this example are hardcoded.  In a normal application
    * they would be configurable.
    */
#define TSEL"trs"

extern int t_errno;

main(int argc, char **argv)
{
    struct hostent*hostinfo;
    struct t_bind*sndbind, *rcvbind;
    struct t_call*call, *rcall;
    struct t_discon *discon;
    int fd, tslen, nslen, flags, nbytes, cc, i;
    charbuf[BUFSIZE], remote_tsel[32];
    unsigned chartaddr[MAXADDRLEN], ip_addr[4];

    /*
     * Get the IP address of the remote host from the supplied hostname.
     */
    if (argc < 2 || argc > 3) {
        printf("Usage: tli/rfcclient remotehost <remote_tsel>\n");
        exit(1);
    }

    if ((hostinfo = gethostbyname(argv[1])) == NULL) {
        perror("gethostbyname failed");
        exit(2);
    }
    memcpy(ip_addr, *hostinfo->h_addr_list, 4);

    /* Read the TSEL - if none present this will default to "trs" */
    if (argc == 3) {
        strcpy(remote_tsel, argv[2]);
    }
    else {
        strcpy(remote_tsel, TSEL);
    }

    /*
     * Print out remote host information.
     */

```

```

printf("tli/rfcclient - remote host %s: address %d.%d.%d.%d, TSEL \"%s\"\\n",
      argv[1], ip_addr[0], ip_addr[1], ip_addr[2], ip_addr[3],
      remote_tsel);

/*
 * Open the /dev/otk6 device.  Open it non-blocking in this example.
 */

if ((fd = t_open("/dev/otk6", O_RDWR | O_NONBLOCK
                , (struct t_info *)NULL)) < 0) {
t_error("t_open /dev/otk6 failed");
exit(3);
}

/*
 * Allocate t_bind structures for the address to be bound to and
 * the address actually bound.
 */

if ((sndbind = (struct t_bind *)t_alloc(fd, T_BIND, T_ALL)) == NULL) {
t_error("t_alloc sndbind");
t_close(fd);
exit(4);
}

if ((rcvbind = (struct t_bind *)t_alloc(fd, T_BIND, T_ALL)) == NULL) {
t_error("t_alloc rcvbind");
t_close(fd);
exit(5);
}

/*
 * Now bind to the address.
 *
 * Addresses are in netbuf format.  This is an array as follows:
 *
 * [tsaplen][tsap.....][nsaplen][nsap.....]
 *
 * where tsaplen and nsaplen are one byte each.
 */

```

```

/*
 * In this case we assume the TSEL is a string of characters - it
 * could be numeric.
 */
tslen = sizeof(TSEL) - 1; /* Don't include \0 as part of TSEL */
taddr[0] = (unsigned char)tslen;
memcpy(&taddr[1], TSEL, tslen);

/*
 * We do not have to supply an NSAP since we will bind to the default.
 */
nslens = 0;
taddr[tslen+1] = (unsigned char)nslens;

sndbind->addr.len = tslen + nslens + 2;
sndbind->addr.buf = (char *)taddr;
sndbind->qlen = 0; /* As a client we will not listen */

/*
 * Initialize the t_bind for returning the bound address.
 */

rcvbind->addr.maxlen = MAXADDRLEN;
rcvbind->addr.buf = (char *)taddr;

/*
 * Now do the bind.
 */

if (t_bind(fd, sndbind, rcvbind) < 0) {
    t_error("t_bind failed");
    t_close(fd);
    exit(6);
}
printf("tli/rfcclient - bind completed successfully\n");

/*
 * Allocate a t_call structure to hold the call information.
 */

if ((call = (struct t_call *)t_alloc(fd, T_CALL, T_ALL)) == NULL) {
    t_error("t_alloc failed call");
    t_close(fd);
    exit(7);
}

```

```

}

/*
 * Allocate a t_call structure to hold the returned call information.
 */

if ((rcall = (struct t_call *)t_alloc(fd, T_CALL, T_ALL)) == NULL) {
    t_error("t_alloc failed rcall");
    t_close(fd);
    exit(7);
}

/*
 * Called address is the same format as the bound address.  However,
 * we need to supply both an NSAP and an TSEL.  The NSAP will be
 * the IP address of the remote system.
 */

/*
 * As before assume that the TSEL is a character string.
 */

tslen = strlen(remote_tsel);
taddr[0] = (unsigned char)tslen;
memcpy(&taddr[1], remote_tsel, tslen);

/*
 * The IP address (ie. the NSAP) is always stored in 4 bytes.
 */

nslen = 4;
taddr[tslen+1] = (unsigned char)nslen;
memcpy(&taddr[tslen+2], ip_addr, 4);

call->addr.len = tslen + nslen + 2;
call->addr.buf = (char *)taddr;
call->opt.buf = NULL; /* No options are supported by RFC1006 */
call->opt.len = 0;
call->udata.len = 0;
call->sequence = 0;

/*
 * Now attempt a connection to the remote system.  If t_connect

```

```

    * fails with t_errno = TNOData then it means that the connection
    * has not completed yet.
    */
printf("tli/rfcclient - doing connect.....\n");
if (t_connect(fd, call, rcall) < 0) {
    if (t_errno == TNOData) {
        printf("tli/rfcclient - waiting on connection\n");
    }
    else {
        t_error("t_connect failed");
        t_close(fd);
        exit(8);
    }
}

/*
 * Wait in a loop for the connection to complete. We could also
 * do other useful things here in a real application.
 */
while (t_rcvconnect(fd, rcall) < 0) {

    /*
     * Need to check to see if this is a "real" error condition
     * or not.
     */

    switch (t_errno) {

    case TNOData: /* Still no connection */
        printf("tli/rfcclient - waiting on connection\n");
        sleep(1);
        continue;
    case TLOOK: /* This is most likely a disconnect */
        switch (t_look(fd)) {

            case T_DATA:
                continue;
            case T_DISCONNECT:
                printf("tli/rfcclient - disconnect received\n");
                if ((discon = (struct t_discon *)
                    t_alloc(fd, T_DIS, T_ALL)) == NULL) {
                    t_error("t_alloc failed discon"
                        );
                }
                exit(9);
        }
    }
}

```

```

        }
        if (t_rcvdis(fd, discon) < 0) {
            t_error("t_rcvdis failed");
            exit(10);
        }
        t_close(fd);
        exit(0);
    case T_EXDATA:
    case T_LISTEN:
    case T_CONNECT:
    case T_UDERR:
    case T_ORDREL:
    default:
        fprintf(stderr, "tli/rfcclient - Protocol error\n");
        t_close(fd);
        exit(11);
    }

default:
    t_error("t_rcvconnect failed");
    t_close(fd);
    exit(12);
}

}

/*
 * As an example receive some data from the remote system. Wait
 * until MAXDATA bytes of data have been received and then disconnect.
 */

cc = 0;
flags = 0;
while (cc < MAXDATA) {
    if ((nbytes = t_rcv(fd, buf, BUFSIZE, &flags)) < 0) {

        /*
         * Need to check if this is a real error condition
         * or not.
         */

        switch (t_errno) {

```



```
case TNODATA:
    printf("tli/rfcclient - No data to read...\n");
    sleep(1);
    continue;
case TLOOK:
    switch (t_look(fd)) {

        case T_DATA:
            printf("tli/rfcclient - T_DATA...\n");
            sleep(1);
            continue;
        case T_DISCONNECT:
            printf("tli/rfcclient - disconnect received during data reception\n");
            if ((discon = (struct t_discon *)
                t_alloc(fd, T_DIS, T_ALL))
                == NULL) {
                t_error("t_alloc failed discon"
                    );
                t_close(fd);
                exit(13);
            }
            if (t_rcvdis(fd, discon) < 0) {
                t_error("t_rcvdis failed");
                t_close(fd);
                exit(14);
            }
            exit(0);
        case T_EXDATA:
        case T_LISTEN:
        case T_CONNECT:
        case T_UDERR:
        case T_ORDREL:
        default:
            fprintf(stderr, "tli/rfcclient - Protocol error\n");
            t_close(fd);
            exit(15);
    }
default:
    t_error("t_rcv failed");
    t_close(fd);
    exit(16);
}
}
else { /* Data has been received */
```

```

        printf("tli/rfcclient - %d bytes of data received\n",
               nbytes);
        cc += nbytes;
    }
}

/*
 * All data received now disconnect the connection.
 */

printf("tli/rfcclient - %d bytes received - disconnecting\n", cc);
if (t_snddis(fd, NULL) < 0) {
    t_error("t_disconnect failed");
}
t_close(fd);
}

```

This sample of code can be found on the product CD-ROM, in:  
 /opt/SUNWconn/osinet/example/rfc1006/rfcserver.c

```

/*****
/*      This code is provided for example purposes only and does not      */
/*      comprise part of the supported product.                          */
*****/

/*****
/* rfcserver                                                                */
/*                                                                 */
/* This is a server program provided to demonstrate the operation of the TLI */
/* interface when used over RFC1006.  It can be used in conjunction with the */
/* rfcclient program also supplied.                                         */
/*                                                                 */
/* This program listens for incoming calls.  When an incoming call is      */
/* received it is accepted on a different file descriptor and a new process */
/* is forked to transfer data to the client.                                */
/*                                                                 */
/* The forked process transmits a set amount of data and then waits for the */
/* client to disconnect.                                                    */
/*                                                                 */
/* The user invokes the program using the command                          */
/*                                                                 */
/* hostname% rfcserver <tsel>                                             */
*****/

```

```

/*
/* where tsel is the Transport Selector on which this server process is
/* bound. In this example it is assumed to be a character string
/*
/*
/*****

#include <sys/tiuser.h>
#include <stropts.h>
#include <fcntl.h>
#include <stdio.h>
#include <signal.h>
#include <sys/types.h>

#define MAXDATA      1000000/* This is an arbitrary value */
#define BUFSIZE      1024
#define MAXADDRLEN   42
#define DISCONNECT-1
#define TSEL"trs"/* This is the default TSEL to listen on */

extern intt_errno;

/*
 * accept_call() - accepts calls from clients.
 *
 *Unless an internal error occurs only a disconnect received from the remote
 * system can cause the t_accept call to fail.
 *
 * Input -
 *         listen_fd - endpoint where server is currently listening.
 *         call - t_call structure containing details of incoming call.
 *         laddr - local address in netbuf format.
 *
 * Output -
 *         function returns value of endpoint on which the call will be
 *         handled.
 */

int
accept_call(int listen_fd, struct t_call *call, unsigned char *laddr)
{
    int call_fd, tslen, nslen;
    struct t_bind*bind, *bound;

```

```

/*
 * Open new file descriptor to accept call.
 */

if ((call_fd = t_open("/dev/otk6", O_RDWR,
    (struct t_info *) NULL)) == -1) {
    t_error("t_open (call_fd) failed");
    exit(8);
}

/*
 * Allocate t_bind structures to contain the address being bound to
 * and the address actually bound.
 */

if ((bind = (struct t_bind *)t_alloc(call_fd, T_BIND, T_ALL)) == NULL) {
    t_error("t_alloc bind failed");
    exit(9);
}

if ((bound = (struct t_bind *)t_alloc(call_fd, T_BIND, T_ALL)) == NULL){
    t_error("t_alloc bound failed");
    exit(10);
}

/*
 * Insert local address into bind structure. This is the same address
 * as was used to bind the listening file descriptor.
 */

tslen = laddr[0];
nslen = laddr[tslen+1];
bind->addr.buf = (char *)laddr;
bind->addr.len = tslen + nslen + 2;
bind->qlen = 0; /* We will not listen on this descriptor */

/*
 * Do the bind.
 */

if (t_bind(call_fd, bind, bound) < 0) {
    t_error("t_bind (call_fd) failed");
    exit(11);
}

```

```

/*
 * Attempt to accept the call on the new file descriptor.
 */

if (t_accept(listen_fd, call_fd, call) < 0) {

/*
 * Find out why accept failed.
 */

if (t_errno == TLOOK) {

    /*
     * This is probably a disconnect, so close the
     * file descriptor anyway. Could find out more
     * information using t_look.
     */
    t_close(call_fd);
    printf("tli/rfcserver - disconnect received\n");
    return(DISCONNECT);
}
t_error("t_accept failed"); /* Non "protocol" problem */
exit(12);
}

/*
 * Call has successfully been accepted so return the new file
 * descriptor.
 */

printf("tli/rfcserver - call successfully accepted\n");
return(call_fd);
}

/*
 * run_server() - fork a process to send some data.
 *
 * This function forks a process which transfers MAXDATA bytes of data to the
 * client. In a real application this would do something more useful.
 *
 * Input -
 *         listen_fd - endpoint for listening.
 *         call_fd - endpoint for data transfer.

```

```

*
*/

void
run_server(int listen_fd, int call_fd)
{
    int nbytes, cc, flags;
    charbuf[MAXDATA];
    struct t_discon *discon;

    /*
     * Fork process to handle data transfer.
     */

    switch (fork()) {

    case -1: /* fork() failed */
        perror("fork failed");
        exit(13);

    default: /* Parent */

        /*
         * Close file descriptor on which call was accepted since
         * the parent only performs listens.
         */
        t_close(call_fd);
        return;

    case 0: /* Child */

        /*
         * Close the listening file descriptor since the child does
         * not do any listens.
         */

        t_close(listen_fd);

        /*
         * Allocate a t_discon structure for use later.
         */

        if ((discon = (struct t_discon *) t_alloc(call_fd, T_DIS,

```

```

        T_ALL)) == NULL) {
    t_error("t_alloc of t_discon failed");
    exit(14);
}

/*
 * Now transfer the data.
 */
printf("tli/rfcserver - child transferring data\n");
cc = 0;
while (cc < MAXDATA) {
    if ((cc += t_snd(call_fd, buf, BUFSIZE, 0)) < 0) {
        t_error("t_snd failed");
        exit(15);
    }
}

/*
 * Wait for disconnect from the client.
 */

if (t_rcv(call_fd, buf, nbytes, &flags) < 0) {
    if (t_rcvdis(call_fd, discon) < 0) {
        t_error("t_rcvdis failed");
        exit(16);
    }
}
else {
    /*
     * What we just received was not a disconnect.  This
     * is invalid in this example so initiate a disconnect
     * from here.
     */

    if (t_snddis(call_fd, NULL) < 0) {
        t_error("t_snddis failed");
        exit(17);
    }
}

/*
 * Finished!
 */

```

```

    t_close(call_fd);
    exit(0);
}

main(int argc, char **argv)
{
    struct t_bind    *sndbind, *rcvbind;
    struct t_call    *call;
    int              fd, call_fd, tslen, nslen;
    char local_tsel[32];
    unsigned char taddr[MAXADDRLEN], baddr[MAXADDRLEN];

    /*
     * Get the local tsel.  Note that this is assumed to be a character
     * string in this example.
     */

    if (argc > 2) {
        printf("Usage: rfcserver <tsel>\n");
        exit(1);
    }
    else if (argc == 2) {
        strcpy(local_tsel, argv[1]);
    }
    else {
        strcpy(local_tsel, TSEL);
    }

    /*
     * Open the /dev/otk6 device.
     */

    if ((fd = t_open("/dev/otk6", O_RDWR, (struct t_info *)NULL)) < 0) {
        t_error("t_open /dev/otk6 failed");
        exit(2);
    }

    /*
     * Allocate t_bind structures for the address to be bound to and
     * the address actually bound.
     */

```



```

if ((sndbind = (struct t_bind *)t_alloc(fd, T_BIND, T_ALL)) == NULL) {
    t_error("t_alloc sndbind");
    t_close(fd);
    exit(3);
}

if ((rcvbind = (struct t_bind *)t_alloc(fd, T_BIND, T_ALL)) == NULL) {
    t_error("t_alloc rcvbind");
    t_close(fd);
    exit(4);
}

/*
 * Now bind to the address.
 *
 * Addresses are in netbuf format. This is an array as follows:
 *
 * [tsaplen][tsap.....][nsaplen][nsap.....]
 *
 * where tsaplen and nsaplen are one byte each.
 */

/*
 * In this case we assume the TSEL is a string of characters - it
 * could be numeric.
 */

tslen = strlen(local_tsel);
taddr[0] = (unsigned char)tslen;
memcpy(&taddr[1], local_tsel, tslen);

/*
 * We can bind with a NULL NSEL.
 */

nslen = 0;
taddr[tslen+1] = nslen;

sndbind->addr.len = tslen + nslen + 2;
sndbind->addr.buf = (char *)taddr;

/*
 * Set the qlen to 1 to receive one call at a time. In fact due to

```

```

    * a bug in the TLI spec trying to accept when there is more than one
    * call on the queue will cause problems.
    */

    sndbind->qlen = 1;

    /*
     * Initialize the t_bind for returning the bound address.
     */

    rcvbind->addr.maxlen = MAXADDRLEN;
    rcvbind->addr.buf = (char*)baddr;

    /*
     * Now do the bind.
     */

    if (t_bind(fd, sndbind, rcvbind) < 0) {
        t_error("t_bind failed");
        t_close(fd);
        exit(5);
    }
    printf("tli/rfcserver - bind completed successfully\n");

    /*
     * Allocate a t_call structure to hold the call information whenever
     * an incoming call is received.
     */

    if ((call = (struct t_call *)t_alloc(fd, T_CALL, T_ALL)) == NULL) {
        t_error("t_alloc failed call");
        t_close(fd);
        exit(6);
    }

    /*
     * Enter infinite loop waiting for incoming connections. Incoming
     * calls are allocated their own file descriptor in accept_call() and
     * a new process is forked for each accepted call in run_server().
     */

    while(1) {

        /*

```

```
    * Listen for incoming calls.
    */

    printf("tli/rfcserver - waiting for a connection....\n");
    if (t_listen(fd, call) < 0) {
        t_error("t_listen failed");
        exit(7);
    }
    printf("tli/rfcserver - incoming call received\n");

    /*
     * Accept call and fork process to do data transfer.
     */

    if ((call_fd = accept_call(fd, call, taddr)) != DISCONNECT) {
        run_server(fd, call_fd);
    }
}

/*
 * Not reached.
 */
}
```

## Makefile *Example*

This sample of code can be found on the product CD-ROM, in:  
/opt/SUNWconn/osinet/example/tli/Makefile.

```
# Makefile for the various TLI test programs.

TSEND    = tclient
TRECVC    = tserver
TSEND-CLTP    = tclient-cltp
TRECVC-CLTP    = tserver-cltp
TSEND-ASYNC    = tclient-async

CFLAGS      += -I /opt/SUNWspro -I /opt/SUNWconn/osinet/include -Xa

OSILIB      = -lnsl

CFILESTS    = tclient.c taddr.c targs.c
CFILESTR    = tserver.c taddr.c targs.c
CFILESTSC   = tclient-cltp.c taddr.c targs.c
CFILESTRC   = tserver-cltp.c taddr.c targs.c
CFILESTSA   = tclient-async.c taddr.c targs.c

OFILESTS    = tclient.o taddr.o targs.o
OFILESTR    = tserver.o taddr.o targs.o
OFILESTSC   = tclient-cltp.o taddr.o targs.o
OFILESTRC   = tserver-cltp.o taddr.o targs.o
OFILESTSA   = tclient-async.o taddr.o targs.o

all : $(TSEND) $(TRECVC) $(TSEND-CLTP) $(TRECVC-CLTP) $(TSEND-ASYNC)

$(TSEND) : $(OFILESTS)
        $(LINK.c) -o $(TSEND) $(OFILESTS) $(OSILIB)

$(TRECVC) : $(OFILESTR)
```

```
$(LINK.c) -o $(TREC) $(OFILSTR) $(OSILIB)

$(TSEND-CLTP) : $(OFILESTSC)
$(LINK.c) -o $(TSEND-CLTP) $(OFILESTSC) $(OSILIB)

$(TREC-CLTP) : $(OFILESTRC)
$(LINK.c) -o $(TREC-CLTP) $(OFILESTRC) $(OSILIB)

$(TSEND-ASYNC): $(OFILESTSA)
$(LINK.c) -o $(TSEND-ASYNC) $(OFILESTSA) $(OSILIB)
```



## *Part 2 — Function Call Reference*

---





## Using TLI Functions with SunLink OSI

## 8

<i>Summary of Call Order</i>	<i>page 118</i>
<i>Where to Find Descriptions</i>	<i>page 120</i>
<i>The netbuf Structure</i>	<i>page 121</i>
<i>Function Reference</i>	<i>page 122</i>

This chapter provides a reference for using TLI functions with SunLink OSI. They are given in alphabetical order. Read Chapter 5, “State Transition Tables” for the correct order of calls, and refer to Chapter 7, “Programming Examples” for some examples of how these calls can be used with SunLink OSI 8.1.

---

**Note** – The information given in this chapter is specific to SunLink OSI 8.1. The online manual pages give more general information on using the TLI functions independently of the transport provider.

---

The flowchart illustrates and summarizes the order and use of these functions, and Table 8-1 on page 120 provides a quick reference for finding each function description.

## Summary of Call Order

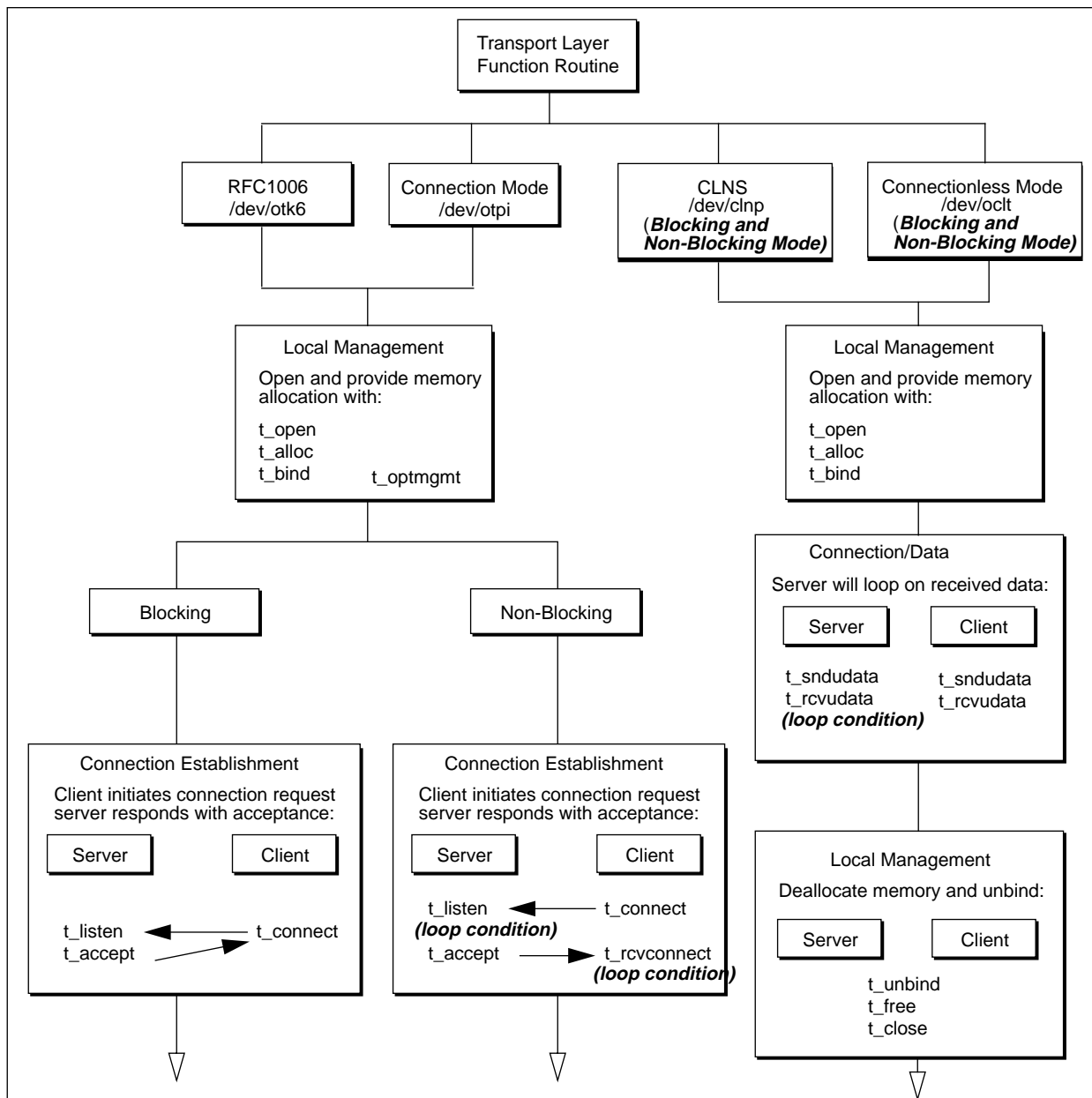
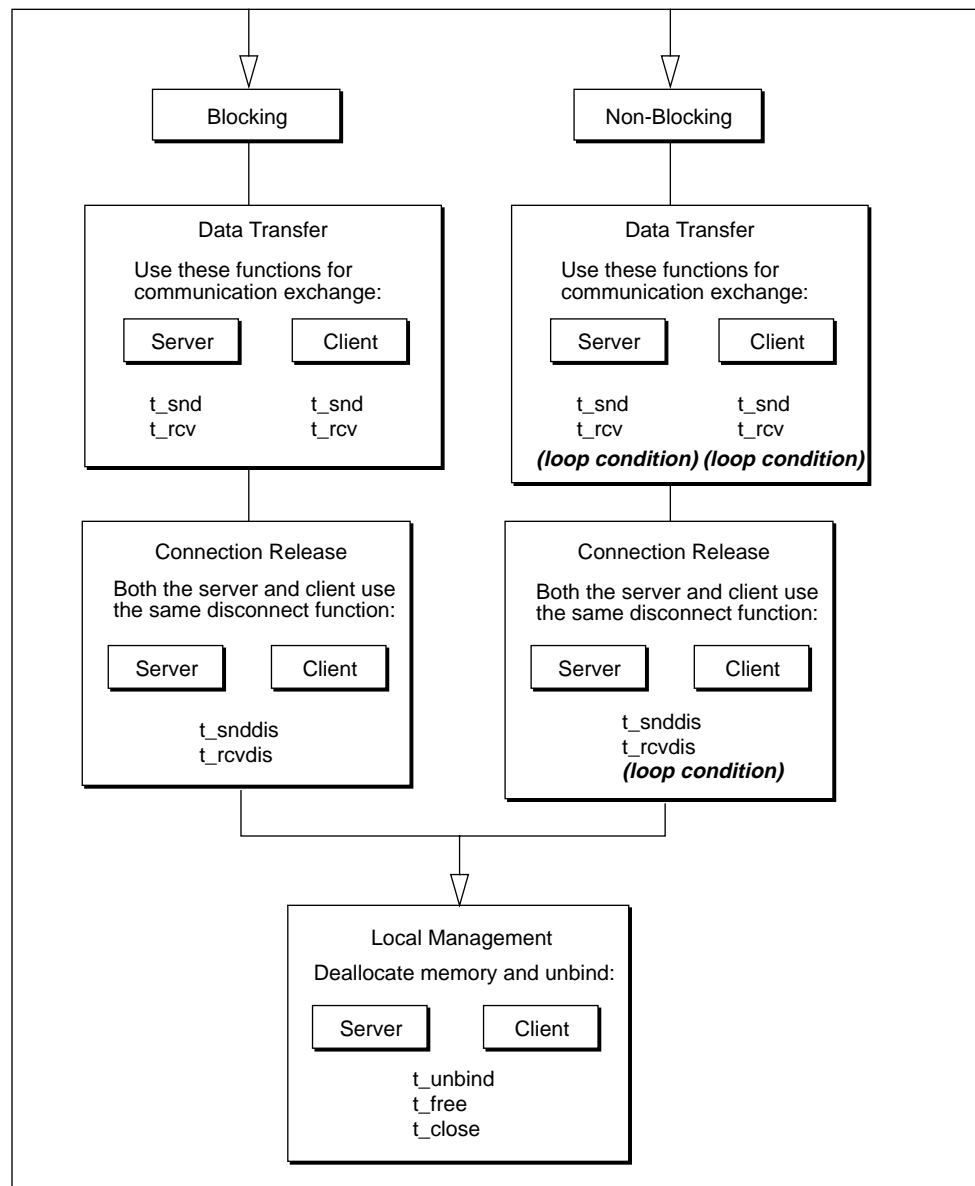


Figure 8-1 Call Order Summary



## Where to Find Descriptions

The following table shows the functions that are explained in this book and where you can find them.

*Table 8-1* Call Reference

Call	Description	Connection Mode	Connectionless Mode	Page
t_accept	Accepts a connection request	✓		122
t_alloc	Allocates a library structure	✓	✓	126
t_bind	Binds an address to an endpoint	✓	✓	129
t_close	Closes a transport endpoint	✓	✓	133
t_connect	Establishes a connection with another transport user	✓		135
t_error	Produces error messages	✓	✓	140
t_free	Frees a library structure	✓	✓	142
t_getinfo	Gets transport provider protocol information	✓	✓	144
t_getstate	Gets the current state	✓	✓	148
t_listen	Listens for a connect request	✓		150
t_look	Looks at the current event on a transport endpoint	✓	✓	153
t_open	Establishes a transport endpoint	✓	✓	155
t_optmgmt	Manages protocol options for a transport endpoint	✓		158
t_rcv	Receives data over a connection	✓		163
t_rcvconnect	Receives the confirmation from a connect request	✓		166
t_rcvdis	Receives information from a disconnect	✓		169
t_rcvudata	Receives a unit data indication		✓	172

Table 8-1 Call Reference

Call	Description	Connection Mode	Connectionless Mode	Page
t_rcvuderr	Receives a unit data error indication		✓	175
t_snd	Sends data over a connection	✓		178
t_snddis	Initiates a connection release	✓		181
t_sndudata	Sends a datagram		✓	184
t_sync	Synchronizes a transport library	✓	✓	187
t_unbind	Disables a transport endpoint	✓	✓	189

## The netbuf Structure

The *netbuf* structure is used in many of the TLI functions and is defined in the `tiuser.h` include file. It consists of:

```
struct netbuf {
    unsigned int maxlen; /* maximum length of buffer, bytes */
    unsigned int len;    /* number of bytes in address */
    char *buf;          /* pointer to address buffer */
};
```

where:

*maxlen* is the maximum length of the buffer in bytes. It has no meaning for the *req* argument. If *maxlen* is not large enough to hold the returned address, an error results. If you use `t_alloc` with `T_ALL`, *maxlen* will automatically be set large enough.

*len* specifies the number of bytes in the bound address.

*buf* points to the address buffer containing the data produced with the `tsaptonet` function.

## Function Reference

The remainder of this chapter is an alphabetical reference of the available functions.

### t\_accept

Accepts a connect request.

#### **Synopsis**

```
#include <tiuser.h>
#include <osi_lib.h>

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

#### **Use**

The t\_accept function has two main purposes:

- To allow the user to specify a new transport endpoint to handle an incoming call.
- To cause the transport provider to respond to the incoming call request. This corresponds to the sending of a Call Confirm PDU in OSI transport.

---

**Note** – This function cannot be used in applications developed over CLNS.

---

#### **Description**

The parameters are:

*fd* identifies the local transport endpoint where the connect request arrived.

*resfd* specifies the local transport endpoint where the connection is to be established.

*call* contains information required by the transport provider to complete the connection. *call* points to a `t_call` structure that contains the following members:

```
struct t_call {
    struct netbuf addr;           /* address          */
    struct netbuf opt;           /* options         */
    struct netbuf udata;         /* user data       */
    int sequence;                /* sequence number */
};
```

where:

*addr* is the address of the caller.

*opt* indicates any protocol-specific parameters associated with the connection. These are described in `t_optmgmt`. The values of parameters specified by *opt* and the syntax of those values are protocol specific.

*udata* points to any user data to be returned to the caller.

*sequence* is the value returned by `t_listen` that uniquely associates the response with a previously received connect indication.

The *netbuf* structure is contained in the `tiuser.h` include file. Refer to page 121 for details.

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

If a different transport endpoint is specified (*resfd* is not equal to *fd*), the endpoint must be bound to a TSAP address and must be in the `T_IDLE` state (see `t_getstate`) before `t_accept` is issued.

For both types of endpoints, `t_accept` will fail and set `t_errno` to `TLOOK` if there are indications (for example, a connect or disconnect) waiting to be received on that endpoint. See “Connection Establishment Phase” on page 11 for the special implications for `TLOOK` with connection mode.

The *udata* argument enables the called transport user to send user data to the caller and the amount of user data must not exceed the limits supported by the transport provider, as returned in the *connect* field of the *info* argument of *t\_open* or *t\_getinfo*. If the *len* field of *udata* is zero, no data is sent to the caller.

## Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned, *t\_errno* is set to indicate the error, and *errno* might be set.

## Errors

On failure, *t\_errno* will be set to one of the following:

TACCES

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

TBADF

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

TBADDATA

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

TBADOPT

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

TBADSEQ

An invalid sequence number was specified.

TLOOK

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



---

**TNOTSUPPORT**

This function is not supported by the underlying transport provider.

**TOUTSTATE**

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

**TSYSERR**

A system error has occurred during execution of this function, *errno* will be set to the specific error.

## t\_alloc

Allocates a library structure.

### Synopsis

```
#include <tiuser.h>
#include <osi_lib.h>

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

### Use

- The `t_alloc` function allocates memory for a specified structure and for buffers referenced by the structure.

---

**Note** – This function cannot be used in applications developed over CLNS.

---

### Description

The parameters are:

*fd* identifies the local transport endpoint for the connection.

*struct\_type* identifies the structure to allocate and can be one of the following structures:

Type	Structure
T_BIND	struct t_bind
T_CALL	struct t_call
T_OPTMGMT	struct t_optmgmt
T_DIS	struct t_discon

Type	Structure
T_UNITDATA	struct t_unitdata
T_UDERROR	struct t_uderr
T_INFO	struct t_info

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

Each of these structures except T\_INFO, contains at least one field of a *netbuf* structure. The *netbuf* structure is contained in the `tiuser.h` include file. Refer to page 121 for details. For each field of this type, the user can specify that the buffer for that field should be allocated as well.

The *fields* argument specifies which buffers to allocate, where the argument is the bitwise-OR of any of the following:

Argument	Description
T_ADDR	The <i>addr</i> field of the <code>t_bind</code> , <code>t_call</code> , <code>t_unitdata</code> , or <code>t_uderr</code> structures.
T_OPT	The <i>opt</i> field of the <code>t_optmgmt</code> , <code>t_call</code> , <code>t_unitdata</code> , or <code>t_uderr</code> structures.
T_UDATA	The <i>udata</i> field of the <code>t_call</code> , <code>t_discon</code> , or <code>t_unitdata</code> structures.
T_ALL	All relevant fields of the given structure. Allocates all <i>netbuf</i> buffers associated with <code>t_bind</code> , not just the structures.

For each field specified, `t_alloc` allocates memory for the buffer associated with that field, and initializes the *buf* pointer and *maxlen* in *netbuf*. The length of the buffer allocated is based on the same size information that is returned to the user on `t_open` and `t_getinfo`. Thus, *fd* must refer to the transport endpoint through which the newly allocated structure will be passed, so that the appropriate size information can be accessed. If the size value associated with any specified field is -1 or -2, `t_alloc` is unable to determine the size of the buffer to allocate and will fail, setting `t_errno` to `TSYSERR` and `errno` to `EINVAL`. A value of -1 specifies that there is no limit on the size of a TSDU, and a value of -2 specifies that the transfer of normal data is not supported by the transport provider.

For any field not specified, *buf* is set to NULL and *maxlen* is set to zero.

Using `t_alloc` to allocate structures helps to ensure the compatibility of user programs with future releases of the transport interface.

Use `t_free` to deallocate memory that was allocated with `t_alloc`.

## **Return Values**

On successful completion, `t_alloc` returns a pointer to the newly allocated structure. On failure, NULL is returned, *t\_errno* is set to indicate the error, and *errno* might be set.

## **Errors**

On failure, *t\_errno* will be set to one of the following:

TBADF

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

TSYSERR

A system error has occurred during execution of this function, *errno* will be set to the specific error.

## t\_bind

Associates or binds an address to a transport endpoint.

### Synopsis

```
#include <tiuser.h>
#include <osi_lib.h>

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

### Use

t\_bind establishes the identity of the endpoint that has been opened between a user and the transport provider. It binds a TSAP address to the transport endpoint and activates that transport endpoint, so that the transport provider can begin accepting or requesting connections on the transport endpoint.

The specific format and contents of the address are meaningful only to the transport provider, and the address field is handled transparently by the TLI library.

### Description

The parameters are:

*fd* identifies the local transport endpoint for the connection.

*req* and *ret* arguments point to a t\_bind structure defining the address associated with the endpoint. It contains the following members:

```
struct t_bind {
    struct netbuf addr; /* address */
    unsigned qlen;      /* queue length */
};
```

where:

*addr* specifies the TSAP address. The format of the TSAP address for SunLink OSI 8.1 is described in Chapter 3, “Addressing.” Note that if you are interfacing to RFC1006, the value for the NSAP field is `rk6`.

*qlen* indicates the maximum number of outstanding connect indications. For connectionless mode this is not significant. For connection mode it defines the maximum number of incoming requests that can be queued.

The *netbuf* structure is contained in the `tiuser.h` include file. Refer to page 121 for details.

*req* is the address to be bound, while *ret* is the address that was actually bound. This may differ from the address specified in *req*, since if the requested address is not available, or if no address is specified in *req* (the *len* field of *addr* in *req* is NULL), SunLink OSI 8.1 assigns an appropriate address to be bound, and returns that address in the *addr* field of *ret*. The user can compare the addresses in *req* and *ret* to determine whether the transport provider bound the transport endpoint to a different address than that requested.

To specify that SunLink OSI 8.1 provides the TSAP address to be bound, set *req* as NULL. The value of *qlen* is assumed to be zero, and the transport provider must assign an address to the transport endpoint. In this case, SunLink OSI assigns a default address for X.25 CONS.

Similarly, *ret* may be NULL if the user is not interested in knowing the bound address and is not interested in the negotiated value of *qlen*. It is possible to set *req* and *ret* to NULL for the same call, in which case the transport provider chooses the address to bind and does not return that information to the user.

The *qlen* field has meaning only when initializing a connection mode service. It specifies the number of outstanding connect indications the transport provider should support for the given transport endpoint. An outstanding connect indication is one that has been passed to the transport user by the transport provider.

If the value of *qlen* is 1, then each connect request is processed completely before the next request is dequeued. A value greater than 1 means that the server can dequeue several requests before responding to any of them. The value of *qlen* is negotiated by the transport provider and may be changed if the

transport provider cannot support the specified number of outstanding connect indications. On return, the *qlen* field in *ret* contains the negotiated value.

More than one transport endpoint can be bound to the same TSAP address. If a user binds more than one transport endpoint to the same TSAP, only one endpoint can be used to listen for connect indications associated with that TSAP. In other words, only one *t\_bind* for a given TSAP may specify a value of *qlen* greater than zero. In this way, the transport provider can identify which transport endpoint should be notified of an incoming connect indication. If a user attempts to bind a TSAP to a second transport endpoint with a value of *qlen* greater than zero, a *TBADADDR* error is returned.

---

**Note** – Each transport provider manages its address space differently. Some transport providers allow a single transport address to be bound to several transport endpoints, while others require a unique address per endpoint. The TLI library supports both. SunLink OSI determines if it can bind the requested address. If not, it chooses another valid address from its address space and binds it to the transport endpoint. As a result the server should check the bound address to ensure that it is one of which client programs are aware.

---

It is not possible to bind more than one TSAP to the same transport endpoint.

SunLink OSI 8.1 supports both connection-oriented and connectionless-oriented protocols. Since each uses a different NSAP, the TSAP address for connection and connectionless networks are different. Therefore, the transport class (TC 0, 1, 2, 3, and 4 over CONS, and TC 4 over CLNS) is selected after the endpoint has been bound.

### **Return Codes**

*t\_bind* returns 0 on success. On failure, *t\_bind* returns -1, *t\_errno* is set to indicate the error, and *errno* might be set.

### **Errors**

On failure, *t\_errno* is set to one of the following:

TACCES

The user does not have permission to use the specified address.

## TBADADDR

The specified TSAP was in an incorrect format or contained illegal information.

## TBADF

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

## TBUFOVFLW

The number of bytes allowed for an incoming argument is not sufficient to store the value of that argument. The provider's state will change to `T_IDLE` and the information to be returned in *ret* is discarded.

## TNOADDR

The transport provider could not allocate an address. `T_IDLE` and the information to be returned in *ret* will be discarded.

## TOUTSTATE

The function was issued in the wrong sequence.

## TSYSERR

A system error has occurred during execution of this function, *errno* will be set to the specific error.



## t\_close

Closes a transport endpoint.

### **Synopsis**

```
#include <tiuser.h>
#include <osi_lib.h>

int t_close(
    int fd
);
```

### **Use**

The `t_close` function informs the transport provider that the user has finished with the transport endpoint and frees any local library resources associated with that endpoint. It also closes the file associated with the transport endpoint.

### **Description**

The parameters are:

*fd* specifies the transport endpoint.

`t_close` should be called from the `T_UNBND` state (see `t_getstate`). However, this function does not check state information, so it may be called from any state to close a transport endpoint. If this occurs, the local library resources associated with the endpoint will be freed automatically. In addition, `close(2)` will be issued for that file descriptor; the close is abortive if no other process has that file open, and breaks any transport connections that are associated with that endpoint.

### **Return Values**

`t_close` returns 0 on success. On failure `t_close` returns -1, `t_errno` is set to indicate the error, and `errno` might be set.

## ***Errors***

On failure, *t\_errno* is set to the following:

TBADF

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

TSYSERR

A system error occurred during execution of this function, *errno* is set to the specific error.

## t\_connect

Establishes a connection with another transport user.

### Synopsis

```
#include <tiuser.h>
#include <osi_lib.h>

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

### Use

- The `t_connect` function allows a transport user to request a connection to the specified destination transport user.

---

**Note** – This function cannot be used in applications developed over CLNS.

---

### Description

The parameters are:

*fd* identifies the local transport endpoint on which the connection is to be established.

*sndcall* and *rcvcall* point to a `t_call` structure that contains the following members:

```
struct t_call {
    struct netbuf addr;           /* address          */
    struct netbuf opt;           /* options         */
    struct netbuf udata;         /* user data       */
    int sequence;               /* sequence number */
};
```

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

The *netbuf* structure is contained in the `tiuser.h` include file. Refer to page 121 for details.

In *sndcall*:

*addr* specifies the TSAP address of the destination transport user. Note that if you are interfacing to RFC1006, the NSAP is replaced by a 4 octet IP address. Refer to “RFC1006 Addressing” on page 24,” for details.

*opt* presents any protocol-specific information that might be needed by the transport provider. The options are the same as those described in the `t_optmgmt` call.

*udata* points to optional user data that may be passed to the destination transport user during connection establishment. The amount of data allowed depends on the value specified in the `t_info` structure returned by `t_open`.

*sequence* has no meaning for this function.

On return in *rcvcall*:

*addr* returns the TSAP associated with the responding transport endpoint.

*opt* presents any protocol-specific information associated with the connection. The options are the same as those described in the `t_optmgmt` call.

*udata* points to optional user data that may be returned by the destination transport user during connection establishment.

*sequence* has no meaning for this function.

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

The *udata* argument enables the caller to pass user data to the destination transport user and receive user data from the destination user during connection establishment. However, the amount of user data must not exceed the limits supported by the transport provider as returned in the connect field of the *info* argument of `t_open` or `t_getinfo`. If the *len* field of *udata* is zero for *sndcall*, no data is sent to the destination transport user.

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

By default, `t_connect` executes in synchronous mode, and waits for the destination user's response before returning control to the local user. A successful return (that is, return value of zero) indicates that the requested connection has been established. However, if `O_NDELAY` or `O_NONBLOCK` is set (using `t_open` or `fcntl(2)`), `t_connect` executes in asynchronous mode.

In this case, the call does not wait for the remote user's response, but will return control immediately to the local user and return -1 with *t\_errno* set to `TNODATA` to indicate that the connection has not yet been established. In this way, the function simply initiates the connection establishment procedure by sending a connect request to the destination transport user.

If the `t_connect` call was made in blocking mode, the function returns if the call is accepted by the server or if the call is rejected by the server or the transport provider.

When making a call using `t_connect` the client can make use of two facilities supplied by the SunLink OSI transport provider:

- User data may be sent with the call by filling the user data field on the transport connect request PDU. The amount of data which can be sent is defined in the connect parameter in the *t\_info* structure returned by the `t_open` function.
- Protocol specific options can be negotiated for the call. The same options and format are used as for the `t_optmgmt` function.

Both these facilities are protocol dependent.

The `t_connect` call is used differently for non-blocking mode. See Chapter 4, “Non-Blocking Mode” for details.

## **Return Codes**

`t_connect` returns 0 on success. On failure `t_connect` returns -1, `t_errno` is set to indicate the error, and *errno* might be set.

## **Errors**

On failure, `t_errno` will be set to one of the following:

TACCES

The user does not have permission to use the specified address or options.

TBADADDR

The specified TSAP was in an incorrect format or contained illegal information.

TBADDATA

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

TBADF

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

TBADOPT

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

TBUFOVFLW

The number of bytes allocated for an incoming argument is not sufficient to store the value of that argument. If executed in synchronous mode, the provider's state, as seen by the user, changes to `T_DATAXFER`, and the connect indication information to be returned in *rcvcall* is discarded.

TLOOK

An asynchronous event has occurred on this transport endpoint and requires immediate attention.

TNODATA

---

`O_NDELAY` or `O_NONBLOCK` was set, so the function successfully initiated the connection establishment procedure, but did not wait for a response from the remote user.

`TNOTSUPPORT`

This function is not supported by the underlying transport provider.

`TOUTSTATE`

The function was issued in the wrong sequence.

`TSYSERR`

A system error has occurred during execution of this function, *errno* will be set to the specific error.

`t_error`

Produces error messages.

## Synopsis

```
#include <tiuser.h>
#include <osi_lib.h>

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

## Use

- This produces a message on the standard error output which describes the last error encountered during a call to a TLI function.

---

**Note** – This function cannot be used in applications developed over CLNS.

---

## Description

The parameters are:

*errmsg* is a user-supplied error message explaining a context to the error.

*t\_errno* contains a value which describes the transport function error connected with the user-supplies error message. If *t\_errno* is `TSYSERR`, `t_error` also prints the standard error message for the current value contained in *errno*.

*t\_errlist* is the array of message strings, to allow user message formatting. *t\_errno* can be used as an index into this array to retrieve the error message string (without a terminating newline).

*t\_nerr* is the maximum index value for the *t\_errlist* array.

*t\_errno* is set when an error occurs and is not cleared on subsequent successful calls.



---

For example, if a `t_connect` function fails on transport endpoint *fd2* because an incorrect address was given, the following call might follow the failure:

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

The diagnostic message would print as:

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

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

## t\_free

Frees a library structure.

### Synopsis

```
#include <tiuser.h>
#include <osi_lib.h>

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

### Use

- The `t_free` function frees memory previously allocated by `t_alloc`. This function frees memory for the specified structure, and can also free memory for buffers referenced by the structure.

---

**Note** – This function cannot be used in applications developed over CLNS.

---

### Description

The parameters are:

*ptr* points to one of the six structure types described for `t_alloc`.

*struct\_type* identifies the type of that structure, which can be one of the following:

Type	Structure
T_BIND	struct t_bind
T_CALL	struct t_call
T_OPTMGMT	struct t_optmgmt
T_DIS	struct t_discon

---

Type	Structure
T_UNITDATA	struct t_unitdata
T_UDError	struct t_uderr
T_INFO	struct t_info

---

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

The `t_free` call checks the *addr*, *opt*, and *udata* fields of the given structure (as appropriate), and frees the buffers pointed to by the *buf* field of the *netbuf* structure. If *buf* is NULL, `t_free` will not attempt to free memory. After all buffers are freed, `t_free` frees the memory associated with the structure pointed to by *ptr*.

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

### **Return Values**

`t_free` returns 0 on success. On failure `t_free` returns -1, `t_errno` is set to indicate the error, and *errno* might be set.

### **Errors**

On failure, `t_errno` will be set to the following:

TSYSERR

A system error has occurred during execution of this function, *errno* will be set to the specific error.

## t\_getinfo

Gets protocol-specific service information.

### Synopsis

```
#include <tiuser.h>
#include <osi_lib.h>

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

### Use

The `t_getinfo` function returns the current characteristics of the SunLink OSI 8.1 transport protocol. It returns the same information as that returned by the `t_open` call, but enables access to this information during any phase of communication.

### Description

The parameters are:

*fd* identifies the transport endpoint.

*info* points to a `t_info` structure containing the following members:

```
struct t_info {
    long addr;          /* size of TSAP */
    long options;       /* size of protocol options */
    long tsdu;          /* size of max transport service data unit */
    long etsdu;         /* size of expedited tsdu */
    long connect;       /* max data for connection primitives */
    long discon;        /* max data for disconnect primitives */
    long servtype;      /* provider service type */
};
```

where:

*addr*

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

*options*

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

*tsdu*

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

*etsdu*

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

*connect*

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

### *discon*

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

### *servtype*

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

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

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

Type	Description
T_COTS	The transport provider supports a connection mode service but does not support the optional orderly release facility.
T_CLTS	The transport provider supports a connectionless mode service. For this service type, <code>t_open</code> returns -2 for <i>etsdu</i> , <i>connect</i> , and <i>discon</i> .

### **Return Values**

`t_getinfo` returns 0 on success. On failure `t_getinfo` returns -1, `t_errno` is set to indicate the error, and *errno* might be set.

---

### ***Errors***

On failure, `t_errno` is set to one of the following:

`TBADF`

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

`TSYSERR`

A system error has occurred during execution of this function, *errno* is set to the specific error.

## t\_getstate

Gets the current state.

### Synopsis

```
#include <tiuser.h>
#include <osi_lib.h>

int t_getstate(
    int fd
);
```

### Use

The `t_getstate` function returns the current state of the transport provider associated with the transport endpoint.

### Description

`fd` specifies the transport endpoint for which the current state is required.

### Return Values

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

State	Description
T_UNBND	Unbound
T_IDLE	Idle
T_OUTCON	Outgoing connection pending
T_INCON	Incoming connection pending
T_DATAXFER	Data transfer

If the provider is undergoing a state transition when `t_getstate` is called, the function fails.



**Errors**

On failure, `t_errno` is set to one of the following:

`TBADF`

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

`TSTATECHNG`

The transport provider is undergoing a state change.

`TSYSERR`

A system error has occurred during execution of this function, *errno* will be set to the specific error.

## t\_listen

Listens for a connect request.

### Synopsis

```
#include <tiuser.h>
#include <osi_lib.h>

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

### Use

- The server waits for incoming calls using the `t_listen` function, which blocks until an incoming call is received. When it receives a call, it must accept or reject it.

---

**Note** – This function cannot be used in applications developed over CLNS.

---

### Description

The parameters are:

*fd* identifies the local transport endpoint where connect indications arrive and on return.

*call* contains information describing the connect indication. It points to a `t_call` structure, which contains the following members:

```
struct t_call {
    struct netbuf addr;           /* address          */
    struct netbuf opt;           /* options          */
    struct netbuf udata;         /* user data        */
    int sequence;                /* sequence number */
};
```

The *netbuf* structure is contained in the `tiuser.h` include file. Refer to page 121 for details.

It returns a `t_call` structure which contains the following values:

*addr* returns the TSAP of the calling transport user.

*opt* returns protocol-specific parameters associated with the connect request.

*udata* returns any user data sent by the caller on the connect request.

*sequence* is a number that uniquely identifies the returned connect indication. The value of *sequence* enables the user to listen for multiple connect indications before responding to any of them.

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

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

The server must accept or reject the client's request. It calls `t_accept` to establish the connection, or `t_snddis` to reject the request.

The `t_listen` call is used differently for non-blocking mode. See Chapter 4, "Non-Blocking Mode" for details.

### **Return Values**

`t_listen` returns 0 on success. On failure `t_listen` returns -1, `t_errno` is set to indicate the error, and *errno* might be set.

### **Errors**

On failure, `t_errno` is set to one of the following:

`TBADF`

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

## TBUFOVFLW

The number of bytes allocated for an incoming argument is not sufficient to store the value of that argument. The provider's state, as seen by the user, changes to `T_INCON`, and the connect indication information to be returned in call is discarded.

## TLOOK

An asynchronous event has occurred on this transport endpoint and requires immediate attention.

## TNODATA

`O_NDELAY` or `O_NONBLOCK` was set, but no connect indications had been queued.

## TNOTSUPPORT

This function is not supported by the underlying transport provider.

## TSYSERR

A system error has occurred during execution of this function, *errno* will be set to the specific error.




---

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

---

`t_look`

Looks at the current event on a transport endpoint.

### ***Synopsis***

```
#include <tiuser.h>
#include <osi_lib.h>

int t_look(
    int fd
);
```

### ***Use***

- This function returns the current event on the specified transport endpoint.

---

**Note** – This function cannot be used in applications developed over CLNS.

---

### ***Description***

The parameters are:

*fd* identifies the transport endpoint.

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

### ***Return Values***

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

`T_LISTEN`

Connection indication received.

T\_CONNECT

Connect confirmation received.

T\_DATA

Normal data received.

T\_EXDATA

Expedited data received.

T\_DISCONNECT

Disconnect received.

T\_UDERR

Datagram error indication.

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

### **Errors**

On failure, `t_errno` is set to one of the following:

TBADF

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

TSYSERR

A system error has occurred during execution of this function, *errno* is set to the specific error.

## t\_open

Establishes a transport endpoint.

### Synopsis

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

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

### Use

t\_open establishes a transport endpoint by opening a file that identifies the SunLink OSI transport provider.

### Description:

The parameters are:

*path* identifies the device driver. This can be one of the following:

Table 8-2

file	transport provider
/dev/otpi	SunLink OSI connection-oriented transport layer protocol
/dev/oclt	SunLink OSI connectionless transport layer protocol
/dev/otk6	RFC1006
/dev/clnp	CLNS

Only one such service can be associated with the transport provider identified by t\_open.

*oflag* identifies any open flags (see online manual page `open(2)`). It is constructed from `O_NDELAY` or `O_NONBLOCK` OR-ed with `O_RDWR`. These flags are defined in the header file `<fcntl.h>` and are used to select non-blocking mode.

*info* points to a `t_info` structure which returns information about the default characteristics of the transport endpoint. If *info* is set to `NULL` by the transport user, no protocol information is returned by `t_open`.

`t_open` returns a file descriptor (*fd*) that is used by all subsequent functions to identify the particular local transport endpoint.

Each transport provider performs a subset of the services provided by TLI. The transport provider characteristics determine the extent of the services that it provides. The `t_open` function returns the default characteristics of a transport endpoint.

These can be changed in connection mode after an endpoint has been opened for negotiated options or using the `t_optmgmt` function to set the default options for the endpoint. The `t_getinfo` function returns the current characteristics of a transport endpoint.

The contents of the `t_info` structure are:

```
struct t_info {
    long addr;           /* size of TSAP */
    long options;        /* size of protocol options */
    long tsdu;           /* size of max transport service data unit */
    long etsdu;          /* size of expedited tsdu */
    long connect;        /* max data for connection primitives */
    long discon;         /* max data for disconnect primitives */
    long servtype;       /* provider service type */
};
```

where:

*addr* is the maximum size of a transport address.

*options* is the maximum bytes of protocol-specific options that may be passed between the transport user and transport provider.

*tsdu* is the maximum message size that may be transmitted in either connection mode or connectionless mode.



*etsdu* is the maximum expedited data message size that may be sent over a transport connection.

*connect* is the maximum number of bytes of user data that may be passed between users during connection establishment.

*discon* is the maximum bytes of user data that may be passed between users during the abortive release of a connection.

*servtype* is the type of service supported by the transport provider.

The service types defined by TLI in SunLink OSI are:

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

Note that the orderly disconnection service specified by T\_COTS\_ORD is not supported by SunLink OSI.

The `t_open` call is used differently for non-blocking mode. See Chapter 4, “Non-Blocking Mode” for details.

### **Return Values**

`t_open` returns a valid file descriptor on success. On failure `t_open` returns -1, `t_errno` is set to indicate the error, and *errno* might be set.

### **Errors**

On failure, `t_errno` is set to one of the following:

TBADFLAG

An invalid flag is specified.

TSYSERR

A system error has occurred during execution of this function, *errno* is set to the specific error.

## t\_optmgmt

Manages options for a transport endpoint.

### Synopsis

```
#include <tiuser.h>
#include <osi_lib.h>

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

### Use

The `t_optmgmt` function enables a transport user to retrieve, verify, or negotiate protocol options with the transport provider.

Each transport protocol defines its own set of negotiable protocol options. Since these options are protocol-specific, applications required to be protocol-independent should not use this function.

### Description

The parameters are:

*fd* identifies a bound transport endpoint.

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

```
struct t_optmgmt {
    struct netbuf opt;    /* options */
    long flags;          /* flags */
};
```

The `netbuf` structure is contained in the `tiuser.h` include file. Refer to page 121 for details. The *maxlen* argument in `netbuf` has no meaning for the *req* argument.\*

where:

*opt* identifies protocol options.

*flags* specifies the action to take with those options. The options are represented by a *netbuf* structure in a manner similar to the address in *t\_bind*.

*req* is used to request a specific action of the provider and to send options to the provider.

SunLink OSI supports four parameters which are passed in a single structure in the *t\_optmgmt* call:

```
struct T_opt_details {
    unsigned short T_opt_idus;    /* IDU size when sending */
    unsigned short T_opt_idur;    /* IDU size when receiving */
    unsigned char  T_opt_clo;     /* Option class */
    unsigned char  T_opt_qos;     /* Quality of Service */
};
```

where:

*T\_opt\_idus* and *T\_opt\_idur* are the sending and receiving Interface Data Unit (IDU) size. The Interface Data Unit (IDU) sizes indicates the maximum amount of data in bytes which may be exchanged with the transport provider in a single read or write. The size of the sending IDU cannot be set with *T\_opt\_idus*, since this value is overridden by the value set in the SunLink OSI 8.1 configuration tool (*ositool*). The size of the receiving IDU specified in *t\_opt\_idur* overrides the values set in the configuration for this connection. A value of zero ensures that the default value according to the configuration is used for this connection. A returned value of zero indicates that the default value is being used.

*T\_opt\_clo* describes the class option field. This is a bit mask, where the bits may be set as follows:

Option	Value	Description
USE_EXPEDITED_CLASS_2	0x01	Select expedited option if class 2 is requested
NO_FLOW_CONTROL_CLASS_2	0x10	No flow control in class 2
USE_CLASS_4_3_2	0x20	Class 4/3/2 is used if class 0 is not proposed
USE_CLASS_0	0x80	Class 0 as the alternative class

These are defined in the `osi_lib.h` include file. The default value for the *T\_opt\_clo* parameter is 0xa1, which sets bits 0x01, 0x20 and 0x80. This is only relevant for transport over CONS and is ignored for transport class 4 CLNP or when using the connectionless protocol.

*T\_opt\_qos* describes the Quality of Service field. This has no significance for SunLink OSI 8.0.1 and should always contain a zero value.

The transport provider may return options and flag values to the user through *ret*. For *ret*, *maxlen* specifies the maximum size of the options buffer and *buf* points to the buffer where the options are to be placed. On return, *len* specifies the number of bytes of options returned. The *maxlen* argument must be set in the *ret* argument to specify the maximum number of bytes the options buffer can hold. The actual structure and content of the options is imposed by the transport provider.

---

**Note** – When using the programming interface to CLNS, you may find it useful set *len* to specify the size of the ESIS routing table, or if you are not using all of the entries in the ESIS routing table, to a size large enough to contain the entries that you are using.

---

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

Action	Description
T_NEGOTIATE	This action enables the user to negotiate the values of the options specified in <i>req</i> with the transport provider. The provider will evaluate the requested options and negotiate the values, returning the negotiated values through <i>ret</i> .
T_CHECK	This action enables the user to verify whether the options specified in <i>req</i> are supported by the transport provider. On return, the <i>flags</i> field of <i>ret</i> has either T_SUCCESS or T_FAILURE set to indicate whether the options are supported. These flags are only meaningful for the T_CHECK request.
T_DEFAULT	This action enables a user to retrieve the default options supported by the transport provider into the <i>opt</i> field of <i>ret</i> . In <i>req</i> , the <i>len</i> field of <i>opt</i> must be zero and the <i>buf</i> field may be NULL.

### Return Codes

*t\_optmgmt* returns 0 on success. On failure *t\_optmgmt* returns -1, *t\_errno* is set to indicate the error, and *errno* might be set.

### Errors

On failure, *t\_errno* will be set to one of the following:

TBADF

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

TACCES

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

TBADFLAG

An invalid flag was specified.

TBADOPT

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

## TBUFOVFLW

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

## TOUTSTATE

The function was issued in the wrong sequence.

## TSYSERR

A system error has occurred during execution of this function, *errno* will be set to the specific error.

`t_rcv`

Receives data or expedited data sent over a connection.

### ***Synopsis***

```
#include <tiuser.h>
#include <osi_lib.h>

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

### ***Use***

- This function receives either normal or expedited data for connection mode. SunLink OSI 8.1 supports expedited data for the transport classes that support and when it is negotiated. Transport class 0 does not support expedited data.

---

**Note** – This function cannot be used in applications developed over CLNS.

---

### ***Description***

The parameter are:

*fd* identifies the local transport endpoint through which data will arrive.

*buf* points to a receive buffer where user data will be placed.

*nbytes* specifies the size of the receive buffer. The maximum message size allowed is returned with the `t_open` call or can be retrieved using `t_getinfo`.

*flags* may be set on return from `t_rcv` and specifies the optional flags described below.

By default, `t_rcv` operates in synchronous mode and waits for data to arrive if none is currently available. However, if `O_NDELAY` or `O_NONBLOCK` is set (using `t_open` or `fcntl(2)`), `t_rcv` will execute in asynchronous mode and will fail if no data is available. (See `TNODATA` below.)

On return from the call, if `T_MORE` is set in *flags*, this indicates that there is more data and the current transport service data unit (TSDU) or expedited transport service data unit (ETSDU) must be received in multiple `t_rcv` calls. Each `t_rcv` with the `T_MORE` flag set indicates that another `t_rcv` must follow to get more data for the current TSDU. The end of the TSDU is identified by the return of a `t_rcv` call with the `T_MORE` flag not set.

On return, the data returned is expedited data if `T_EXPEDITED` is set in *flags*. If the number of bytes of expedited data exceeds *nbytes*, `t_rcv` sets `T_EXPEDITED` and `T_MORE` on return from the initial call. The use of expedited data is explained in more detail in “Data Transfer” on page 15.

Subsequent calls to retrieve the remaining TSDU will have `T_EXPEDITED` set on return. The end of the ETSDU is identified by the return of a `t_rcv` call with the `T_MORE` flag not set.

If expedited data arrives after part of a TSDU has been retrieved, receipt of the remainder of the TSDU will be suspended until the ETSDU has been processed. Only after the full ETSDU has been retrieved (`T_MORE` not set) will the remainder of the TSDU be available to the user. Note that there is no relationship of how data is divided into units between the user to TLI transfer and the way in which the transport provider handles it. If too much data is received, flow control may be used according to the transport class defined for the connection.

The `t_rcv` call is used differently for non-blocking mode. See Chapter 4, “Non-Blocking Mode” for details.

## **Return Values**

On successful completion, `t_rcv` returns the number of bytes received. On failure `t_rcv` returns -1, `t_errno` is set to indicate the error, and *errno* might be set.

## **Errors**

On failure, `t_errno` will be set to one of the following:



**TBADF**

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

**TLOOK**

An asynchronous event has occurred on this transport endpoint and requires immediate attention. `TNODATA`, `O_NDELAY` or `O_NONBLOCK` was set, but no data is currently available from the transport provider.

**TNOTSUPPORT**

This function is not supported by the underlying transport provider.

**TSYSERR**

A system error has occurred during execution of this function, *errno* will be set to the specific error (`EPROTO`).

## t\_rcvconnect

Receives the confirmation from a connect request.

### Synopsis

```
#include <tiuser.h>
#include <osi_lib.h>

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

### Use

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

---

**Note** – This function cannot be used in applications developed over CLNS.

---

### Description

The parameters are:

*fd* identifies the local transport endpoint where communication will be established.

*call* contains information associated with the newly established connection. It points to a `t_call` structure which contains the following members:

```
struct t_call {
    struct netbuf addr;           /* address          */
    struct netbuf opt;           /* options          */
    struct netbuf udata;         /* user data        */
    int sequence;               /* sequence number */
};
```

where:

*addr* returns the TSAP associated with the responding transport endpoint.

*opt* presents any protocol-specific information associated with the connection.

*udata* points to optional user data that may be returned by the destination transport user during connection establishment.

*sequence* has no meaning for this function.

The *netbuf* structure is contained in the `tiuser.h` include file. Refer to page 121 for details.

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

If `O_NDELAY` or `O_NONBLOCK` is set (using `t_open` or `fcntl(2)`), `t_rcvconnect` executes in asynchronous mode, and reduces to a poll for existing connect confirmations. If none are available, `t_rcvconnect` fails and returns immediately without waiting for the connection to be established. (See `TNODATA` below.) `t_rcvconnect` must be re-issued at a later time to complete the connection establishment phase and retrieve the information returned in call.

The `t_rcvconnect` call is used differently for non-blocking mode. See Chapter 4, “Non-Blocking Mode” for details.

### **Return Values**

`t_rcvconnect` returns 0 on success. On failure `t_rcvconnect` returns -1, *t\_errno* is set to indicate the error, and *errno* might be set.

### **Errors**

On failure, *t\_errno* will be set to one of the following:

TBADF

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

## TBUFOVFLW

The number of bytes allocated for an incoming argument is not sufficient to store the value of that argument and the connect information to be returned in call will be discarded. The provider's state, as seen by the user, will be changed to `DATAXFER`.

## TLOOK

An asynchronous event has occurred on this transport connection and requires immediate attention.

## TNODATA

`O_NDELAY` or `O_NONBLOCK` was set, but a connect confirmation has not yet arrived.

## TNOTSUPPORT

This function is not supported by the underlying transport provider.

## TSYSERR

A system error has occurred during execution of this function, *errno* will be set to the specific error.

## t\_rcvdis

Retrieves information from disconnect.

### Synopsis

```
#include <tiuser.h>
#include <osi_lib.h>

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

### Use

- This function is used to identify the cause of a disconnect, and to retrieve any user data sent with the disconnect. It cleans up after receiving a t\_snddis call. The t\_rcvdis call is used for connection mode only.

---

**Note** – This function cannot be used in applications developed over CLNS.

---

### Description

The parameters are:

*fd* identifies the local transport endpoint where the connection existed.

*discon* points to a t\_discon structure containing the following members:

```
struct t_discon {
    struct netbuf udata;      /* user data          */
    int reason;               /* reason code        */
    int sequence;             /* sequence number    */
};
```

where:

*udata* identifies any user data that was sent with the disconnect.

*reason* specifies the reason for the disconnect through a protocol-dependent reason code.

*sequence* may identify an outstanding connect indication with which the disconnect is associated. It is only meaningful when `t_rcvdis` is issued by a passive transport user who has executed one or more `t_listen` functions and is processing the resulting connect indications. If a disconnect indication occurs, *sequence* can be used to identify which of the outstanding connect indications is associated with the disconnect.

The *netbuf* structure is contained in the `tiuser.h` include file. Refer to page 121 for details.

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

## Return Values

`t_rcvdis` returns 0 on success. On failure `t_rcvdis` returns -1, `t_errno` is set to indicate the error, and *errno* might be set.

## Errors

On failure, `t_errno` will be set to one of the following:

TBADF

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

TBUFOVFLW

The number of bytes allocated for incoming data is not sufficient to store the data. The provider's state, as seen by the user, will change to `T_IDLE`, and the disconnect indication information to be returned in *discon* will be discarded.

---

**TNODIS**

No disconnect indication currently exists on the specified transport endpoint.

**TNOTSUPPORT**

This function is not supported by the underlying transport provider.

**TSYSERR**

A system error has occurred during execution of this function, *errno* will be set to the specific error.

## t\_rcvudata

Receives a unit data indication.

### Synopsis

```
#include <tiuser.h>
#include <osi_lib.h>

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

### Use

This function is used to receive a datagram from another transport user. It is used for connectionless mode only.

### Description

The parameters are:

*fd* identifies the local transport endpoint through which data will be received.

*unitdata* holds information associated with the received data unit. It points to a *t\_unitdata* structure containing the following members:

```
struct t_unitdata {
    struct netbuf addr;          /* address    */
    struct netbuf opt;          /* options   */
    struct netbuf udata;        /* user data  */
};
```

where:

*addr* holds the source address of incoming datagrams and the destination address of outgoing datagrams.



*opt* holds any protocol options on the datagram. For connectionless mode, SunLink OSI does not support protocol options so *maxlen* is set to zero in the *opt netbuf* structure.

*udata* holds the data.

*flags* is set on return to indicate that the complete data unit was not received.

The *netbuf* structure is contained in the `tiuser.h` include file. Refer to page 121 for details.

The *maxlen* field (in *netbuf*) of *addr*, *opt*, and *udata* must be set before issuing this function to indicate the maximum size of the buffer for each. The *addr*, *opt*, and *udata* fields must all be allocated with buffers large enough to hold any possible incoming values. If `t_alloc` is used with a `T_ALL` argument to allocate the `t_unitdata` structure the TLI library sets the *maxlen* field of each *netbuf* structure accordingly.

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

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

By default, `t_rcvudata` operates in synchronous mode and waits for a data unit to arrive if none is currently available. However, if `O_NDELAY` or `O_NONBLOCK` is set (using `t_open` or `fcntl(2)`), `t_rcvudata` executes in asynchronous mode and will fail if no data units are available.

The `t_rcvudata` call is used differently for non-blocking mode. See Chapter 4, “Non-Blocking Mode” for details.

### **Return Values**

`t_rcvudata` returns 0 on successful completion. On failure `t_rcvudata` returns -1, `t_errno` is set to indicate the error, and *errno* might be set.

## **Errors**

On failure, `t_errno` is set to one of the following:

TBADF

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

TBUFOVFLW

The number of bytes allocated for the incoming TSAP or options is greater than zero but not sufficient to store the information. The unit data information to be returned in *unitdata* is discarded.

TLOOK

An asynchronous event has occurred on this transport endpoint and requires immediate attention.

TNODATA

`O_NDELAY` or `O_NONBLOCK` was set, but no data units are currently available from the transport provider.

TNOTSUPPORT

This function is not supported by the underlying transport provider.

TSYSERR

A system error has occurred during execution of this function, *errno* is set to the specific error.

## t\_rcvuderr

Receives a unit data error indication.

### Synopsis

```
#include <tiuser.h>
#include <osi_lib.h>

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

### Use

- This function is used to receive information concerning an error on a previously sent data unit, and should be issued only after a unit data error indication. It informs the transport user that a data unit with a specific destination address and protocol options produced an error. It is only used for connectionless mode.

---

**Note** – This function cannot be used in applications developed over CLNS.

---

### Description

The parameters are:

*fd* identifies the local transport endpoint through which the error report will be received.

*uderr* points to a `t_uderr` structure containing the following members:

```
t_uderr {
    struct netbuf addr;    /* address    */
    struct netbuf opt;    /* options    */
    long error;           /* long error */
};
```

where:

*addr* identifies the destination address specified in the bad datagram.

*opt* identifies the protocol options specified in the bad datagram

*error* is a protocol-specific error code.

The *netbuf* structure is contained in the `tiuser.h` include file. Refer to page 121 for details.

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

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

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

## Return Values

`t_rcvuderr` returns 0 on successful completion. On failure `t_rcvuderr` returns -1, `t_errno` is set to indicate the error, and *errno* might be set.

## Errors

On failure, `t_errno` is set to one of the following:

TBADF

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

TBUFOVFLW

The number of bytes allocated for the incoming TSAP or options is not sufficient to store the information. The unit data error information to be returned in *uderr* will be discarded.

TNOTSUPPORT

This function is not supported by the underlying transport provider.

---

**TNOUDERR**

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

**TSYSERR**

A system error has occurred during execution of this function, *errno* will be set to the specific error.

`t_snd`

Sends data or expedited data over a connection.

### ***Synopsis***

```
#include <tiuser.h>
#include <osi_lib.h>

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

### ***Use***

- This function sends either normal or expedited data for connection mode. SunLink OSI 8.1 supports expedited data for the transport classes that support it and when it is negotiated. Transport class 0 does not support expedited data.

---

**Note** – This function cannot be used in applications developed over CLNS.

---

### ***Description***

The parameters are:

*fd* identifies the local transport end-point over which data should be sent.

*buf* points to the user data.

*nbytes* specifies the number of bytes of user data to be sent.

*flags* specifies any optional flags described below.

By default, `t_snd` operates in synchronous mode and may wait if flow control restrictions prevent the data from being accepted by the local transport provider at the time the call is made. However, if `O_NDELAY` or `O_NONBLOCK` is set (using `t_open` or `fcntl(2)`), `t_snd` executes in asynchronous mode, and will fail immediately if there are flow control restrictions.

Even when there are no flow control restrictions, `t_snd` waits if STREAMS internal resources are not available, regardless of the state of `O_NDELAY` or `O_NONBLOCK`.

On successful completion, `t_snd` returns the number of bytes accepted by the transport provider. Normally this equals the number of bytes specified in *nbytes*. However, if `O_NDELAY` or `O_NONBLOCK` is set, it is possible that only part of the data will be accepted by the transport provider.

In this case, `t_snd` sets `T_MORE` for the data that was sent and returns a value less than *nbytes* indicating the amount of data remaining. If *nbytes* is zero and sending of zero bytes is not supported by the underlying transport provider, `t_snd` will return -1 with `t_errno` set to `TBADDATA`. A return value of zero indicates that the request to send a zero-length data message was sent to the provider.

If `T_EXPEDITED` is set in *flags*, the data is sent as expedited data, depending on which expedited data has been negotiated. The support of expedited data is described in “Data Transfer Phase” on page 18.

If `T_MORE` is set in *flags*, or is set as described above, an indication is sent to the transport provider that the transport service data unit (TSDU) or expedited transport service data unit (ETSDU) is being sent through multiple `t_snd` calls. Each `t_snd` with the `T_MORE` flag set indicates that another `t_snd` follows with more data for the current TSDU. The end of the TSDU (or ETSDU) is identified by a `t_snd` call with the `T_MORE` flag not set. Use of `T_MORE` enables a user to break up large logical data units without losing the boundaries of those units at the other end of the connection. The flag implies nothing about how the data is packaged for transfer below the transport interface.

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

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

The `t_snd` call is used differently for non-blocking mode. See Chapter 4, “Non-Blocking Mode” for details.

## **Return Values**

On successful completion, `t_snd` returns the number of bytes accepted by the transport provider. On failure `t_snd` returns -1, `t_errno` is set to indicate the error, and `errno` might be set.

## **Errors**

On failure, `t_errno` will be set to one of the following:

`TBADDATA`

*nbytes* is zero and sending zero bytes is not supported by the transport provider.

`TBADF`

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

`TFLOW`

`O_NDELAY` or `O_NONBLOCK` was set, but the flow control mechanism prevented the transport provider from accepting data at this time.

`TNOTSUPPORT`

This function is not supported by the underlying transport provider.

`TSYSERR`

A system error (see `intro(2)`) has been detected during execution of this function.



## t\_snddis

Sends a user-initiated disconnect request.

### Synopsis

```
#include <tiuser.h>
#include <osi_lib.h>

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

### Use

- This function is used to initiate an abortive release on an already established connection or to reject a connect request. The t\_rcvdis call should follow this call to “clean up” the connection release process. It is used only for connection mode.

---

**Note** – This function cannot be used in applications developed over CLNS.

---

### Description

The parameters are:

*fd* identifies the local transport endpoint of the connection.

*call* specifies information associated with the abortive release. It points to a t\_call structure that contains the following members:

```
struct t_call {
    struct netbuf addr;           /* address          */
    struct netbuf opt;           /* options          */
    struct netbuf udata;         /* user data        */
    int sequence;                /* sequence number */
};
```

where:

*addr* and *opt* fields are ignored.

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

*sequence* is ignored, except when rejecting a request (see below).

The *netbuf* structure is contained in the `tiuser.h` include file. Refer to page 121 for details.

The `t_call` structure must be used when rejecting an incoming connect request. When rejecting a connect request, it must be non-NULL and contain a valid value of *sequence* to identify uniquely the rejected connect indication to the transport provider.

In all other cases, *call* need only be used when data is being sent with the disconnect request. The *addr*, *opt*, and *sequence* fields of the `t_call` structure are ignored. If the user does not wish to send data to the remote user, the value of *call* may be NULL.

Both servers and clients can initiate the connection release.

### **Return Values**

`t_snddis` returns 0 on success. On failure `t_snddis` returns -1, `t_errno` is set to indicate the error, and *errno* might be set.

### **Errors**

On failure, `t_errno` will be set to one of the following:

TBADDATA

The amount of user data specified was not within the bounds allowed by the transport provider. The transport provider's outgoing queue will be flushed, so data may be lost.

TBADF

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

**TBADSEQ**

An invalid sequence number was specified, or a NULL call structure was specified when rejecting a connect request. The transport provider's outgoing queue will be flushed, so data may be lost.

**TLOOK**

An asynchronous event has occurred on this transport endpoint and requires immediate attention.

**TNOTSUPPORT**

This function is not supported by the underlying transport provider.

**TOUTSTATE**

The function was issued in the wrong sequence. The transport provider's outgoing queue may be flushed, so data may be lost.

**TSYSERR**

A system error has occurred during execution of this function, *errno* will be set to the specific error.

`t_sndudata`

Sends a datagram.

## Synopsis

```
#include <tiuser.h>
#include <osi_lib.h>

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

## Use

This function is used in connectionless mode to send a data unit to another transport user. It is used only for connectionless mode.

## Description

The parameters are:

*fd* identifies the local transport endpoint through which data will be sent.

*unitdata* points to a `t_unitdata` structure containing the following members:

```
struct t_unitdata {
    struct netbuf addr;    /* address    */
    struct netbuf opt;    /* options   */
    struct netbuf udata;  /* user data */
};
```

where:

*addr* holds the source address of incoming datagrams and the destination address of outgoing datagrams.

*opt* holds any protocol options on the datagram. In connectionless mode SunLink OSI does not support protocol options so *maxlen* is set to zero in the *opt netbuf* structure.

*udata* specifies the user data to be sent.

The *netbuf* structure is contained in the `tiuser.h` include file. Refer to page 121 for details.

If the *len* field of *udata* is zero, and the sending of zero bytes is not supported by the underlying transport provider, `t_sndudata` returns -1 with `t_errno` set to `TBADDATA`.

The *addr*, *opt*, and *udata* fields must all be allocated with buffers large enough to hold any possible incoming values. If `t_alloc` is used with a `T_ALL` argument to allocate the `t_unitdata` structure the TLI library sets the *maxlen* field of each *netbuf* structure accordingly.

The `T_MORE` flag may be used to indicate that data spread over more than one buffer should be grouped together.

By default, `t_sndudata` operates in synchronous mode and may wait if flow control restrictions prevent the data from being accepted by the local transport provider at the time the call is made. However, if `O_NDELAY` or `O_NONBLOCK` is set (using `t_open` or `fcntl(2)`), `t_sndudata` executes in asynchronous mode and will fail under such conditions.

If `t_sndudata` is issued from an invalid state, or if the amount of data specified in *udata* exceeds the TSDU size as returned in the *tsdu* field of the *info* argument of `t_open` or `t_getinfo`, the provider will generate an `EPROTO` protocol error (see `TSYSERR` below). If the state is invalid, this error may not occur until a subsequent reference is made to the transport endpoint.

The `t_sndudata` call is used differently for non-blocking mode. See Chapter 4, “Non-Blocking Mode” for details.

### **Return Values**

`t_sndudata` returns 0 on successful completion. On failure `t_sndudata` returns -1, `t_errno` is set to indicate the error, and *errno* might be set.

## **Errors**

On failure, `t_errno` is set to one of the following:

TBADDATA

*nbytes* is zero and sending zero bytes is not supported by the transport provider.

TBADF

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

TFLOW

`O_NDELAY` or `O_NONBLOCK` was set, but the flow control mechanism prevented the transport provider from accepting data at this time.

TNOTSUPPORT

This function is not supported by the underlying transport provider.

TSYSERR

A system error has occurred during execution of this function, *errno* will be set to the specific error.

`t_sync`

Synchronizes transport library.

### ***Synopsis***

```
#include <tiuser.h>
#include <osi_lib.h>

int t_sync(
    int fd
);
```

### ***Use***

- This function synchronizes the data structures managed by the transport library with information from the underlying transport provider. In doing so, it can convert a raw file descriptor (obtained using `open(2)`, `dup(2)`, or as a result of a `fork(2)` and `exec(2)`) to an initialized transport endpoint, assuming that file descriptor referenced a transport provider. This function also allows two cooperating processes to synchronize their interaction with a transport provider.

---

**Note** – This function cannot be used in applications developed over CLNS.

---

### ***Description***

The parameters are:

*fd* identifies the transport endpoint.

If a process forks a new process and issues an `exec(2)`, the new process must issue a `t_sync` to build the private library data structure associated with a transport endpoint and to synchronize the data structure with the relevant provider information.

It is important to remember that the transport provider treats all users of a transport endpoint as a single user. If multiple processes are using the same endpoint, they should coordinate their activities so as not to violate the state of the provider. The `t_sync` function returns the current state of the provider to the user, thereby enabling the user to verify the state before taking further

action. This coordination is only valid among cooperating processes; it is possible that a process or an incoming event could change the provider's state after a `t_sync` is issued. If the provider is undergoing a state transition when `t_sync` is called, the function fails.

## Return Values

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

`T_UNBND`

Unbound.

`T_IDLE`

Idle.

`T_OUTCON`

Outgoing connection pending.

`T_INCON`

Incoming connection pending.

`T_DATAXFER`

Data transfer.

## Errors

On failure, `t_errno` is set to one of the following:

`TBADF`

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

`TSTATECHNG`

The transport provider is undergoing a state change.

`TSYSERR`

A system error has occurred during execution of this function, `errno` is set to the specific error.



t\_unbind

Disables a transport endpoint.

### **Synopsis**

```
#include <tiuser.h>
#include <osi_lib.h>

int t_unbind(
    int fd
);
```

### **Use**

This function disables the specified transport endpoint which was previously bound by t\_bind.

### **Description**

The parameters are:

*fd* identifies the transport endpoint to unbind.

On completion of this call, no further data or events destined for this transport endpoint will be accepted by the transport provider.

### **Return Values**

t\_unbind returns 0 on success. On failure t\_unbind returns -1, t\_errno is set to indicate the error, and *errno* might be set.

### **Errors**

On failure, t\_errno is set to one of the following:

TBADF

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

TLOOK

An asynchronous event has occurred on this transport endpoint.

### TOUTSTATE

The function was issued in the wrong sequence.

### TSYSERR

A system error has occurred during execution of this function, *errno* is set to the specific error.

## Address Manipulation Functions

9

<i>getmyclnsnsap</i> —use the local CLNS NSAP.	<i>page 192</i>
<i>getmyconsnsap</i> —use the local CONS NSAP.	<i>page 193</i>
<i>getnamebynsap</i> —convert an NSAP into a hostname.	<i>page 194</i>
<i>getnsapbyname</i> —convert a hostname into an NSAP	<i>page 195</i>
<i>gettselbyname</i> —convert a service name into a TSEL.	<i>page 196</i>
<i>nsap2net</i> —write the NSAP to the netbuf structure.	<i>page 197</i>
<i>tsap2net</i> —write the TSAP to the netbuf structure.	<i>page 198</i>

The SunLink OSI 8.1 User Library contains a set of functions that let you use hostnames instead of NSAPs and TSAPs in application and user programs.

The library uses the file `/etc/net/oclt/host` for hostname to NSAP conversion, and the file `/etc/net/oclt/services` for service name to TSEL conversion. In order to use it, you must:

- Add the directory `/opt/SUNWconn/osinet/lib` to your `LD_LIBRARY_PATH` environment variable.
- Compile setting the options `-I/opt/SUNWconn/osinet/include` and `-L/opt/SUNWconn/osinet/lib -losi`.

This chapter describes the address manipulation functions in alphabetical order.

## getmyclnsnsap—*use the local CLNS NSAP.*

### **Synopsis**

```
#include <tiuser.h>
#include <osi.h>

int getmyclnsnsap(
    int nsap_maxlen,
    char *nsap
);
```

### **Use**

Returns the local CLNS NSAP.

### **Description**

The parameters are:

*nsap* points to a buffer reserved for the local CLNS NSAP.

*nsap\_maxlen* is the maximum allowed size of the NSAP. An NSAP can be up to 20 octets.

### **Return Values**

getmyclnsnsap returns the length of the NSAP in octets on success. On failure getmyclnsnsap returns -1, and *errno* might be set.

---

## getmyconsnsap—*use the local CONS NSAP.*

### **Synopsis**

```
#include <tiuser.h>
#include <osi.h>

int getmyconsnsap(
    int nsap_maxlen,
    char *nsap
);
```

### **Use**

Returns the local CONS NSAP.

### **Description**

The parameters are:

*nsap* points to a buffer reserved for the local CONS NSAP.

*nsap\_maxlen* is the maximum allowed size of the NSAP. An NSAP can be up to 20 octets.

### **Return Values**

getmyconsnsap returns the length of the NSAP in octets on success. On failure getmyconsnsap returns -1, and *errno* might be set.

getnamebynsap—*convert an NSAP into a hostname.*

### **Synopsis**

```
#include <tiuser.h>
#include <osi.h>

int getnamebynsap(
    int hostname_maxlen,
    char *hostname,
    char *nsap
);
```

### **Use**

When supplied with an NSAP, `getnamebynsap` returns the hostname defined in the `/etc/net/oclt/host` file.

### **Description**

The parameters are:

*nsap* points to a buffer containing the NSAP.

*hostname* points to a buffer reserved for the hostname.

*hostname\_maxlen* is size of the buffer reserved for the hostname.

### **Return Values**

`getnamebynsap` returns the length in octets of the hostname on success. On failure `getnamebynsap` returns -1, and *errno* might be set.

---

## getnsapbyname—*convert a hostname into an NSAP*

### **Synopsis**

```
#include <tiuser.h>
#include <osi.h>

int getnsapbyname(
    int nsap_maxlen,
    char *hostname,
    char *nsap
);
```

### **Use**

When supplied with a hostname, `getnsapbyname` returns the NSAP defined in the `/etc/net/oclt/host` file.

### **Description**

The parameters are:

*hostname* points to a buffer containing the hostname.

*nsap* points to a buffer reserved for the NSAP.

*nsap\_maxlen* is the size of the buffer reserved for the NSAP.

### **Return Values**

`getnsapbyname` returns the length of the NSAP in octets on success. On failure `getnsapbyname` returns -1, and *errno* might be set.

gettselbyname—*convert a service name into a TSEL.*

## **Synopsis**

```
#include <tiuser.h>
#include <osi.h>

int gettselbyname(
    int tsel_maxlen,
    char *name,
    char *tsel
);
```

## **Use**

When supplied with a service name, `gettselbyname` returns the TSEL defined in the `/etc/net/oclt/services` file.

## **Description**

The parameters are:

*name* points to a buffer containing the service name.

*tsel* points to a buffer reserved for the TSEL.

*tsel\_maxlen* specifies the size of the buffer reserved for the TSEL. The TSEL can be up to 32 octets.

## **Return Values**

`gettselbyname` returns the length of the NSAP in octets on success. On failure `getnsapbyname` returns -1, and *errno* might be set.



---

`nsap2net`—*write the NSAP to the `netbuf` structure.*

### **Synopsis**

```
#include <tiuser.h>
#include <osi.h>

int nsap2net(
    int net_len,
    char *net,
    char *nsap
);
```

### **Use**

Writes the NSAP to the *netbuf* structure. It can then be used directly by the TLI transport provider. For more information on the *netbuf* structure, refer to “The *netbuf* Structure” on page 121.”

### **Description**

The parameters are:

*net* points to the *netbuf* structure.

*nsap* points to the buffer containing the NSAP.

*net\_len* specifies the size of the *netbuf* structure.

### **Return Values**

`nsap2net` returns the length of the buffer on success. On failure `nsap2net` returns -1, and *errno* may be set.

`tsap2net`—*write the TSAP to the netbuf structure.*

## **Synopsis**

```
#include <tiuser.h>
#include <osi.h>

int tsap2net(
    int net_len,
    char *net,
    char *nsap,
    char *tsel
);
```

## **Use**

Writes the TSAP, composed of the NSAP plus the TSEL, to the *netbuf* structure. This can then be used directly by the TLI transport provider. For more information on the *netbuf* structure, refer to “The netbuf Structure” on page 121.”

## **Description**

The parameters are:

*net* points to the *netbuf* structure.

*nsap* points to the buffer containing the NSAP.

*tsel* points to the buffer containing the TSEL.

*net\_len* specifies the size of the *netbuf* structure.

## **Return Values**

`nsap2net` returns the length of the buffer on success. On failure `nsap2net` returns -1, and *errno* may be set.

# *Index*

---

## **A**

- abortive release, 4, 16, 156
- allocating memory, 19
- asynchronous mode, 5

## **B**

- bind
  - address, 129
  - to endpoint, 129
- blocking mode, 5

## **C**

- call
  - local management, 10
  - reference table, 120
  - summary, 118
- class of transport, 4
- client, 4, 11
- CLNP, 4
- CLNS, 3, 25, 192
- code samples, 43
- compiling, 37
- connection
  - establishment, 4, 11
  - release, 4
  - request, 12

- connection mode

- connection establishment, 11
  - connection release, 16
  - data transfer, 15
  - general, 4, 8
  - phases, 4, 8
  - transport, 4

- connectionless mode, 4

- data transfer, 18
  - general, 4, 5, 17
  - local management, 18

- Connectionless Transport Protocol - See CLNP

- Connection-Oriented Transport Protocol - See CONS

- CONS, 4, 193

- conventions used, xvii

## **D**

- data
  - expedited, 15
  - recovery, 4, 5
  - transfer, 4, 5, 15, 18
- datagram, 5, 14, 18, 176
- datastream, 4
- document set, xv

---

## **E**

- endpoint, 9
- error handling, 6, 19
- errors, 6, 14
- events, 31
- examples, 41
- expedited data, 15

## **F**

- flow control, 16, 30
- freeing memory, 19
- function call
  - error, 6
  - reference, 117

## **G**

- getmyclnsnsap, 192
- getmyconsnsap, 193
- getnamebynsap, 194
- getnsapbyname, 195
- gettselbyname, 196
- guaranteed delivery, 4

## **H**

- handling errors, 6, 19
- hostname, 22
- hostname to NSAP conversion, 22, 191
- hostname to TSEL conversion, 22, 191

## **I**

- IDU, 159
- include file, 38
- incoming event, 14
- independency of protocol, 39
- Interface Data Unit - See IDU

## **L**

- library, 37

- level errors, 6
- structure, 126

- linking, 37
- listening endpoint, 14
- local error, 14
- local management, 18
  - connection mode, 4
  - connectionless mode, 5
  - general, 9
- lost data, 5

## **M**

- Makefile, 38
- memory management, 19
- modes of service, 7

## **N**

- non-blocking mode, 5
- normal data, 15
- NSAP, 22, 25
- nsap2net, 197

## **O**

- operating system errors, 6
- orderly disconnection, 155
- OSI reference model, 2
- outgoing events, 32, 34

## **P**

- phases, 4
- poll, 27, 30
- protocol
  - independency, 39
  - options, 10
- provider, 3

## **Q**

- QOS, 160
- Quality of Service - See QOS

---

## R

retrieve data, 169  
RFC1006, 2, 24

## S

select, 30  
server, 4, 11  
specifications, xvi  
standards referenced, xvi  
state transitions, 6, 31  
synchronous mode, 5

## T

t\_accept, 12, 33, 34, 122, 151  
t\_alloc, 35, 39, 126  
t\_bind, 10, 18, 33, 39, 129  
t\_call, 150  
T\_CHECK, 161  
t\_close, 33, 35, 133, 192, 193  
t\_connect, 12, 29, 33, 35, 39, 135  
T\_DEFAULT, 161  
t\_error, 35, 140  
t\_free, 35, 142  
t\_getinfo, 35, 39, 144, 156  
t\_getstate, 35, 148  
t\_info structure, 156  
t\_listen, 12, 28, 29, 34, 39, 150  
t\_look, 14, 35, 153  
T\_MORE, 15  
T\_NEGOTIATE, 161  
t\_open, 10, 18, 33, 39, 155  
t\_optmgmt, 10, 18, 33  
t\_rcv, 15, 30, 34, 163  
t\_rcvconnect, 12, 29, 34  
t\_rcvdis, 16, 34, 39, 169  
t\_rcvrel, 40  
t\_rcvudata, 18, 30, 34, 172  
t\_rcvuderr, 18, 34, 40, 175  
t\_snd, 15, 30, 33, 178

t\_snddis, 16, 33, 39, 151, 181  
t\_sndrel, 40  
t\_sndudata, 18, 19, 30, 33, 39, 184  
t\_sync, 35, 187  
t\_unbind, 33, 189

## TLI

connection establishment, 11  
connection phase, 11  
data transfer, 18  
library, 37  
local management, 5  
modes of service, 4  
states, 32

TLOOK, 14

## transport

address, 156  
class, 4  
endpoint, 9  
provider states, 6  
service characteristics, 9  
service provider, 3  
service user, 2

Transport Service Access Point - See TSAP

Transport Service Data Unit - See TSDU

TSAP, 22, 23

TSAP address, 9, 129

tsap2net, 198

TSDU, 15, 39  
size, 156

TSEL, 23

## U

user, 2  
user library, 22

