DataGeneral

Customer Documentation

Programming with Transport Layer Interface (TLI) for AViiON® Systems

AViiON®
PRODUCT LINE

Programming with Transport Layer Interface (TLI) for AViiON® Systems

069–000482–01

For the latest enhancements, cautions, documentation changes, and other information on this product, please see the Release Notice (085–series) supplied with the software.

Ordering No. 069–000482–01

Printed in the United States of America
Rev. 01, May 1991

Notice

Programming with Transport Layer (TLI) Interfaces for AViiON® Systems
069–000482–01

Revision History:

Original Release – May 1990
First Revision – May 1991

Effective with:

DG/NETBEUI for AViiON® Systems, Rev. 1.00
DG/UX for AViiON® Systems, Rev. 4.32
NetWare® for AViiON® Systems, Rev. 1.00
OSI/Platform for AViiON® Systems, Rev. 1.00

A vertical bar in the margin of a page indicates substantive technical change from the previous revision. Chapters 2 through 8 of the *Programming with Transport Interfaces for AViiON® Systems* (069–000482–00) were moved to form a new manual, *Programming the NETBIOS Interface for AViiON® Systems* (069–000565–00).

Preface

This manual explains the specifics of the Transport Layer Interface (TLI) Library for the different Distributed Applications Architecture OPEN LAN communication protocols available on the DG/UX™ system. The manual assumes that you are familiar with the TLI interface and the DG/UX™ system programming environment. The TLI Library is part of the DG/UX™ system.

Organization

This manual contains five chapters. The following list gives an overview of what you will find in these chapters:

Chapter 1 Provides an overview of the TLI.

Chapter 2 Describes TLI/NetBIOS Interface.

Chapter 3 Describes the Internet Packet Exchange (IPX) Interface.

Chapter 4 Describes the Sequence Packet Exchange (SPX) Interface.

Chapter 5 Describes the OSI/Platform for AViiON® Systems Interface.

Related Documents

Within this manual, we refer to the following manuals:

Programmer's Reference for the DG/UX™ system, Volumes I and II (093–701041 and 093–701042)

This manual contains the printed versions of the on–line manual pages for the commands and calls relating to programming on the DG/UX™ system.

Programming with TCP/IP on the DG/UX™ System (093–701024)

This manual contains a chapter that provides an overview of communications programming and introduces the TLI for a TCP/IP network.

Setting up and Managing the OSI/Platform for AViiON Systems (093–000738)

This manual explains how to install, configure, manage, and troubleshoot the OSI/Platform for AViiON System.

Additional Manuals

Other documents that may be of interest to you include:

- *AT&T UNIX System V Documentation Set*
- *IBM NetBIOS Application Development Guide* (IBM # 68X2270)

  This manual describes the NetBIOS interface in the DOS environment.

- *UNIX® System V/386 Network Programmer's Guide*, AT&T, 1988
- *UNIX® System V/386 Programmer's Reference Manual*, AT&T, 1988
- *StarGROUP™ Software Application Programmer's Reference Manual*, AT&T, 1989

To order AT&T manual(s), contact

AT&T Customer Information Center
Customer Service Representative
P.O. Box 19901
Indianapolis, Indiana 46219

Reader, Please Note:

Data General manuals use certain symbols and styles of type to indicate different meanings. The Data General symbol and typeface conventions used in this manual are defined in the following list. You should familiarize yourself with these conventions before reading the manual.

This manual also presumes the following meanings for the terms command line, syntax line, and format line. A *command line* is an example of a command string that you should type verbatim; it is preceded by a system prompt and is followed by a delimiter such as the curved arrow symbol for the New Line key. A *syntax line* is a fragment of program code that shows how to use a particular routine. A *format line* shows how to structure a command line or syntax line; it shows the variables that must be supplied and the available options.

Convention	Meaning
boldface	In command lines and format lines: Indicates text (including punctuation) that you type verbatim from your keyboard. All DG/UX™ commands, pathnames, and names of files, directories, and manual pages also use this typeface.
`monospace`	Represents a system response on your screen. Syntax lines also use this font.
italic	In format lines: Represents variables for which you supply values; for example, the names of your directories and files, your username and password, and possible arguments to commands. In text: Indicates a term that is defined in the text.
[*optional*]	In format lines: These brackets surround an optional argument. Don't type the brackets; they only set off what is optional. The brackets are in regular type and should not be confused with the boldface brackets shown below.

[ ]	In format lines: Indicates literal brackets that you should type. These brackets are in boldface type and should not be confused with the regular brackets shown above.
...	In format lines and syntax lines: Means you can repeat the preceding argument as many times as desired.
$ and %	In command lines and other examples: Represent the system command prompt symbols used for the Bourne and C shells, respectively. Note that your system might use different symbols for the command prompts.
#	In command lines and other examples: Represents the system command Superuser prompt symbol.
↵	In command lines and other examples: Represents the New Line key, which is the name of the key used to generate a new line. (Note that on some keyboards this key might be called Enter or Return instead of New Line.) Throughout this manual, a space precedes the New Line symbol; this space is used only to improve readability; you can ignore it.
< >	In command lines and other examples: Angle brackets distinguish a command sequence or a keystroke (such as **<Ctrl–D>** and **<Esc>**) from surrounding text. For example, when you see the **<Ctrl–C>** symbol, hold the Control key down and press the C key on your terminal keyboard. Note that these angle brackets are in regular type and that you do not type them; there are, however, boldface versions of these symbols (described below) that you do type.
<, >, >>	In text, command lines, and other examples: These boldface symbols are redirection operators, used for redirecting input and output. When they appear in boldface type, they are literal characters that you should type.
□	In command lines and other examples: The box represents the cursor, which indicates your current typing position on the screen.

Contacting Data General

Data General wants to assist you in any way it can to help you use its products. Please feel free to contact the company as outlined below.

Manuals

If you require additional manuals, please use the enclosed TIPS order form (United States only) or contact your local Data General sales representative.

Telephone Assistance

If you are unable to solve a problem using any manual you received with your system, free telephone assistance is available with your hardware warranty and with most Data General software service options. If you are within the United States or Canada, contact the Data General Service Center by calling 1–800–DG–HELPS. Lines are open from 8:00 a.m. to 5:00 p.m., your time, Monday through Friday. The center will put you in touch with a member of Data General's telephone assistance staff who can answer your questions.

If you are unable to solve a problem using any manual you received with your system, and you are within the United States or Canada, contact the Data General Service Center by calling 1–800–DG–HELPS for toll–free telephone support. The center will put you in touch with a member of Data General's telephone assistance staff who can answer your questions.

For telephone assistance outside the United States or Canada, ask your Data General sales representative for the appropriate telephone number.

Joining Our Users Group

Please consider joining the largest independent organization of Data General users, the North American Data General Users Group (NADGUG). In addition to making valuable contacts, members receive *FOCUS* monthly magazine, a conference discount, access to the Software Library and Electronic Bulletin Board, an annual Member Directory, Regional and Special Interest Groups, and much more. For more information about membership in the North American Data General Users Group, call 1–800–877–4787 or 1–512–345–5316.

End of Preface

Contents

Chapter 1 – Introduction

Chapter 2 – TLI/NetBIOS Interface

Chapter 3 – SPX Transport Layer Interface

Chapter 4 – IPX Network Layer Interface

Chapter 5 – OSI/Platform Interface

Tables

Table

Figures

Figure

Chapter 1
Introduction

The Transport Layer Interface (TLI) is a user application library developed by AT&T that uses STREAMS mechanisms to access transport–level services in the kernel. The interface provides you, an application programmer, with an easy access to the transport services. The DG/UX™ system provides these optional services for several communication protocols. Figure 1–1 identifies these communication protocols and shows a sample list of their functions.

NOTE: Although this manual describes the Transport Layer Interface for several communication protocols, your release tape contains support for only one of these protocols.

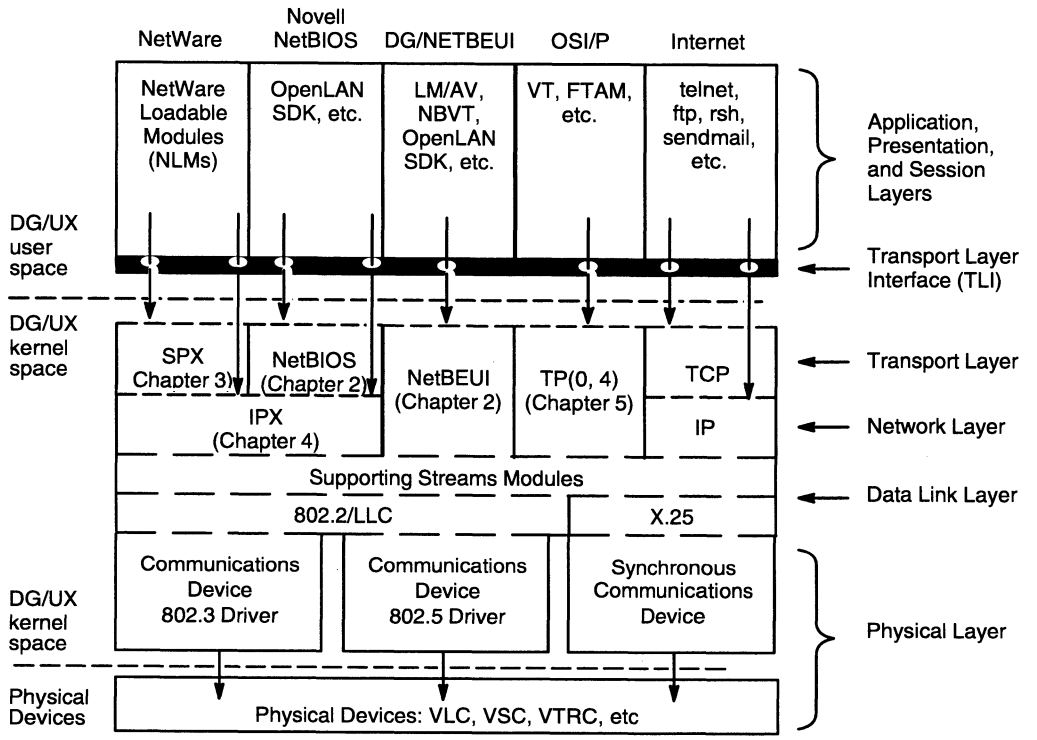


Figure 1–1 Network–level Programming Environment

Many widely–used network architectures conform in some way to the model developed by the International Organization for Standardization (ISO). This model is called the Reference Model for *Open Systems Interconnection (OSI)*. The OSI model consists of seven layers. As shown in Figure 1–1, the transport layer interface (AT&T TLI) is the interface to services equivalent to those define by the transport and network layers of the OSI Model. Using this layered network approach and the transport level interface, your application that may run over the TCP/IP protocol can run with minor modifications over another network protocol (SPX, NetBIOS, etc.).

As previously stated in the Preface, this manual assumes that you are familiar with the TLI interface and the DG/UX™ system programming environment. For a complete description of the TLI calls, please refer to the *Programmer's Reference for the DG/UX system, Volumes I and II* or to the on–line DG/UX™ system man pages.

If you plan to write an application with the TLI that runs over TCP/IP, please refer to the *Programming with TCP/IP on the DG/UX System* manual. It provides an informative explanation of using the TLI. The manual compares the TLI structures and usage with the sockets interface to TCP/IP. If you require additional information on the transport level interface, you may want to obtain a copy of the *AT&T UNIX System V Documentation Set*. See the Preface for additional information on these and other related documentation.

The remainder of this manual explains the DG/UX™ system–specific and protocol–dependent TLI programming information.

Using the TLI on the DG/UX System

Compiling Your Program

For your program to access the transport layer interface at run–time, you must provide a link to the TLI library when you compile your program. For example, use

% **cc user.c –lnsl_s** ↵

The command compiles your program (**user.c**) and includes the TLI Library (**nsl_s**).

Error Handling

The TLI Library returns to your application two types of error codes. The TLI Library returns a TLI error code in the **t_errno** variable. The *Programmer's Reference for the DG/UX system, Volumes I and II* describes the codes. The DG/UX™ system returns system error codes in the **errno** variable. The **/usr/include/sys/errno.h** file lists all of the possible system error codes.

NOTE: You must clear the **errno** and **t_errno** variables after detecting an error and before executing the next TLI call.

The next two paragraphs explain two global error–handling issues. The first issue involves interrupt handling. The second issue involves a TLOOK error on a **t_accept()** call.

In addition, all of the TLI calls are interruptable. You will need to write error handling code that tests for this condition. When the system interrupts a TLI call, the system sets **t_errno** to TSYERR and **errno** to EINTR.

Finally, a TLOOK error on a **t_accept()** call may mean a connect or disconnect is outstanding. That connect or disconnect must be completed before the **t_accept()** will be successful. You must issue either a **t_look()** call or set the **qlen** parameter in a **t_bind()** call to 1. With the **qlen** parameter set to 1, only 1 outstanding connection is possible. Then when a TLOOK error occurs, it is the result of a disconnect.

Using the TLI with Different Protocols

Most of the communications protocols require that you include more than one header file. For example Table 1–1 shows all of the TLI–supported communications protocols and the include files that they require.

Table 1–1 TLI Include Header Files

Protocol Name	Standard Header File	Additional Header Files
Novell NetBIOS	tiuser.h	nb_app.h
DG/NETBEUI	tiuser.h	nb_app.h
OSI/P	tiuser.h	sys/osip/osip.h
Internet	tiuser.h	
SPX	tiuser.h	ipx_app.h spx_app.h
IPX	tiuser.h	ipx_app.h

In addition, each of the protocols use protocol–dependent device names and network addresses. For example in the **t_open()** call, the path variable contains the pathname to a streams transport device driver (protocol–dependent device name). Table 1–2 lists the protocols and the TLI–supported devices.

Table 1–2 Summarizing TLI Device Names

Protocol Name	Connection–oriented Transport Service (COTS)	Connectionless Network Service (CLNS)
Novell NetBIOS	/dev/nbio	/dev/nbdg
DG/NETBEUI	/dev/ntpc	/dev/ntpd
OSI/P	/dev/cots	N/A
TCP/IP	/dev/tcp	/dev/udp
SPX	dev/nspx	N/A
IPX	N/A	/dev/ipx

NOTE: N/A in the table indicates that the protocol does not support the device type.

The addr variable in the t_bind structure of the **t_bind()** call contains a protocol–dependent network address. Table 1–3 summaries the protocol–dependent network address formats.

Table 1–3 Summarizing Protocol–Dependent Network Addressing

Protocol Name	Network Address Format
TLI/NetBIOS	**#define NB_NAME_SIZE 16** **struct nb_addr {** **unsigned char name[NB_NAME_SIZE];**
OSI/P	The network address format contains two variable fields. See Chapter 5.
TCP/IP	See the *Programming with TCP/IP on the DG/UX™ System* manual.
SPX & IPX	**typedef struct ipxAddr_s {** **unsigned char net[4];** **unsigned char node[6];** **unsigned char sock[2];** **}ipxAddr_t;**

The DG/UX system expects addresses to be supplied in a hi→low byte order. Some machines, however, reverse this order. Thus, your program may need to byte–swap a network address. The DG/UX system supplies routines to handle byte–swapping of network addresses and values. The header file **/usr/include/netinet/in.h** defines these routines. Table 1–4 summarizes the routines that handle byte–swapping of network addresses and values.

Table 1–4 Routines for Byte–Swapping a Network Address

Routine	Meaning	What it Does
htonl(val)	host to network long	Convert 32–bit value from host to network byte order.
htons(val)	host to network short	Convert 16–bit value from host to network byte order.
ntohl(val)	network to host long	Convert 32–bit value from network to host byte order.
ntohs(val)	network to host short	Convert 16–bit value from network to host byte order.

For more information on these routines, see *Programming with TCP/IP on the DG/UX™ System* manual and the **byteorder** on–line man page.

Using the TLI Calls

Table 1–5 lists all of the TLI calls. As you can see in the table, protocols (such as TCP/IP) support all of the TLI calls in accordance with the TLI standard; while other protocols require some special considerations.

Table 1–5 TLI Calls

TLI Call	Novell NetBIOS & DG/NETBEUI	OSI/P	Internet (TCP/IP)	SPX	IPX
t_accept	...	...	...	...	N
t_alloc	...	...	...	...	...
t_bind	Δ	...	...	Δ	Δ
t_close	...	...	...	...	Δ
t_connect	...	...	...	...	N
t_error	...	...	...	...	...
t_free	...	...	...	...	...
t_getinfo	...	...	...	...	...
t_getstate	...	...	...	...	...
t_listen	...	...	...	...	N
t_look	...	...	...	...	...
t_open	...	...	...	...	...
t_optmgmt	...	...	...	...	Δ
t_rcv	...	...	...	Δ	N
t_rcvconnect	...	...	...	...	N
t_rcvdis	...	...	...	Δ	N
t_rcvrel	...	N	...	N	N
t_rcvudata	...	N	...	N	Δ
t_rcvuderr	...	N	...	N	...
t_snd	...	...	...	...	N
t_snddis	...	...	...	...	N
t_sndrel	...	N	...	N	N
t_sndudata	...	N	...	N	Δ
t_sync	...	...	...	...	...
t_unbind	...	...	...	...	Δ

NOTES:

... The protocol supports the standard TLI call.
N The protocol does not support the TLI call.
Δ The protocol supports the TLI call with changes.

The remaining chapters in this manual explain the protocol–dependent TLI programming information that require some special considerations.

End of Chapter

Chapter 2
TLI/NetBIOS Interface

The Transport Layer Interface for NetBIOS (TLI/NetBIOS) is a user application library. This interface makes no assumptions about the underlying NetBIOS implementation. The TLI/NetBIOS interface is closely compatible with AT&T's StarGROUP™ NetBIOS Interface.

This chapter explains the TLI/NetBIOS interface which is shared by two communications protocols, Novell NetBIOS and DG/NETBEUI. Each protocol uses its own transport provider stack. Figure 2–1 shows the TLI/NetBIOS interface and the two transport provider stacks. The Internet Packet Exchange (IPX) supports a datagram service.

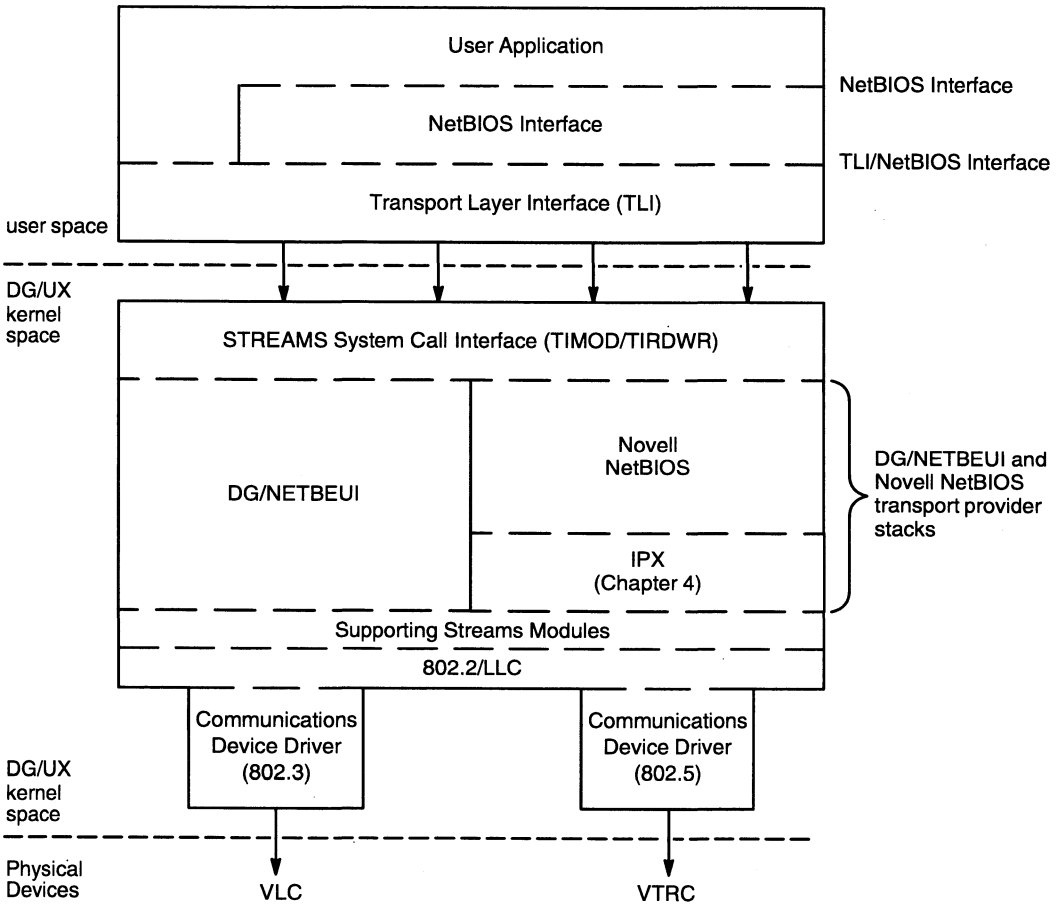


Figure 2–1 Overall TLI/NetBIOS Network–level Programming Environment

Overview of the TLI/NetBIOS Interface

To support access to a NetBIOS service, the TLI/NetBIOS interface requires a deviation from the TLI specification. The TLI specification for **t_bind** specifies that if the requested name is not available, the transport provider binds the fd to another name generated by the provider. In the TLI/NetBIOS interface, if a name is not available, the transport provider returns an error code.

Refer to the *AT&T System V Network Programmer's Guide* for the errors that may occur with these calls.

Data Structures

This section does not define a format for the network address or option control structures. The TLI/NetBIOS interface uses the following constants and structures to define addresses and options. These structures are defined in the file **nb_app.h**.

Name Structure

```
#define NB_NAME_SIZE 16                    /* maximum size of NetBIOS names */

struct nb_addr {
    unsigned char    name[NB_NAME_SIZE];        /* Actual Netbios name */
}
```

In the TLI/NetBIOS interface a name is specified in a TLI t_bind structure, described later in this chapter. The t_bind structure contains a netbuf structure in which the buf pointer points to a NetBIOS name. The netbuf structure also contains a length field. If the length field is 0, a "permanent name" is assigned by the provider. Names less than 16 bytes long are padded on the right with zeros.

Option Management Structure

The following constants and structures are used for option management.

```
#define U_GOS_OPTION 0x01          /* Option not used yet */
#define U_TSDU_OPTION 0x02         /* Option not used yet */
#define U_GROUP_OPTION 0x04        /* Bind to the specified group name */
#define U_BCAST_OPTION 0x08        /* Receive Broadcast Datagrams */
#define U_TK_OPTION 0x10           /* Communication options */

struct TKOPTIONS {
    unsigned int    chksum;        /* enable/disable checksum */
    unsigned int    expedited;     /* enable/disable expedited data */
    unsigned int    extended;      /* enable/disable extended features */
    unsigned int    class;         /* transport classification*/
    unsigned int    acktime;       /* acknowledge fail time in seconds */
    unsigned int    credit;        /* sliding window size */
    unsigned int    tpdusize;      /* 2^n bytes/packet */
}
```

```
struct sl_t_opts {
    unsigned long    options;          /* option from above */
    long             gos_option;       /* not used yet */
    long             tsdu_options      /* not used yet */
    unsigned char    group_option[16]; /* group name string */
    unsigned long    bcast_option;     /* broadcast enable/disable flag */

    struct TKOPTIONS tk_opts;          /* see below */
}

typedef struct sl_t_opts U_OPTIONS;
```

This structure is defined in the **nb_app.h** header file.

The options field is set to one of the defined U_xxx_OPTIONS. If the option U_GROUP_OPTION is requested, the provider will attempt to bind the name in group_option as a group name to the specified end–point. If U_BCAST_OPTION is requested, a value of 0 in the bcast_option field will turn off receipt of Broadcast Datagrams and a 1 will turn it on. Legal values for the tk_opts structure vary between interfaces, but both should respond correctly to the appropriate T_DEFAULT, T_CHECK and T_NEGOTIATE requests, so check the response of every call. These definitions may be found in the AT&T–supplied header files **slanuser.h** or **tiuser.h**.

TLI/NetBIOS Interface Descriptions

This section describes the TLI calls. The descriptions include a format of the call, any default parameter information, and predefined values. The calls are functionally grouped and presented in several subsections. Table 2–1 lists the calls alphabetically and the page where the call description begins.

Table 2–1 TLI/NetBIOS Calls Listed Alphabetically

Function	Page	Description
t_accept()	2-21	Accepts incoming connections.
t_alloc()	2-10	Returns a pointer to the memory allocated.
t_bind()	2-7	Associates a protocol address with a transport end point.
t_close()	2-9	Closes a transport end point.
t_connect()	2-19	Attempts to initiate a connection with a remote user.
t_error()	2-12	Prints a message on the standard error output which describes the last transport error encountered.
t_free()	2-11	Frees memory previously allocated by **t_alloc()**.

(Continues)

Table 2–1 TLI/NetBIOS Calls Listed Alphabetically

Function	Page	Description
t_getinfo()	2-13	Returns the current characteristics of the underlying transport protocol associated with file descriptor.
t_getstate()	2-14	Returns the current state of the provider associated with the transport end point.
t_listen()	2-20	Listens for incoming connect indications.
t_look()	2-15	Returns a value indicating what event has occurred.
t_open()	2-6	Returns transport end point file descriptor (fd) to be used in future TLI function calls.
t_optmgmt()	2-17	Manages options for a transport end point.
t_rcv()	2-24	Receives connection–oriented data on a transport end point.
t_rcvconnect()	2-22	Determines the status of a **t_connect()** function that was previously issued in asynchronous mode.
t_rcvdis()	2-28	Acknowledges rejection or failure of a connection.
t_rcvrel()	2-30	Acknowledges an orderly release request from the remote end point of an established session.
t_rcvudata()	2-33	Receives both directed datagrams and broadcast datagrams.
t_rcvuderr()	2-34	Returns information on datagrams which could not be transmit to the remote end point.
t_snd()	2-25	Sends connection–oriented data on a transport end point.
t_snddis()	2-27	Initiates a disconnect on an established connection or rejects a connection request.
t_sndrel()	2-29	Requests an orderly release of an established session.
t_sndudata()	2-32	Sends NetBIOS datagrams.
t_sync()	2-16	Synchronizes the data structures managed by the transport library with information from the underlying transport provider.
t_unbind()	2-8	Disables a transport end point.

(Concluded)

Local Management Functions

Table 2–2 shows the local management functions. The functions are presented in the order that you use them.

Table 2–2 TLI/NetBIOS Local Management Functions

Function	Page	Description
t_open()	2-6	Returns transport end point file descriptor (fd) to be used in future TLI function calls.
t_bind()	2-7	Associates a protocol address with a transport end point.
t_unbind()	2-8	Disables a transport end point.
t_close()	2-9	Closes a transport end point.
t_alloc()	2-10	Returns a pointer to the memory allocated.
t_free()	2-11	Frees memory previously allocated by **t_alloc()**.
t_error()	2-12	Prints a message on the standard error output which describes the last transport error encountered.
t_getinfo()	2-13	Returns the current characteristics of the underlying transport protocol associated with file descriptor.
t_getstate()	2-14	Returns the current state of the provider associated with the transport end point.
t_look()	2-15	Returns a value indicating what event has occurred.
t_sync()	2-16	Synchronizes the data structures managed by the transport library with information from the underlying transport provider.
t_optmgmt()	2-17	Manages options for a transport end point.

t_open() Establish a Transport End Point.

Usage

#include <tiuser.h>
#include <nb_app.h>

```
int             t_open( path, oflag, info )   /* returns transport end point file
                                                 descriptor (fd) to be used in future TLI function calls. */
char            *path;                        /* path of the multiplexer device, e.g. "/dev/nbio" */
int             oflag;                        /* open flags, the same as in the open system call.*/
struct t_info   *info                         /* structure where information about the end point
                                                 is stored.*/
```

Description

The **t_open()** function creates a transport end point (e.g., a file descriptor) for use in subsequent network interface operations. The path variable is the name of the streams transport device driver. There are two device drivers used for NetBIOS operations, one for connection–oriented service and one for connectionless service. See Table 2–3 for a list of the device drivers.

Table 2–3 NetBIOS Device Drivers

Protocol Name	Connection–oriented Transport Service (COTS)	Connectionless Network Service (CLNS)
Novell NetBIOS	/dev/nbio	/dev/nbdg
DG/NETBEUI	/dev/ntpc	/dev/ntpd

t_bind() Bind an Address to a Transport End Point.

Usage

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

int          t_bind( fd, req, ret ) /* returns binary error code indicating either
                                       success (0) or failure (–1). */
int                  fd             /* The transport end point fd.*/
struct t_bind        *req;          /* The requested address. */
struct t_bind        *ret;          /* The actual address bound. */
```

Description

The **t_bind()** function activates a TLI/NetBIOS transport end point by associating a NetBIOS name with it. If the name is not defined (req → addr.len = 0 or req = NULL), the provider will assign a name. A name can start with any character except an asterisk (052_8 or $2A_{16}$) or a NULL (0). Names starting with company initials such as IBM, AT&T, and DGC are discouraged.

NOTES: Connection requests are received on a session end–point if the qlen is greater than zero. Each unique name may have only one stream bound with qlen greater than zero, but group names may have many streams with qlen greater than zero. The qlen is ignored on datagram devices.

All names are added as unique names. To bind to a group name, you must first bind to a unique name and then make a **t_optmgmt()** call with flags set to T_NEGOTIATE, options set to U_GROUP_OPTION. The group_option must contain the NetBIOS group name to be added.

Similarly, to bind to a name to receive broadcast datagrams, first bind to a unique name. Then make a t_optmgmt() call with flags set to T_NEGOTIATE, options set to U_GROUP_OPTION, and bcast_option set to 1.

You can add unique names if they are not currently used at a remote end–point or as a group name locally or remotely. You can add group names if they are not already used as a unique name.

t_unbind() Disable a Transport End Point.

Usage

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

int     t_unbind( fd );     /* returns a binary error code indicating
                                   success (0) or failure (non–zero). */
int     fd;                 /* The fd of the end_point. */
```

Description

The **t_unbind()** function disables the transport end point that the fd parameter specifies. If this transport end–point is the last end point bound to the name on this node, **t_unbind()** also deletes the name from the NetBIOS name table.

t_close() Close a Transport End Point.

Usage

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

    t_close( fd )                /* returns a binary error code indicating success (0) or
                                                                    failure (non–zero). */
    int         fd;                                                 /* fd to be closed. */
```

Description

The **t_close()** function closes the transport end point that the fd parameter specifies. The transport end point is deleted and any resources associated within the transport provider or the TLI library are released. If this transport end point is the last end point bound to the name on this node, **t_close()** also deletes the name from the NetBIOS name table.

t_alloc() Allocate a Library Structure.

Usage

#include <tiuser.h>
#include <nb_app.h>

```
char    *t_alloc( fd, struct_type, fields )        /*Returns a pointer to the
                                                        memory allocated. */
int     fd;             /* The file descriptor of the transport end–point
                                                        the memory is to be used for. */
int     struct_type;    /* The type of structure to be allocated */
int     fields;         /* Additional fields to be allocated */
```

Description

The **t_alloc()** function dynamically allocates memory other TLI calls will use. The fd parameter specifies the transport end point for the memory. The struct_type field indicates the type of structure to be allocated. Fields indicates which buffers to allocate. This function returns a pointer to the allocated memory or NULL if an error occurs.

t_free() — Free a Library Structure.

Usage

#include <tiuser.h>
#include <nb_app.h>

```
int     t_free( ptr, struct_type ) /* returns a binary error code indicating
                                      success (0) or failure (non–zero). */
char    *ptr;           /* pointer to memory block to be freed. */
int     struct_type;    /* type of structure to be freed */
```

Description

The **t_free()** function frees memory allocated with the **t_alloc()** function. The ptr parameter points to the memory block to be freed and struct_type specifies the type of the structure. This function returns either success or failure.

t_error() — Produce an Error Message.

Usage

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

    void      t_error( errmsg )
    char      *errmsg;          /* user error message to be printed. */
```

Description

The **t_error()** function prints a message on the standard error output which describes the last transport error encountered. The string pointed to by errmsg is printed, followed by the text for the error message that **t_errno** specifies.

t_getinfo() Get Protocol–specific Service Information.

Usage

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

    int             t_getinfo( fd, info )   /* returns a binary error code
                                               indicating success (0) or failure (non–zero). */
    int             fd;                     /* The fd for the end point. */
    struct t_info   *info;                  /* returned information block. */
```

NOTE: Refer to the section on the t_info structure for information on the values that the **t_getinfo()** call returns.

Description

The **t_getinfo()** function returns the current characteristics of the TLI/NetBIOS transport end point. The fd parameter specifies the transport end point and the info structure contains the returned information. This routine returns 0 for success or –1 for failure.

t_getstate() — Get the Current State.

Usage

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

int     t_getstate( fd );          /* returns the current state of the
                                      end point or –1 if an error occurs. */
int     fd;                        /* The fd for the end point. */
```

Description

The **t_getstate()** function returns the current state of the end point that the fd parameter specifies.

t_look() Look at the current event on a transport end point.

Usage

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

int     t_look( fd )        /* Returns a value indicating what event
                               has occurred, 0 if no event has occurred, or –1 if error */
int     fd;                 /* The fd for the end point. */
```

Description

The **t_look()** function returns the current event on the transport end point that the fd parameter specifies. If no event has occurred it returns 0 and if an error occurs it returns –1.

t_sync() Synchronize Transport Library.

Usage

#include <tiuser.h>
#include <nb_app.h>

```
int     t_sync( fd )        /* returns the state of the transport provider on
                                successful completion or –1 */
int     fd;                 /* fd to be synchronized. */
```

Description

The **t_sync()** function synchronizes the data structures in the transport library with those in the transport provider. In doing so, it can convert a raw file descriptor (obtained via **open()**, **dup()**, or as a result of a **fork()** and **exec()** system call) to an initialized transport end point. The fd parameter indicates the file descriptor to be synchronized. This function returns the state of the transport end point or –1 if an error occurs.

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

t_optmgmt() Manage Options for a Transport End Point.

Usage

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

int          t_optmgmt( fd, req, ret )      /* returns a binary error code
                                               indicating success (0) or (–1) failure). */
int          fd;                            /* fd of the end point */
struct t_optmgmt*req;                       /* requested options */
struct t_optmgmt*ret;                       /* returned options */
```

Description

The **t_optmgmt()** function retrieves, verifies, or negotiates protocol options for the transport end point. The structures pointed to by req and ret contain a netbuf structure containing the sl_t_opts structure. The req structure, if present, contains the requested TLI/NetBIOS options. On a nonerror return, the ret structure, if present, will contain the actual options.

NOTE: The sl_t_opts structure contains the tk_opts structure. All fields of the structure are unused except the "acktime" field. Use this field to specify the send time–out period for a connection in seconds. In other words, the field will contain the amount of time that the transport system should wait for an acknowledgment from a remote node after sending data. If the time–out period expires without an acknowledgment, the connection is terminated abnormally. A value of 0 in this field specifies no time–out value, and the transport system should keep trying to send data indefinitely, until an acknowledgment does arrive.

Connection Establishment Functions

Table 2–4 shows the connection establishment functions. The functions are presented in the order that you use them.

Table 2–4 TLI/NetBIOS Connection Establishment Functions

Function	Page	Description
t_connect()	2-19	Attempts to initiate a connection with a remote user.
t_listen()	2-20	Listens for incoming connect indications.
t_accept()	2-21	Accepts incoming connections.
t_rcvconnect()	2-22	Determines the status of a **t_connect()** function that was previously issued in asynchronous mode.

t_connect() Establish a Connection with Another Transport User.

Usage

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

int             t_connect( fd, sndcall, rcvcall)  /* returns a binary argument
                                                     indicating success (0) or failure (–1).*/
int             fd;                               /* The file descriptor. */
struct t_call   *sndcall;                         /* The callers address. */
struct t_call   *rcvcall;                         /* The responding address */
```

Description

The **t_connect()** function attempts to initiate a connection with a remote user. The fd parameter is the file descriptor for a transport end point that is bound to a name. The sndcall address structure contains the NetBIOS name to be called.

NOTE: If the **t_connect()** function is called in synchronous mode, the application will pend until the connection is accepted by the remote application or the connect request times out. Thus, when the function returns successfully, the rcvcall → addr structure contains the name of the remote user with which a connection was established.

If the **t_connect()** function is called in asynchronous mode, the transport provider initiates a connection request with the remote user. The user must poll the file descriptor with the **t_look()** function until a T_CONNECT indicator is received and then must do a **t_rcvconnect()** function call to complete connection set–up. If the T_DISCONNECT indicator is received, the user must do a **t_rcvdis** to abort the session.

t_listen() Listen for a Connect Request.

Usage

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

int             t_listen( fd, call )    /* returns a binary argument
                                           indicating success or failure (–1). */
int             fd;                     /* fd for the end point. */
struct t_call   *call;                  /* caller's addressing information. */
```

Description

The **t_listen()** function listens for incoming connect indications. The fd parameter must be the file descriptor for a transport end point that is bound to a name that has been bound to receive incoming connection requests (i.e. qlen > 0) or the request may pend forever. On return the call → addr.buf field points to the NetBIOS name of the caller. The call → opt and call → udata fields will be 0 on return.

t_accept() Accept a Connect Request.

Usage

#include <tiuser.h>
#include <nb_app.h>

```
t_accept( fd, resfd, call )      /* used to indicate which
int               fd;            fd the incoming connection request came in on. */
int               resfd;         /* fd used to accept the connection. */
struct t_call     *call;         /* callers address. */
```

Description

The **t_accept()** function accepts incoming connections. The fd parameter is the file descriptor for a transport end point on which a listen was returned. The resfd parameter is the file descriptor for the transport end point on which the connection is to be accepted. The t_call structure contains the address and sequence number returned by the **t_listen()** call.

t_rcvconnect() — Receive the Confirmation from a Connect Request.

Usage

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

int             t_rcvconnect( fd, call )        /* returns a binary error code
                                                   indicating success (0) or failure (non–zero). */
int             fd;             /* The fd of the end point. */
struct t_call   *call           /* netbuf structure containing the name of the
                                                   remote NetBIOS end point*/
```

Description

The **t_rcvconnect()** function determines the status of a **t_connect()** function that was previously issued in asynchronous mode. The connection is established on successful completion of this function. The fd parameter specifies the local transport end point for the connection and the call structure is used to return the remote address information. The addr field in the call structure specifies the NetBIOS name of the responding application. The remainder of the fields in the call structure - opt, udata, and sequence - are unused for this call.

Connection–Oriented Data Transfer Functions

Table 2–5 shows the connection–oriented data transfer functions. The functions are presented in the order that you use them.

Table 2–5 TLI/NetBIOS Connection–Oriented Data Transfer_Functions

Function	Page	Description
t_rcv()	2-24	Receives connection–oriented data on a transport end point.
t_snd()	2-25	Sends connection–oriented data on a transport end point.

t_rcv() Receive Data or Expedited Data Sent Over a Connection.

Usage

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

int      t_rcv( fd, buf, nbytes, flags )   /* returns the number of bytes receive
                                                              or -1. */
int      fd;                               /* the fd to receive data on. */
char     *buf;                             /* pointer to the data buffer. */
unsigned nbytes;                           /* length of the buf in bytes */
int      *flags;                           /* returned flags.*/
```

Description

The **t_rcv()** function receives connection–oriented data on a transport end point. The fd parameter is the transport end point, buf is a pointer to the buffer to receive the data, nbytes is the size of the buffer, and flags will contain the returned flags. The **t_rcv()** function will receive a complete message unless the buffer is not large enough to receive the entire message or the connection terminates in the middle of the message.

t_snd() Send Data or Expedited Data Over a Connection.

Usage

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

int       t_snd( fd, buf, nbytes, flags )  /* returns the number of bytes
                                                    sent or –1. */
int       fd;                              /* the fd to send data on. */
char      *buf;                            /* pointer to the data buffer. */
unsigned  nbytes;                          /* length of the buf in bytes */
int       flags;                           /* flags. */
```

Description

The **t_snd()** function sends connection–oriented data on a transport end point. The fd parameter is the transport end point over which the data is sent. The buf parameter points to the data to be sent. The nbytes parameter is the number of bytes to send (maximum of 65535 bytes). The flags parameter contains the send flags. The **t_snd()** call will not support data sends with a 0 byte length.

Connection Release Functions

Table 2–6 shows the connection release functions. The functions are presented in the order that you use them.

Table 2–6 TLI/NetBIOS Connection Release Functions

Function	Page	Description
t_snddis()	2-27	Initiates a disconnect on an established connection or rejects a connection request.
t_rcvdis()	2-28	Acknowledges rejection or failure of a connection.
t_sndrel()	2-29	Requests an orderly release of an established session.
t_rcvrel()	2-30	Acknowledges an orderly release request from the remote end point of an established session.

t_snddis() Send User–initiated Disconnect Request.

Usage

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

    int             t_snddis( fd, call )    /* returns a binary error code
                                               indicating success (0) or failure (non–zero). */
    int             fd;                     /* The fd for the end point.*/
    struct t_call   *call;                  /* call structure containing disconnect
                                               data and sequence number */
```

Description

The **t_snddis()** function initiates a release on an established connection or rejects a connection request. The fd parameter is the file descriptor of the transport end point. If an already established connection is being released, the call pointer should be NULL. If the **t_snddis()** function is rejecting an incoming connect request the sequence field in the call structure must contain the sequence number of the connection to be rejected, as returned from the **t_listen()** call.

NOTE: Since the **t_snddis()** function clears all queue buffers, it is possible that data that has not yet been sent or received may be lost when the call is made. There is no way of knowing if this happens.

t_rcvdis() Retrieve Information from Disconnect.

Usage

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

    int                 t_rcvdis( fd, discon )  /* Returns a binary error code
                                                   indicating success (0) or failure (non–zero). */
    int                 fd;                     /* The fd for the end point. */
    struct t_discon     *discon;                /* The disconnect structure. */
```

NOTE: The udata field in the discon structure is 0.

Description

The **t_rcvdis()** function acknowledges rejection or failure of a connection. The fd parameter is the file descriptor of the transport end point. When **t_rcvdis()** returns, the discon structure contains a reason code. Additionally, if a release occurs on a connection on which a **t_listen()** has been returned but which has not yet been accepted by **t_accept**, then the discon sequence field contains the sequence number (which was returned in the **t_listen()**) of the connection.

Since the **t_snddis()** function clears all queue buffers, it is possible that data which has not yet been sent or received may be lost when the call is made. There is no way of knowing if this happens.

t_sndrel() Initiate an Orderly Release.

Usage

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

    int         t_sndrel( fd )      /* Returns a binary error code indicates
                                       success (0) or failure (non–zero). */
    int         fd;                 /* The fd for the end point. */
```

Description

The **t_sndrel()** function is supported only on devices of type T_COTS_ORD (see **t_getinfo()** or **t_open()**). Use **t_sndrel()** to request an orderly release of an established session. The fd parameter specifies the end point. The connection is terminated after successful completion of a subsequent **t_rcvrel()**, **t_rcvdis()**, or **t_snddis()** function.

t_rcvrel() Acknowledge Receipt of an Orderly Release Indication.

Usage

#include <tiuser.h>
#include <nb_app.h>

```
int     t_rcvrel( fd )      /* Returns a binary error code indicating
                                   success (0) or failure (non–zero). */
int     fd;                 /* The fd for the end point. */
```

Description

The **t_rcvrel()** function acknowledges an orderly release request from the remote end point of an established session. If a **t_rcvrel()** completes successfully and a **t_sndrel()** has already been done on the end point, the session is closed. Otherwise, the session will not be closed until a **t_sndrel()** or **t_snddis()** is called.

Datagram Service Functions

Table 2–7 shows the datagram service functions. The functions are presented in the order that you use them.

Table 2–7 TLI/NetBIOS Datagram Service Functions

Function	Page	Description
t_sndudata()	2-32	Sends NetBIOS datagrams.
t_rcvudata()	2-33	Receives both directed datagrams and broadcast datagrams.
t_rcvuderr()	2-34	Returns information on datagrams which could not be transmit to the remote end point.

t_sndudata() — Send a Data Unit.

Usage

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

    t_sndudata( fd, unitdata )                  /* returns bytes transmitted. */
    int                 fd;                     /* fd of the transport end point */
    struct t_unitdata *unitdata;                /* structure containing address,
                                                        options and data to be sent. */
```

Description

The **t_sndudata()** function sends NetBIOS datagrams. The fd parameter specifies the transport end point to be used to send the datagram. In the unitdata structure, the addr value specifies the NetBIOS name of the destination, udata specifies the datagram to be sent, and opt should be 0. To send a broadcast datagram, the first character of the NetBIOS name should be an asterisk (*).

t_rcvudata() Receive a Data Unit.

Usage

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

    t_rcvudata( fd, unitdata, flags ) /* returns the size of the datagram received. */
    int                 fd;                 /* fd of the transport end point*/
    struct t_unitdata *unitdata;            /* structure containing additional
                                                    options and data to be sent. */
    int                 *flags;             /* flags returned. */
```

Description

The **t_rcvudata()** function receives both directed datagrams and broadcast datagrams. The fd parameter specifies the transport end point where datagrams will be received. The unitdata structure has pointers to where the remote address, options, and datagram should be stored. The udata.maxlen field should be set to the size of the receive buffer. On return, the T_MORE bit may be set in the flags word to indicate that the buffer was not large enough for the received datagram.

NOTE: Broadcast Datagrams are received only on those datagram streams that have requested broadcasts through the use of the option management call.

t_rcvuderr() Receive a Unit Data Error Indication.

Usage

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

    int             t_rcvuderr( fd, uderr )     /* Returns a binary error code
                                indicating success (0) or failure (non–zero). */
    int             fd;                         /* The fd for the end point. */
    struct t_uderr  *uderr;                     /* Failed destination information */
```

Description

The **t_rcvuderr()** function returns information on datagrams that could not be transmitted to the remote end point. A provider–dependent error code is returned in the error field of the **t_uderr** structure.

End of Chapter

Chapter 3
SPX Transport Layer Interface

The Sequence Packet Exchange (SPX) is a connection–based, reliable, sequenced–transport protocol. In the DG/UX environment, SPX is accessed through the DG/UX Transport Layer Interface (TLI). SPX under the TLI reliably delivers a series of data units in sequence. Figure 3–1 shows the TLI interface and associated transport provider stack.

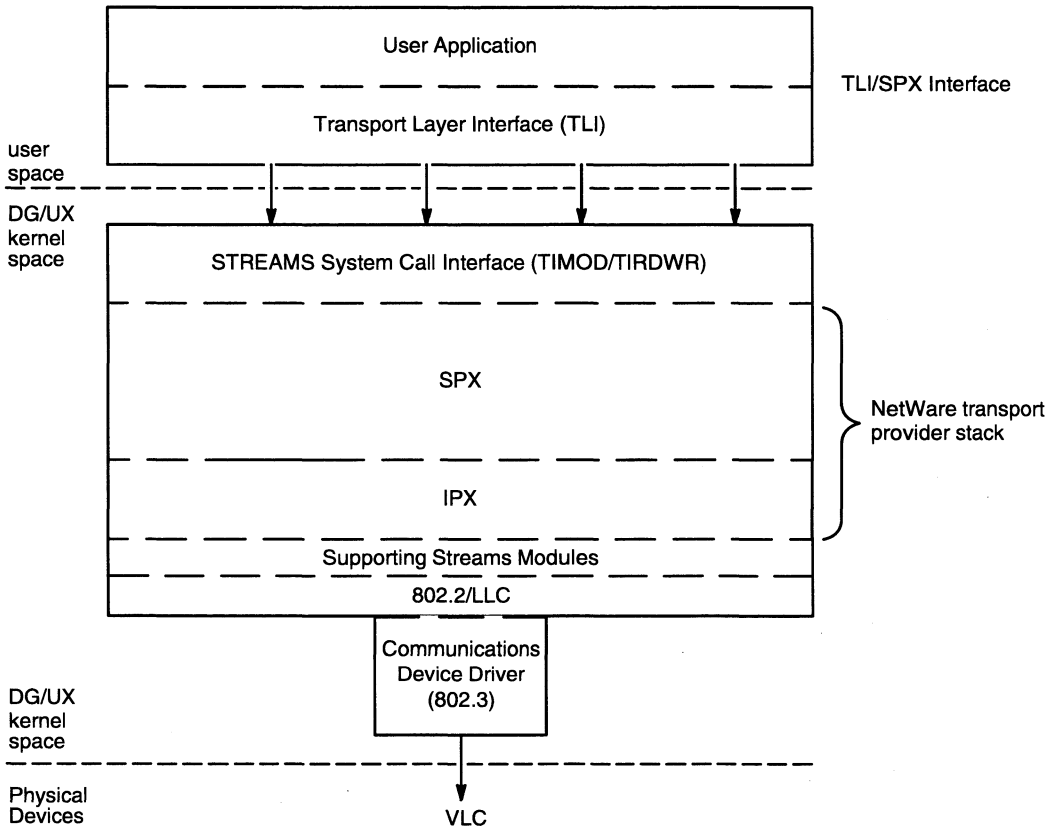


Figure 3–1 TLI / SPX Programming Interface

Because it is a connection–based service, SPX will notify the user if any errors occur during data transmission. Upon encountering a data transmission error, SPX will retry a given number of times before closing the connection and notifying the connection user. SPX will also notify the user if a disconnection indication is received from the remote connection end point.

SPX functions in both synchronous and asynchronous modes. The choice of modes is strictly up to the application developer.

The user interface for SPX is the AT&T Transport Interface Library. Refer to the *AT&T UNIX V Documentation Set* for a description of the order and use of the SPX calls. This chapter provides the details of the peculiarities of the SPX calls. Any TLI calls not mentioned in this chapter functions with the SPX driver as specified in the *AT&T UNIX System V Network Programmer's Guide*.

Procedures

- The procedure for synchronizing a SPX Server program is
  1. Open **/dev/nspx** using **t_open()** getting file descriptor fd.
  2. Bind using **t_bind()** to a well–known socket number on which connection requests will arrive.
  3. Open **/dev/nspx** using **t_open()** again getting fd2.
  4. Bind fd2 to a dynamic socket number using **t_bind()**.
  5. Post a listen using **t_listen()**.
  6. Upon receipt of a connection request, create a child process.
  7. (parent) close fd2; then go to the third step.
  8. (child) Issue a **t_accept()** to accept the connection request (**t_accept(***fd***, *fd2*, *res*)**).
  9. (child) Use **t_snd()** or **t_rcv()** to send or receive data on fd2.
  10. (child) Listen for or send a disconnection indication using **t_rcvdis()** or **t_snddis()**.
  11. (child on exiting) **t_unbind()** fd2.
  12. (child on exiting) **t_close()** fd2.
  13. (parent on exiting) **t_unbind()** fd.
  14. (parent on exiting) **t_close()** fd.
- The procedure for synchronizing a SPX Client program is
  1. Open **/dev/nspx** using **t_open()** fd.
  2. Obtain the address of the server you wish to connect to. The method for obtaining the address is up to the SPX user. The most common method is the use of a "yellow pages" file of the server names and addresses.
  3. Bind using **t_bind()** to a specific or dynamic socket.
  4. Send a **t_connect()** request to the desired server.
  5. Use **t_snd()** or **t_rcv()** to send or receive data on fd.
  6. Listen for or send a disconnection indication using **t_rcvdis()** or **t_snddis()**.
  7. Use **t_unbind()** to unbind the original file descriptor (step 1).
  8. Use **t_close()** to close the original file descriptor (step 1).

SPX Calls

This section describes the SPX calls in alphabetical order.

t_accept() Accept a Connect Request.

This call is issued by the passive user to accept a particular connect request after a connect indication has been received.

Usage

#include <tiuser.h>
#include <ipx_app.h>
#include <spx_app.h>

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

Description

This call works as specified in the *AT&T UNIX System V Network Programmers Guide*.

For server applications (applications that wait for incoming connection requests), we recommend that the server application programmer use one file descriptor and socket to listen for incoming connection requests, and another separate file descriptor and socket to accept connections. Usually a listen file descriptor is used to open the SPX driver and bind to a well–known socket (fd). Upon receiving a connection request, the SPX user opens another file descriptor (resfd), binds to a dynamic socket, and issues a **t_accept()**.

A connection request can be accepted on the same fd as the listen fd (fd=resfd), but we don't recommend it as it involves further multiplexing and more complicated state transition handling for the SPX user. Only a single connection request can be accepted if file descriptors are equal; any further connection requests to the local transport end point will be dropped by the SPX driver.

The DG/UX SPX application must not delay more than the time–out of the client between a **t_listen()** and a **t_accept()**. If the connection request is originating from a DOS client, the client may time out before the DG/UX SPX application does a **t_accept()** to acknowledge the connection request.

If for some reason the **t_accept()** call fails, the SPX driver will mark as free the outstanding connection request to which the **t_accept()** was replying. The SPX user cannot retry the **t_accept()** using the same sequence number as the **t_accept()** that failed.

After issuing a **t_accept()**, the SPX driver will "ping" the connection. The SPX driver will calculate the round trip time to the remote transport end point to facilitate the transmission retries of **t_snd()**.

Errors

The possible errors returned from **t_accept()** are

t_errno	errno	
TOUTSTATE:		The local transport end point or the accepting stream (specified by the accepting fd) is not in the appropriate state for a **t_accept()**.
TSYSERR:	ENXIO:	The socket associated with this connection was not found in the SPX/IPX socket table. This is an SPX driver error.
	ENODEV:	The queue specified to accept the connection (via the accept fd) was not found in the SPX driver's internal structures.
	EBADMSG:	The TPI primitive header (T_CONN_RES) was too small.
TBADSEQ:		The connection request specified by the sequence number has already been answered.

State

The state after a successful connection establishment is T_DATAXFER for both the client and server. An unsuccessful **t_accept()** will leave the state T_BND.

Example

```
int spxFd2;
ipxAddr_t addressToBind;
struct t_bind spxBindInfo;
struct t_info spxInfo;

if ((spxFd2=t_open( spxDevice, O_RDWR, &spxInfo))<0) {
        t_error(" t_open failed ");
        ..
        ..
}

/* we want to bind to a dynamic socket */
addressToBind.sock[0] = 0;
addressToBind.sock[1] = 0;

/* this end point will not receive connection requests*/
spxBindInfo.qlen = 0;

spxBindInfo.maxlen = sizeof(ipxAddr_t);
spxBindInfo.len = sizeof(ipxAddr_t);
spxBindInfo.buf = (char *)&addressToBind;
```

```
if (t_bind(spxFd2, &spxBindInfo, &spxBindInfo)<0) {
        t_error(" t_bind failed ");
        ..
        ..
}

/* spxFd is the file descriptor representing the stream that the
connection request arrived on. The call structure is also the same
call structure that was returned from the t_listen when the
connection request arrived */

if (t_accept(spxFd, spxFd2, &call)<0) {
        t_error(" t_accept failed ");
        ..
        ..
}
```

t_bind() Bind an Address to a Transport End Point.

This call associates a protocol address with a given transport end point, thereby activating the end point. It also directs the transport provider to begin accepting incoming packets.

Usage

```
#include <tiuser.h>
#include <ipx_app.h>
#include <spx_app.h>

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

Description

This call works as specified in the *AT&T UNIX System V Network Programmers Guide* except for the following:

The **t_bind()** call allows an application to bind to an socket number, which is either dynamic or specific. SPX keeps track of which socket number (dynamic or specific) is bound to which transport end point.

A dynamic socket number is an unused (not bound to any other transport end point) socket number returned by the SPX driver. The dynamic socket is guaranteed to be a unique unused number on the local DG/UX host. A dynamic socket is a random value between 0x4000 and 0x5000, inclusive.

A specific socket number is the socket number requested in the req **t_bind()** structure. If it is unused, it is granted and returned in the ret **t_bind()** structure in the range of 1 to FFFFh.

The **t_bind()** call requires that a pointer to an ipxAddr_t type structure be passed in the req → addr.buf field. The net and node numbers don't need to be filled in the ipxAddr_t structure, but the socket number (in hi–low order) must be in the socket field. The SPX driver will look at the socket field in the ipxAddr_t structure for the SPX user's desired socket number. If the socket number (desired or allocated) is not currently being used by another SPX user, the SPX driver will return the socket number in the socket field of the ipxAddr_t structure of the ret → addr.buf field. The SPX driver will also return the local net and local node along with the socket number, all in hi–low order in the ret → addr.buf field.

```
typedef struct ipxAddr_s {
    unsigned char net[4];
    unsigned char node[6];
    unsigned char sock[2];
} ipxAddr_t;
```

If an SPX user passes zeros in the ipxAddr_t socket field the SPX driver will attempt to allocate a dynamic socket number.

The SPX user can pass NULL instead of req and ret. The SPX driver will assume that the SPX user has requested a dynamic socket number, and the SPX driver will try to allocate and return a dynamic socket number.

Only one process can bind to a given socket number at a time. If the user tries to bind to a socket that has already been bound to, an error will result and the bind will fail.

The req → qlen field in the **t_bind()** structure indicates to SPX the total number of outstanding connection requests allowed on this transport end point. An outstanding connection request is a connection request that has arrived and has been delivered to the UNIX application, but the application has not yet responded with a connection request acknowledge (**t_accept()**) or connection request reject (**t_snddis()**).

SPX will allow up to SPX_MAX_LISTENS_PER_SOCKET outstanding connection requests per transport end point. This parameter is currently set to 10. If the UNIX application requests more than 10 only 10 will be given.

If a value greater than 1 is specified in the qlen field during a **t_bind()**, a connection request can arrive from a remote transport end point, making the **t_listen()** unblock. If another connection request arrives between the time the **t_listen()** unblocks and the **t_accept()** is issued, the **t_accept()** will fail, saying an event has occurred. You will not be able to accept the connection requests until all pending connection requests have been retrieved off the stream head using **t_listen()**. A **t_bind()** with qlen=1 should be issued to avoid this problem.

Services written to run over SPX generally have well–known socket numbers associated with them. By using well–known socket numbers, SPX users can be sure that their server and client application types match. Another method to coordinate servers and clients is to use SAPs (Service Advertising Protocol). Contact Novell for more information on well–known socket numbers and SAPs.

Errors

The possible errors returned from **t_bind()** are:

t_errno	**errno**	
TSYSERR:	ENOSR:	No message buffers were available to acknowledge the bind request.
TOUTSTATE:		This transport end point is in a state that invalidates a **t_bind()** request.

TNOADDR:	No unused dynamic socket numbers exist. The SPX user should try again later.
TBADADDR:	The address passed down was not the same size of an ipxAddr_t, or the size of the address was not zero (NULL bind pointer).
TACCES:	The socket number requested was in use.

State

After a successful bind the state is T_IDLE. After an unsuccessful bind, the state is T_BND.

Example

```
#define SOCKET_TO_BIND_HIGH 0x45
#define SOCKET_TO_BIND_LOW 0x00
struct t_bind spxBindInfo;
ipxAddr_t addressToBind;

/* we want to bind to the specific socket 0x4500. The Spx Driver
will fill in the other fields in the addressToBind structure with
the local net and node */

addressToBind.sock[0] = SOCKET_TO_BIND_HIGH;
addressToBind.sock[1] = SOCKET_TO_BIND_LOW;

/* we want one connection request at a time */
spxBindInfo.qlen = 1;
spxBindInfo.maxlen = sizeof(ipxAddr_t);
spxBindInfo.len = sizeof(ipxAddr_t);
spxBindInfo.buf = (char *)&addressToBind;

/* passing the address of spxBindInfo in both the request and
return fields will cause the local information to be read and
written to the addressToBind structure */

if (t_bind(spxFd, &spxBindInfo, &spxBindInfo)<0) {
        t_error(" t_bind failed ");
        ..
        ..
}
```

t_close() Close a Transport End Point.

This call informs the transport provider that the user is finished with the transport end point, and frees any local resources associated with that end point.

Usage

#include <tiuser.h>
#include <ipx_app.h>
#include <spx_app.h>

int t_close(*fd***)**
int *fd***;**

Description

This call works as specified in the *AT&T UNIX System V Network Programmers Guide*.

A **t_close()** call will terminate and release a connection. The **t_close()** function should be called only after the connection has been broken. The SPX driver supports only a disorderly release using **t_snddis()** or **t_rcvdis()**. The **t_close()** call will not cause a terminate connection packet to be sent.

The **t_close()** call will dismantle the stream and release all buffers and messages associated with the transport end point. Upon **t_close()** execution, any data arriving or any data queued for transmission or reception will be dropped.

A transport end point opened with **t_open()** should be closed with **t_close()** to facilitate TLI/SPX functions. Although **t_close()** calls **t_unbind()**, be sure to issue a **t_unbind()** before issuing a **t_close()**.

Errors

There are no errors returned from the SPX driver.

State

State is not applicable after a **t_close()**.

Example

```
t_close (fd);
```

t_connect() Establish a Connection With Another Transport User.

This call requests a connection to the transport user at a specified destination and waits for the remote user's response. This call may be executed in either synchronous or asynchronous mode. In synchronous mode, the call waits for the remote user's response before returning control to the local user. In asynchronous mode, the call initiates connection establishment but returns control to the local user before a response arrives.

Usage

#include <tiuser.h>
#include <ipx_app.h>
#include <spx_app.h>

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

Description

This call works as specified in the *AT&T UNIX System V Network Programmers Guide*.

The **t_connect()** call sends an SPX connection request to the SPX address specified in sndcall → addr. The sndcall → addr should point to an ipxAddr_t structure. The network, node, and socket fields must be filled in with the remote transport end points address in high–low order. The method of obtaining the remote transport end points address is an implementation decision of the SPX user. All information passed in the ipxAddr_t structure must be in high–low byte order; on 80x86 machines byte swapping also needs to be done. The netbuf structure elements opt, udata, and sequence are not used and are ignored by the SPX driver.

If the rcvcall structure is passed to **t_connect()**, the remote transport end points address will be returned in the rcvcall → addr structure as an ipxAddr_t type structure. The remote transport end point's connection ID and window size (allocationNumber) will be passed back as two short integers (2 bytes = short integer) structure in the rcvcall → opt field. If a rcvcall structure is passed, the maxlen, len, and buf variables must be set appropriately to receive these structures.

```
typedef struct SPX_OPTS_s {
   unsigned char connectionId[2];
   unsigned char allocationNumber[2];
} SPX_OPTS;
```

Errors

The SPX driver will try a given number of times to connect with the remote transport end point. After trying a given number of times without receiving an acknowledgment, the SPX driver will generate a disconnect indication with reason set to TLI_SPX_CONNECTION_FAILED (refer to **t_rcvdis()**). If this error occurs, the state of the stream will be set to T_IDLE. Other errors are as documented in the *AT&T UNIX System V Network Programmers Guide*.

State_

The state after a successful connection establishment is T_DATAXFER for both the client and server. The state after an unsuccessful connection establishment is T_BND.

Example

```
struct t_call connectionRequest;
SPX_OPTS remoteServersOptions;
unsigned char remoteServerName[MAX_REMOTE_NAME_LENGTH];
ipxAddr_t remoteServerAddress;
int spxFd;
struct t_info spxInfo;
struct t_bind spxBindInfo;
ipxAddr_t addressToBind;
char *spxDevice = "/dev/nspx";

This routine will find which remote server the user wants to
connect to and place the name in remoteServerName. */

if (GetServerName(remoteServerName)<0) {
        printf(" could not get remote host name \n");
        ..
        ..
}

/* This next routine will take the server name and look in a file
to find the ipx address associated with that server. GetIpxAddress
will fill in remoteServerAddress with the remote servers address
in high-low byte order. */

if(GetIpxAddress(remoteServerName,&remoteServerAddress)<0) {
        printf(" could not get remote server address \n");
        ..
        ..
}

if ((spxFd=t_open( spxDevice, O_RDWR, &spxInfo))<0) {
        t_error(" t_open failed ");
        ..
        ..
}
```

```
/* we want to bind to a dynamic socket */
addressToBind.sock[0] = 0;
addressToBind.sock[1] = 0;

/* this end point will not receive connection requests*/
spxBindInfo.qlen = 0;

spxBindInfo.maxlen = sizeof(ipxAddr_t);
spxBindInfo.len = sizeof(ipxAddr_t);
spxBindInfo.buf = (char *)&addressToBind;

if (t_bind(spxFd, &spxBindInfo, &spxBindInfo)<0) {
        t_error(" t_bind failed ");
        ..
        ..
}

connectionRequest.addr.maxlen = sizeof(ipxAddr_t);
connectionRequest.addr.len = sizeof(ipxAddr_t);
connectionRequest.addr.buf = (char *)&remoteServerAddress;

/* Upon successful return remoteServerOptions will have the
connection identification number and allocation number of the
server. */

connectionRequest.opt.maxlen = sizeof(SPX_OPTS);
connectionRequest.opt.len = sizeof(SPX_OPTS);
connectionRequest.opt.buf = (char *)&remoteServerOptions;

connectionRequest.udata.maxlen = 0;
connectionRequest.udata.len = 0;
connectionRequest.udata.buf = (char *)NULL;

/* We pass the address of connectionRequest for both the request
and return fields so that the connectionRequest.opt
(remoteServerOptions) field will be filled in with the servers
information. */

if ( t_connect(spxFd, &connectionRequest,
               &connectionRequest) <0 ) {
        t_error(" t_connect failed ");
        ..
        ..
}
/* A connection has been established with the server.*/
```

t_listen() Listen for a Connect Request.

This call enables the passive transport user to receive indications of connect requests from other transport users.

Usage

#include <tiuser.h>
#include <ipx_app.h>
#include <spx_app.h>

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

Description

This call works as specified in the *AT&T UNIX System V Network Programmers Guide*.

The **t_listen()** will retrieve any connection requests residing on the stream head. The **t_listen()** call can function synchronously or asynchronously. When the call functions synchronously, it blocks until a connection request comes in. When the call functions asynchronously, it checks for a connection request and returns failure if there is not one.

SPX will insure that each connection indication is unique by dropping any duplicate connection requests. A duplicate request is a request that came from the same network, node, socket, and connectionId as a previous request.

Once **t_listen()** returns, the SPX user can accept the connection request by using **t_accept()**, or the user can refuse a connection request by issuing a **t_snddis()**. The sequence number in the call structure used for the **t_snddis()** and in the **t_listen()** call structure must be the same.

Since SPX doesn't have a "connection–request–nak" packet the SPX driver will not send a packet to deny the connection request. (This differs from the DOS TLI/SPX library in that DOS will send a terminate connection indication if the application issues a **t_snddis()** after a **t_listen()** return.) The client machine issuing the connection request will time out after trying a period of time.

If **t_listen()** returns successfully, call → addr will point to an ipxAddr_t structure that contains the net, node, and socket of the remote transport end point requesting the connection. The net, node, and socket will be in high–low byte order. The call → opt will point to an SPX_OPTS structure that contains the remote transport

end point's connectionId and allocation number. (The remote transport end points connection id and allocation number were separated from the ipxAddr_t structure to maintain compatibility between DOS and OS/2.) The call → udata will not contain anything.

Take special note of the qlen field in a **t_bind()** request.

Errors

The errors are specified in the *AT&T UNIX System V Network Programmers Guide*.

State

This call can be issued only from the T_BND state.

Example

```
struct t_call call;
SPX_OPTS spxOptions;
ipxAddr remoteAddress;

/* set up call structure for t_listen call */
call.addr.maxlen = sizeof(ipxAddr_t);
call.addr.len = sizeof(ipxAddr_t);
call.addr.buf = (char *)&remoteAddress;

call.opt.maxlen = sizeof(SPX_OPTS);
call.opt.len = sizeof(SPX_OPTS);
call.opt.buf = (char *)&spxOptions;

call.udata.maxlen = 0;
call.udata.len = 0;
call.udata.buf = (char *)NULL;

/* Since we are synchronous this call will block until a
connection request comes in. Upon returning the call.addr and
call.opt fields will contain the remote address, and connection
and allocation number of the remote end point.

If we were in asynchronous mode the t_listen call will return fail
if no connection requests have arrived, or success if one has
arrived. */

if (t_listen(spxfd, &call)<0) {
        t_error(" t_listen failed ");
        ..
        ..
}
```

t_open() Establish a Transport End Point.

This call creates a transport end point and returns protocol–specific information associated with that end point. It also returns a file descriptor that serves as the local identifier of the end point.

Usage

#include <tiuser.h>
#include <ipx_app.h>
#include <spx_app.h>

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

Description

This call works as specified in the *AT&T UNIX System V Network Programmers Guide*. SPX may be used either synchronously or asynchronously.

The **t_open()** call returns a file descriptor and an information structure (of type **t_info()**) upon the successful return of an open. This information structure contains pertinent information about the SPX driver. The following is a brief summary of the data returned in the **t_info()** structure:

1. The maximum transport service data unit (TSDU) is 534 bytes.
2. No expedited TSDU is supported.
3. There is no limit to the amount of data sent during a session.
4. No data is transmitted with a disconnect request.
5. The address size is 12 bytes:

   Node is 4 bytes

   Network is 6 bytes

   Socket is 2 bytes
6. 4 bytes of options data are allowed.
7. The maximum transport interface data unit is 534 bytes.
8. The service type is always T_COTS. SPX is a connection–oriented service with a disorderly release.

The path and name of the cloneable SPX device is **/dev/nspx**.

Errors

DG/UX Streams necessitates that a daemon run in the background to build the protocol stack before SPX can be used. This daemon links IPX to the Ethernet driver and then links SPX to IPX. If this daemon has not been run, all attempts to open SPX will fail. Make sure your daemon is running by executing "startnwd" or by bringing up NetWare® for AViiON systems using "SCONSOLE."

If no more minor device numbers are available, the open will fail. If that number has been reached, any attempt to open SPX will fail.

If there are no STREAMS resources available, the open will also fail.

Since none of the above three errors return the same values to **t_open()** there is no way to distinguish which was the true cause of the error. Usually the problem is that the daemon hasn't been run.

State

t_open() will change the state of the service connection to T_UNBND (unbound).

Example

```
char *spxDevice = "/dev/nspx";
struct t_info spxInfo;
int spxFd;

if ((spxFd=t_open( spxDevice, O_RDWR, &spxInfo))<0) {
        t_error(" t_open failed ");
        ..
        ..
}
```

t_optmgmt() Option Management.

This call enables the user to get or negotiate protocol options with the transport provider.

Usage

#include <tiuser.h>
#include <ipx_app.h>
#include <spx_app.h>

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

Description

This call works as specified in the *AT&T UNIX System V Network Programmers Guide*.

This call enables the SPX user to set the maximum number of retries when the SPX driver tries to reliably deliver data to the opposite transport end point. req → opt.buf and ret → opt.buf point to the following structure:

```
struct spxOptMgmt_s {
   unsigned char spxo_max_retries;
   unsigned char spxo_watchdog_flag;
   unsigned int spxo_min_retry_delay;
} spxOptMgmt;
```

The spxo_watchdog_flag is not supported in the DG/UX SPX driver. This flag was intended to decrease network traffic generated by the watchdog in the DOS and OS/2 SPX drivers. But the DG/UX SPX driver's version of Watchdog (WatchEmu) is used to monitor data connections without generating excessive traffic; WatchEmu only sends when it has detected a period of inactivity.

The spxo_min_retry_delay is not supported.

The value in spxo_max_retries will become the new maximum number of retries unless its value exceeds the maximum retry value of 5.

The flags T_NEGOTIATE, T_CHECK, and T_DEFAULT are all supported.

Errors

The possible t_errno errors returned from **t_bind()** are:

TOUTSTATE:	This request was issued in some state other than T_IDLE.
TBADOPT:	The size of the structure spxOptMgmt was less than sizeof(spxOptMgmt) bytes, or there has been an internal TLI/SPX error.
TBADFLAG:	The flag specified is invalid.

State

Not applicable.

Example

```
/* this will set the maximum retry count for a connect request and
a data unit delivery. */

#define SPX_MIN_RETRIES 10
SPX_OPTMGMT spxOptionSet;
struct t_optmgmt optmgmt;

spxOptionSet.spxo_retry_count = SPX_MIN_RETRIES;

optmgmt.opt.maxlen = sizeof(SPX_OPTMGMT);
optmgmt.opt.len = sizeof(SPX_OPTMGMT);
optmgmt.opt.buf = (char *)&spxOptionSet;

/* Spx also supports the T_CHECK and T_DEFAULT flags
for t_optmgmt */

optmgmt.flags = T_NEGOTIATE;

if (t_optmgmt(spxFd, &optmgmt, &optmgmt)<0) {
        t_error(" t_optmgmt failed ");
        ..
        ..
}
```

t_rcv() Receive Data Over a Connection.

This call enables transport users to receive either normal or expedited data on a transport connection.

Usage

#include <tiuser.h>
#include <ipx_app.h>
#include <spx_app.h>

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

Description

This call works as specified in the *AT&T UNIX System V Network Programmers Guide*, except that the T_EXPEDITED flag will never be set.

Errors

It is possible that the remote transport end point could go away and the local transport end point will block while waiting for the incoming packet. This would cause the local transport end point to block forever. WatchEmu solves this problem by periodically checking all active connections. If WatchEmu determines that the remote transport end point is no longer participating in this connection, it will generate a disconnect indication to the stream head (please refer to **t_rcvdis()** and WatchEmu).

The errors are specified in the *AT&T UNIX System V Network Programmers Guide*.

State

The **t_rcv()** call is allowed only in the T_DATAXFER state.

Example

```
/* the maximum amount of data we could receive per packet */
#define MAX_DATA_BYTES 534
int flags;
int bytesReceived;
unsigned char spxData[MAX_DATA_BYTES];
```

```
flags |= TMORE;

while  (flags & TMORE) {

       /* we indicate we can receive MAX_DATA_BYTES, bytesReceived
       will have the actual number of bytes we received */

       if ((bytesReceived=
                t_rcv(spxFd2, spxData, MAX_DATA_BYTES,&flags))<0) {
                         t_error(" t_rcv failed ");
                         ..
                         ..
       }

       /* if TMORE flag is off this is end of this transmission */
                if (flags & TMORE) {
       ..
       ..
       }
} /* end while */
```

t_rcvdis() Receive a Disconnection Notice.

This call identifies the reason for the abortive release of a connection, where the connection is released by the transport provider or another transport user. This call follows AT&T's definition of a disorderly release.

Usage

```
#include <tiuser.h>
#include <ipx_app.h>
#include <spx_app.h>

    int t_rcvdis(fd, discon)
    int fd;
    struct t_discon *discon;
    struct t_discon {
            struct netbuf udata;
            int reason;
            int sequence;
    }
```

Description

This call works as specified in the *AT&T UNIX System V Network Programmers Guide* except for the following.

SPX does not support the transmission of any user data with a disconnect request. Upon receiving a disconnect request, SPX will set the stream to a T_IDLE state.

If the program is expecting the remote transport end point to send an SPX terminate connection packet, the program should "spin," waiting until a disconnection indication such as the following arrives:

Upon receipt of an SPX disconnect indication the SPX driver assumes worst case and generates a disorderly release indication. If the DG/UX server application is expecting to receive a disconnection indication the application should spin, issuing a **t_rcvdis()** call. (This tends to use up CPU resources since **t_rcvdis()** is nonblocking; for a better way see **t_snddis()**.) Eventually the application will receive the disconnection indication, or WatchEmu will time out the connection.

Since a transmission error or a disconnect indication can arrive at any moment from the remote transport end point, the SPX user needs to be conscious of the asynchronicity of the disconnect indication arrival. The SPX user should check for a disconnect indication after every **t_snd()**; if one is received the reason code should also be checked. By checking the reason code it can be determined whether the error was generated by the SPX driver (i.e., transmission failure) or if a disconnect indication was received by the opposite transport end point.

The reason for issuing a **t_rcvdis()** after every **t_snd()** is that **t_snd()** doesn't check the read side of the local transport end point's stream for disconnect indications. If the remote end point fails to acknowledge an SPX data transmission, the SPX driver will generate a disconnect indication to the stream head and set the state of the local transport end point to T_IDLE. In the T_IDLE state SPX will drop all outbound data per AT&T's specification.

Errors

If any of the following conditions occur, a disconnect indication will be generated and passed to the stream head. The reason integer of the **t_discon()** structure will be set accordingly in the following situations:

The possible errors returned from **t_rcvdis()** are:

t_errno	errno	
TLI_SPX_ CONNECTION_ FAILED:		The remote transport end point fails to acknowledge any transmission. This is generated by WatchEmu after efforts to contact the remote transport end point. or The SPX driver could not reliably deliver the data or connection request. The remote transport end point doesn't acknowledge transmissions.
	ENOSTR:	The SPX driver has tried the maximum number of times to send the received data to the stream head unsuccessfully. This is usually caused by a very slow or deadlocked local process. The tunable for this number of retries can be incremented.
TLI_SPX_ CONNECTION_ TERMINATED:		No error. An SPX terminate connection packet was received from the remote transport end point.

State

The state after this call is T_IDLE.

Example

```
struct t_discon disconnectInfo;

/* If you know for sure that a disconnect indication has arrived t
hen you can issue one t_rcvdis. */

if (t_rcvdis(spxFd, &disconnectInfo)<0) {
        t_error(" t_rcvdis failed ");
        ..
        ..
}
```

```
switch (disconnectInfo.reason) {
case TLI_SPX_CONNECTION_FAILED:
        printf(" connection failed \n");
        ..
        ..
        break;
case TLI_SPX_CONNECTION_TERMINATED:
        printf("connection terminated by remote end point ");
        ..
        ..
        break;
default:
        printf(" t_rcvdis returned 0x%X \n",
        disconnectInfo.reason);
        ..
        ..
        break;
}
/* If you want to wait until a disconnect indication arrives you
will need to loop waiting for one to come in. You are guaranteed
to not loop forever. If the remote end point goes away without
sending a disconnect, WatchEmu will generate a disconnect
indication. */
while ((t_rcvdis(spxFd, &disconnectInfo)<0)
        && (t_errno == TNODIS));
/* or
while ((t_look(spxFd)==0) && (t_errno != T_DISCONNECT));
if (t_rcvdis(spxFd, &disconnectInfo)<0) {
        t_error(" t_rcvdis failed ");
        ..
        ..
}
*/
switch (disconnectInfo.reason) {
        case TLI_SPX_CONNECTION_FAILED:
              printf(" remote end point went away \n");
              ..
              ..
              break;
        case TLI_SPX_CONNECTION_TERMINATED:
              printf(" connection failed ");
              ..
              ..
              break;
        default:
              printf(" t_rcvdis returned 0x%X \n",
              disconnectInfo.reason);
              ..
              ..
              break;
}
```

t_snd() Send Data Over a Connection.

This call enables transport users to send either normal or expedited data over a transport connection.

Usage

#include <tiuser.h>
#include <ipx_app.h>
#include <spx_app.h>

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

Description

This call works as specified in the *AT&T UNIX System V Network Programmers Guide*.

All data including EXPEDITED data is sent on a first–come basis. The application must take care not to close the connection before all the data is sent since the **t_snd()** call could and does return before the data has actually been transmitted.

If data is transmitted in the T_IDLE state, the transport provider will drop the data. If the stream is in any state other than T_DATAXFER and the user program issues a data request, the stream will freeze with errno set to EPROTO and without t_errno set to TSYSERR.

The T_MORE flag is supported by SPX. If the DG/UX application sets the T_MORE flag when doing a **t_snd()**, the not–EOF bit is set in the SPX packet header. If the receiving client is a DG/UX machine the T_MORE flag will be set in the **t_rcv()** call. If the receiving client is a DOS/OS2 application the not–EOF bit will be set in the SPX packet header.

TLI will break down any buffer larger than 534 bytes into message requests 534 bytes long (the last message request will be the buffer size of mod 534), and will send these requests to SPX. If the buffer size is some multiple of 534, then the throughput of data will be maximized, since the only message size smaller than 534 bytes will be the last message. Since SPX cannot control the size of incoming packets, it will be receiving more than 512–byte packets.

Since most sector sizes are 512 and multiples thereof, it is best to use 512 as a send buffer size to maximize the throughput of data from the UNIX application to the network.

The SPX driver will not send an empty SPX packet. If nbytes is zero, no data or packet will be sent.

Errors

A known problem with TLI is its inability to notify the user within reasonable time that the **t_snd()** has failed. The **t_snd()** call doesn't check for disconnect indications before and after doing the **t_snd()**. The best way to notify **t_snd()** of an error is for the SPX driver to generate an M_ERROR message, but since this is not part of the TLI specification, another method must be used.

If the remote transport end point has gone away or failed to acknowledge the transmitted data, the SPX driver will generate a disconnect indication and change the state of the local transport end point to T_IDLE. Following the TLI specification, any **t_snd()** issued in the T_IDLE state is dropped by the SPX driver. From the SPX user's point of view, this indication will not be noticed unless the SPX user issues some call other than **t_snd()**. The SPX user should check the return code in the disconnect indication to make sure it is TLI_SPX_CONNECTION_TERMINATED.

The following errors can occur during the send request. These errors will lock the stream and disable the local transport end point. Only errno will be set to the following values; t_errno will not be affected.

If any of the following errors occur, close the connection with **t_close()**.

EPROTO: The data request was issued from a state other than T_DATA_XFER. This transport end point is no longer valid and must be closed.

or

The size of the data request header or data portion of the message received by SPX was invalid (too small or too large).

If the data cannot be reliably delivered to the remote transport end point the SPX driver will generate a disconnect indication (refer to **t_rcvdis()**) with reason set to the appropriate number reflecting the reason for failure.

State

t_snd() is only allowed in the T_DATAXFER state. If any errors occur, the state moves to T_IDLE.

Example

```
/* to optimize reads and use of streams buffers we use 512 */

#define TRANS_BUFFER_SIZE 512

int flags;
int bytesRead;
unsigned char readBuffer[TRANS_BUFFER_SIZE];
char *someFileString = "someFileName";
FILE *fp;
struct t_discon disconnectInfo;
```

```
/* open the file to send */
if ((fp=fopen(someFileString, "r+b")) == NULL) {
        perror(" open failed ");
        ..
}

/* while there is data in the file tell the remote end point that
there is still data for this transmission */

flags = TMORE;
while (!feof(fp)) {
        bytesRead=fread(readBuffer, 1, TRANS_BUFFER_SIZE, fp);

        if (t_snd(spxFd2, readBuffer, bytesRead, flags)<0) {
              t_error(" t_snd failed ");
              ..
              ..
        }
        ..
        ..
}

/* Check and make sure we haven't been cut off by remote end.
NOTICE that the return code is greater than zero if we received a
disconnect indication. */

if (t_rcvdis(spxFd2, &disconnectInfo)>0) {
        printf(" remote end point aborted connection \n");
        ..
        ..
}

/* send one byte with the TMORE flag turned off */
flags = 0;
if (t_snd(spxFd2, readBuffer, 1, flags)<0) {
        t_error(" t_snd failed ");
        ..
        ..
}
```

t_snddis() Send User Initiated Disconnect Request.

This call initiates the abortive release of a transport connection, and can be issued by either transport user. It may also be used to reject a connect request during the connection establishment phase.

Usage

#include <tiuser.h>
#include <ipx_app.h>
#include <spx_app.h>

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

Description

This call works as specified in the *AT&T UNIX System V Network Programmers Guide*.

Due to the original specification, SPX doesn't support the TLI concept of an orderly release. The correct procedure for aborting or terminating a connection is to use the **t_snddis()/t_rcvdis()** combination.

A **t_snddis()** call will send a connection termination request. This call is used when the application must abort or break a connection. A **t_snddis()** will generate an SPX terminate connection request and will release all outstanding data messages on both the local and remote transport end point. The terminate request is not reliably delivered; only one terminate request will be sent. If an acknowledge is not received, the terminate request will not be retried. After an application issues a **t_snddis()**, the application can do a **t_unbind()** and **t_close()** immediately without waiting.

SPX doesn't support the function of sending user data along with a disconnect request.

The correct procedure for terminating an SPX connection is for both transport end points to correlate the moment that the connection is no longer needed; both end points should then terminate the connection.

Errors

The possible errors returned from **s_snddis()** are:

t_errno	errno	
TSYSERR:	ENXIO:	The socket number for this transport end point was not found in the internal SPX table. This is an SPX driver error.

State

Regardless of the success of the call, the state will be T_UNBND.

Example

```
struct t_call disconnectIndication;

/* If you wish to terminate the current connection (the state is
T_DATAXFER) it is not necessary to send a pointer to the t_call
structure. */

if (t_snddis(spxFd, (struct t_call *) NULL )<0) {
         t_error(" t_snddis failed ");
         ..
         ..
}

/* If you wish to deny a connection request you must supply the
sequence number returned in the t_call returned from the t_listen
that received the connection request. The remoteAddress structure
is from the t_listen example. */

disconnectIndication.sequence = remoteAddress.sequence;

if (t_snddis(spxFd, &disconnectIndication )<0) {
         t_error(" t_snddis failed ");
         ..
         ..
}
```

t_unbind() Disable a Transport End Point.

This call disables a transport end point so that no further requests destined for the given end point will be accepted by the transport provider.

Usage

#include <tiuser.h>
#include <ipx_app.h>
#include <spx_app.h>

int t_unbind(*fd***)**
int *fd***;**

Description

This call works as specified in the *AT&T UNIX System V Network Programmers Guide*.

This call releases the socket number used by this transport end point for future use. This allows the process to bind to a new socket number.

Errors

The possible errors returned from **t_unbind()** are:

t_errno	**errno**	
TSYSERR:	ENXIO:	The socket number for this connection was not found in the socket table. Notify Novell. This is an SPX driver error.
TOUTSTATE:		This transport end point is not in the T_BND state so **t_unbind()** doesn't apply.

State

This call places the transport end point in the T_UNBND state.

Example

```
if (t_unbind(spxFd)<0) {
        t_error(" t_unbind failed ");
        . .
        . .
}
```

Unsupported Transport Interface calls

Table 3–1 lists the TLI calls not supported by SPX for AViiON Systems. If any of these calls are issued, errno will be set to TNOTSUPPORT.

Table 3–1 Unsupported SPX TLI Calls

Function	Description
t_rcvrel()	Receive an orderly disconnect
t_sndrel()	Send an orderly disconnect
t_sndudata()	Send a data unit (connectionless)
t_rcvudata()	Receive data unit (connectionless)
t_rcvuderr()	Receive a unit data error indication (connectionless)

The **t_sndudata()**, **t_rcvudata()**, and **t_rcvuderr()** calls are not supported by SPX because SPX is a connection–based service; these calls are used for connection–less service only.

The **t_sndrel()** and **t_rcvrel()** calls are not supported by SPX because SPX is a T_COTS service; these calls are used only for T_COTS_ORD service.

End of Chapter

Chapter 4
IPX Network Layer Interface

This chapter defines the use of the Internet Packet Exchange (IPX) under the DG/UX Transport Layer Interface (TLI). The DG/UX IPX driver supports all calls for a datagram service. Figure 4–1 shows the TLI interface and associated IPX transport provider stack.

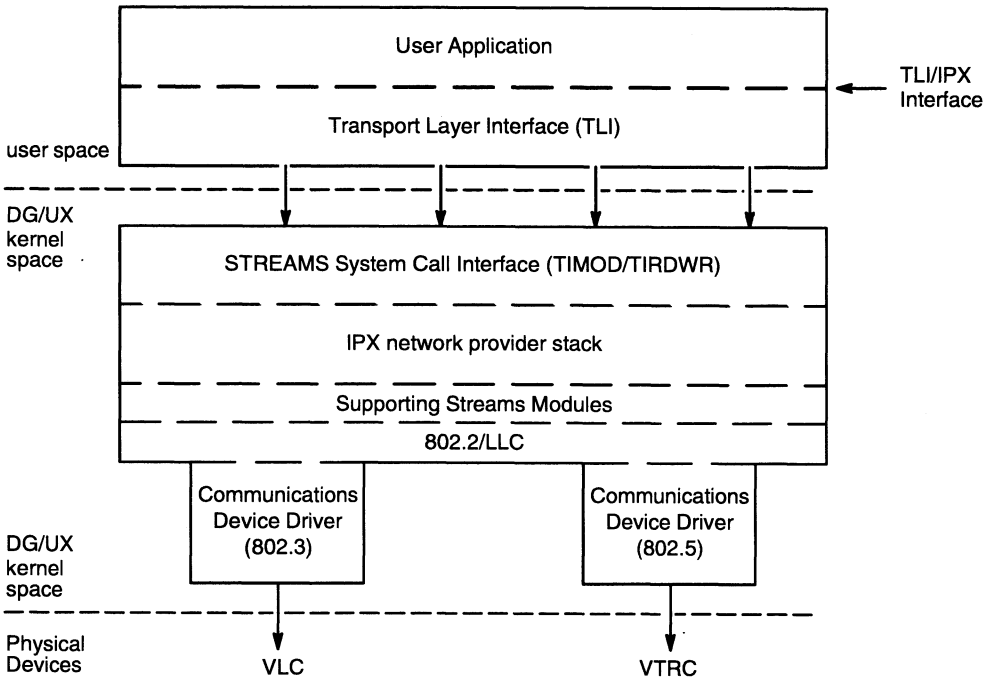


Figure 4–1 TLI/IPX Programming Interface

The Internet Packet Exchange (IPX) is a datagram service under the Transport Layer Interface (TLI). IPX is connectionless, and does not support a connection or guarantee delivery. However when delivery is made, IPX guarantees the accuracy of the data sent.

IPX allows a user process to send and receive individual packets. Use it when you do not require a guaranteed service, such as on a very reliable network, or in cases where an occasional lost packet is not critical, as in service advertising. (You can build guaranteed services such as the Sequence Packet Exchange (SPX) on top of IPX.) Use IPX for processes that need datagram messages.

A DG/UX process follows these steps in order to use IPX:

1. **t_open(***fd***);**
2. **t_bind(***fd***,** *bind***,** *bind***);**
3. **t_optmgmt(***fd***,** *req***,** *ret***);** (optional)

4. **t_sndudata(***fd, ud, flags***);**

   or

   t_rcvudata(*fd, ud,* **&***flags***);**

5. **t_unbind(***fd***);**

6. **t_close(***fd***);**

The standard use of these calls is described in the *AT&T UNIX System V Documentation Set*. This chapter provides the details of the peculiarities of the IPX calls.

IPX Considerations

Consider the following issues before you use IPX:

- The IPX driver must be installed to enable the use of IPX. If the driver is not installed, the **t_open()** call will fail.
- The IPX device name for DG/UX is **/dev/ipx**.
- The IPX controlling device name is **/dev/ipx0**.
- IPX follows the state diagram in the *AT&T System V Network Programmers Guide* for a connectionless service.
- Since STREAMS allocates message blocks for IPX that are sized in powers of two, outgoing data requests are best done with a buffer size of 512 bytes. Since IPX cannot control the data size of incoming packets, the driver will support packet data sizes of 0 through 546 bytes.
- If a signal is sent to the IPX application during a **t_rcvudata()**, **t_rcvudata()** will fail with errno set to EINTR.

IPX Calls

This section describes the IPX calls. The descriptions include the format of the call, a description, possible errors, states, and an example. The calls are presented in the order that you use them. Table 4–1 lists the calls alphabetically and the page where the call description begins.

Table 4–1 IPX Calls Listed Alphabetically

Function	Page	Description
t_bind()	4-6	Returns binary error code indicating either success (0) or failure (–1).
t_close()	4-16	Returns a binary error code indicating success (0) or failure (non–zero).
t_open()	4-4	Returns transport end point file descriptor (fd) to be used in future TLI function calls.
t_optmgmt()	4-8	Returns a binary error code indicating success (0) or (–1) failure).
t_rcvudata()	4-13	Receives both directed datagrams and broadcast datagrams.
t_sndudata()	4-10	Sends NetBIOS datagrams.
t_unbind()	4-15	Returns a binary error code indicating success (0) or failure (non–zero).

t_open() Establish a Transport End Point.

This call creates a transport end point and returns protocol–specific information associated with that end point. It also returns a file descriptor that serves as the local identifier of the end point.

Usage

#include <tiuser.h>
#include <ipx_app.h>

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

Description

This call is supported as documented in the *AT&T System V Network Programmers Guide*. You may use IPX either synchronously or asynchronously.

The UNIX streams architecture requires that a process be run to build and hold a stream. When you install the Ethernet driver, link it to the IPX driver. If this driver is not installed, all attempts to open IPX will fail.

t_open() returns a file descriptor and an information structure (of type t_info) upon the successful return of an open. This information structure contains pertinent information about IPX. The data format is as follows:

1. The maximum transport service data unit (TSDU) is 546 bytes.
2. No expedited TSDU is supported.
3. No data is transmitted with a disconnect request.
4. The address size is 12 bytes, of which network is 4 bytes, node is 6 bytes, and socket is 2 bytes.
5. The service type is always T_CLTS.

Errors

Refer to the *AT&T System V Network Programmer's Guide* for the errors that may occur with this call.

If the IPX driver cannot allocate memory it will generate an error. This error will set t_errno to TSYSERR and errno to ENOSR.

State

The state will always follow the state diagram in the *AT&T System V Network Programmer's Guide*.

Example

```
char *ipxPath = "/dev/ipx";
struct t_info ipxInfo;
int ipxFd;
if ((ipxFd=t_open(ipxPath,O_RDWR,&ipxInfo))<0) {
        t_error(" t_open failed ");
        ..
        ..
}
```

t_bind() Bind an Address to a Transport End Point.

This call associates a protocol address with a given transport end point, thereby activating the end point. It also directs the transport provider to begin accepting incoming packets.

Usage

#include <tiuser.h>
#include <ipx_app.h>

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

Description

This call is supported as documented in the *AT&T System V Network Programmers Guide*, with the following additions:

The **t_bind()** call allows an application to bind to a socket number, which can be either dynamic or specific. IPX keeps track of which socket number (dynamic or specific) is bound to which transport end point.

A dynamic socket number is an unused socket number returned by the IPX driver, and is guaranteed to be a unique unused number on the local DG/UX system. A dynamic socket is a random value between 0x4000 and 0x5000, inclusive.

A specific socket number is the socket number requested in the req **t_bind()** structure. If it is unused, it is granted and returned in the ret **t_bind()** structure in the range of 1 to FFFFh.

The **t_bind()** call requires that a pointer to an ipxAddr_t type structure be passed in the req–>addr.buf field. This ipxAddr_t structure is as follows:

```
typedef struct ipxAddr {
    unsigned char net[4];
    unsigned char node[6];
    unsigned char socket[2];
} ipxAddr_t;
```

The IPX driver will look at the socket field in the ipxAddr_t structure for the IPX user's desired socket number. The socket number must be passed in high–low byte order. If the socket number desired is not currently being used by another IPX user, the IPX driver will return the local net, local node, and the allocated or requested socket number in the corresponding fields of the ipxAddr_t structure of the ret–>addr.buf field.

If an IPX user passes zero in the ipxAddr_t socket field, the IPX driver will attempt to allocate a dynamic socket number.

The IPX user can pass NULL instead of req and ret. The IPX driver will assume that the IPX user has requested a dynamic socket number and the IPX driver will try to allocate and return a dynamic socket number.

Only one process can bind to a given socket number at a time. If you try to bind to a socket that is already bound, an error results and the second bind fails. The first bind is unaffected.

Services written to run over IPX generally have well–known socket numbers associated with them. By having well–known socket numbers, IPX users can be sure that their server and client application types match. Another method to coordinate servers and clients is to use Service Advertising Protocols (SAPs). Contact Novell for more information on well–known socket numbers and SAPs.

Errors

If the IPX driver cannot allocate memory it will generate an error. This error will set t_errno to TSYSERR and errno to ENOSR.

The possible errors returned from **t_bind()** are

t_errno	**errno**	
TSYSERR:	ENOSR:	There were no message buffers available to acknowledge the bind request.
TOUTSTATE:		This connection is in a state that invalidates a **t_bind()** request.
TNOADDR:		There are no unused dynamic socket numbers. The IPX user should try again.
TBADADDR:		The socket number requested is in use.

State

The state will always follow the state diagram in the *AT&T System V Network Programmer's Guide*.

Example

```
#define SOCKET_TO_BIND_HIGH 0x45 /*high order byte */
#define SOCKET_TO_BIND_LOW 0x00 /*low order byte */
struct t_bind bind;
ipxAddr_t localAddress;
localAddress.sock[0] = SOCKET_TO_BIND_HIGH;
localAddress.sock[1] = SOCKET_TO_BIND_LOW;
bind.addr.len = sizeof(ipxAddr_t);
bind.addr.maxlen = sizeof(ipxAddr_t);
bind.addr.buf = (char *)&localAddress;
bind.qlen = 0;
if (t_bind( ipxFd, &bind, &bind)<0) {
         t_error(" t_bind failed ");
         ..
         ..
}
```

t_optmgmt() Option Management Request (Get Local Info).

This call lets the user process get or negotiate protocol options with the transport provider.

Usage

#include <tiuser.h>
#include <ipx_app.h>

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

Description

This call is supported as documented in the *AT&T System V Network Programmers Guide*, except that it returns the local address instead of negotiating options.

This option management call will return all the local information about this transport end point. It will return the source net, source node, and source socket pertaining to this local end point.

The req.buf and ret.buf must point to a structure large enough to hold an ipxAddr_t (12 bytes). Upon successful completion, *ret.buf will contain the source information. The first 4 bytes will be the local net, the next 6 the node, and the last 2 the local socket number. The local socket number will be valid only if this local end point has already bound. All this information will be in high–low byte order.

Errors

Refer to the *AT&T System V Network Programmer's Guide* for the errors that may occur with this call.

If the IPX driver cannot allocate memory, it will generate an error. This error will set t_errno to TSYSERR and errno to ENOSR.

State

The state will always follow the state diagram in the *AT&T System V Network Programmer's Guide*.

Example

```
struct t_optmgmt optionsRequest;
ipxAddr_t localIpxAddress;
optionsRequest.opt.maxlen = sizeof(ipxAddr_t);
optionsRequest.opt.len = sizeof(ipxAddr_t);
optionsRequest.opt.buf = (char *)&localIpxAddress;

/* flags are not used with the IPX options request */
optionsRequest.flags = 0;

/* ipxFd is the file descriptor of an opened IPX device*/

if(t_optmgmt(ipxFd, &localIpxAddress, &localIpxAddress)<0) {
         t_error(" t_optmgmt failed ");
         ..
         ..
}
```

t_sndudata() Send Unit Data.

This call enables transport users to send a self–contained data unit to the user at the specified protocol address.

Usage

#include <tiuser.h>
#include <ipx_app.h>

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

Description

This call is supported as documented in the *AT&T System V Network Programmers Guide*, except that IPX doesn't support option data.

The address (addr) must point to an ipxAddr_t type. This is a full IPX address structure. The destination net, node, and socket must be filled (high–low byte order) in this IPX address structure by the user program. The IPX user can set the packet type of the outgoing IPX packet by passing one byte specifying the packet type in the options (opt) field. The checksum, length, transport control, source net address, source node address, and source socket address in the outgoing packet will be filled in by the IPX driver.

If the destination net is not found in the router table the packet will be dropped.

The fact that the **t_sndudata()** call has returned successfully doesn't guarantee that the data has been sent. It does guarantee that the data has been queued up to be sent. If a **t_close()** is issued all queued data is lost.

All data, including EXPEDITED data, is sent on a first–come basis.

Since the largest number of bytes that can be sent with any **t_sndudata()** is 546, any attempt to send more than 546 bytes will result in an error.

Errors

If the IPX driver cannot allocate memory, it will generate an error. This error will set t_errno to TSYSERR and errno to ENOSR.

The following errors can also occur. errno will be set to one of the following values. t_errno will not be affected.

EPROTO:

1. The data request was issued from a state other than T_IDLE. This transport end point is no longer valid and must be closed.

2. The size of the data request header or data portion of the message received by IPX was invalid (too large).

ENOLINK: The link to IPX has been broken. Probably the ipx daemon was killed. The IPX driver will no longer function.

State

The state will always follow the state diagram in the *AT&T System V Network Programmer's Guide*.

Example

```
unsigned char remoteServerName[MAX_REMOTE_NAME_LENGTH];
ipxAddr_t remoteServerAddress;
unsigned char ipxPacketType;
unsigned char ipxData[IPX_MAX_DATA_SIZE];

/* There are different approaches to obtaining the address of
the end point you wish to send to. One can query a netware
bindery for a server of the type you wish. Or one can create a
"yellow pages" file that maps a server name to an address. This
example will show the "yellow pages" scenario.

This routine will find which remote server the user wants to
send to and place the name in remoteServerName. */

if (GetServerName(remoteServerName)<0) {
        printf(" could not get remote host name \n");
        ..
        ..
}

/* This next routine will take the server name and look in a
file to find the ipx address associated with that server.
GetIpxAddress will fill in remoteServerAddress with the remote
servers address in high-low byte order. */
```

```
if(GetIpxAddress(remoteServerName,&remoteServerAddress)<0) {
        printf(" could not get remote server address \n");
        ..
        ..
}

ud.opt.len = 1;
ud.opt.maxlen = 1;
ud.opt.buf = (char *)&ipxPacketType;

ud.addr.len = sizeof(ipxAddr_t);
ud.addr.maxlen = sizeof(ipxAddr_t);
ud.addr.buf = (char *)&remoteServerAddress;

ud.udata.maxlen = IPX_MAX_DATA_SIZE;

/* actual number of data bytes sent */
ud.udata.len = IPX_MAX_DATA_SIZE ;
ud.data.buf = (char *)&ipxData[0];

/* flags are not applicable to IPX datagrams */
flags = 0;
if (t_sndudata(ipxFd, &ud, &flags)<0) {
        t_error(" t_sndudata failed \n");
        ..
        ..
}
```

t_rcvudata() Receive Unit Data.

This call enables transport users to receive data units from other users.

Usage

#include <tiuser.h>
#include <ipx_app.h>

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

Description

This call is supported as documented in the *AT&T System V Network Programmers Guide*, except that IPX doesn't support option data.

This call functions exactly opposite of **t_sndudata()**. The address of the sender is returned to the ud–>addr field. The packet type is in the ud.opt field, and the packet data is in the ud–>udata field. There is no flow control on incoming data since IPX is a datagram service. If the IPX application cannot service the incoming data as fast as the sender generates it, the IPX driver will drop the excess incoming packets. The len field of opt, udata, and addr will be set according to the incoming packet. The amount of data received in the IPX packet will be in udata.len.

Errors

Refer to the *AT&T System V Network Programmer's Guide* for the errors that may occur with this call.

If the IPX driver cannot allocate memory it will generate an error. This error will set t_errno to TSYSERR and errno to ENOSR.

State

The state will always follow the state diagram in the *AT&T System V Network Programmer's Guide*.

Example

```
#define IPX_MAX_DATA 546
struct t_unitdata ud;
unsigned char ipxPacketType;
unsigned char ipxDataBuf[IPX_MAX_DATA];
ipxAddr_t sourceAddress;
int flags;
```

```
/* When the t_rcvudata unblocks ipxPacketType will have the
packet type from the IPX packet */

ud.opt.len = 1;
ud.opt.maxlen = 1;
ud.opt.buf = (char *)&ipxPacketType;

/* When the t_rcvudata unblocks sourceAddress will have the IPX
address of the datagram sender */

ud.addr.len = sizeof(ipxAddr_t);
ud.addr.maxlen = sizeof(ipxAddr_t);
ud.addr.buf = (char *)&sourceAddress;

/* When the t_rcvudata unblocks ipxDataBuf will contain the
data in the IPX packet */

ud.udata.len = IPX_MAX_DATA;
ud.udata.maxlen = IPX_MAX_DATA;
ud.data.buf = (char *)&ipxDataBuf[0];

if (t_rcvudata(ipxFd, &ud, &flags)<0) {
        t_error(" t_rcvudata failed ");
        ..
        ..
}
```

t_unbind() Disable a Transport End Point.

This call disables a transport end point so that no further requests destined for the given end point will be accepted by the transport provider.

Usage

#include <tiuser.h>
#include <ipx_app.h>

int t_unbind(*fd***)**
int *fd***;**

Description

This call is supported as documented in the *AT&T System V Network Programmers Guide*, with the following additions:

This call releases the socket number that this transport end point used. This call also places the transport end point in a T_UNBND state, allowing the process to bind to a new socket number. The **t_close()** routine calls the **t_unbind()** routine.

Error

If the IPX driver cannot allocate memory, it will generate an error. This error will set t_errno to TSYSERR and errno to ENOSTR.

The following errors can also occur:

t_errno	**errno**	
TSYSERR:	ENOSTR:	This is an IPX driver error. No streams buffers were available. The unbind was still successful.
TOUTSTATE:		This transport end point is not in the T_IDLE state, so **t_unbind()** doesn't apply.

State

The state will always follow the state diagram in the *AT&T System V Network Programmer's Guide*.

Example

```
int ipxFd;
if (t_unbind(ipxFd)<0) {
        t_error(" t_unbind failed ");
        ..
        ..
}
```

t_close() Close a Transport End Point.

This call informs the transport provider that the user is finished with the transport end point, and frees any local resources associated with that end point.

Usage

#include <tiuser.h>
#include <ipx_app.h>

int t_close(*fd***)**
int *fd***;**

Description

This call is supported as documented in the *AT&T System V Network Programmers Guide*, with the following additions:

A transport end point opened with **t_open()** should be closed with **t_close()** to facilitate IPX/TLI functioning. **t_close()** will call **t_unbind()**, but it is bad behavior for the application to issue a **t_close()** before issuing a **t_unbind()**.

Errors

Refer to the *AT&T System V Network Programmer's Guide* for the errors that may occur with this call.

If the IPX driver cannot allocate memory it will generate an error. This error will set t_errno to TSYSERR and errno to ENOSR.

State

The state will always follow the state diagram in the *AT&T System V Network Programmer's Guide*.

Example

```
t_close(ipxFd);
```

End of Chapter

Chapter 5
OSI/Platform Interface

The Open System Interconnect/ Platform for AViiON® Systems (OSI/Platform) provides a TLI interface to the OSI transport layer services. The OSI/Platform provides connection–based, reliable, sequenced–transport protocol services. Figure 5–1 shows the TLI interface and the associated OSI/Platform transport provider stack.

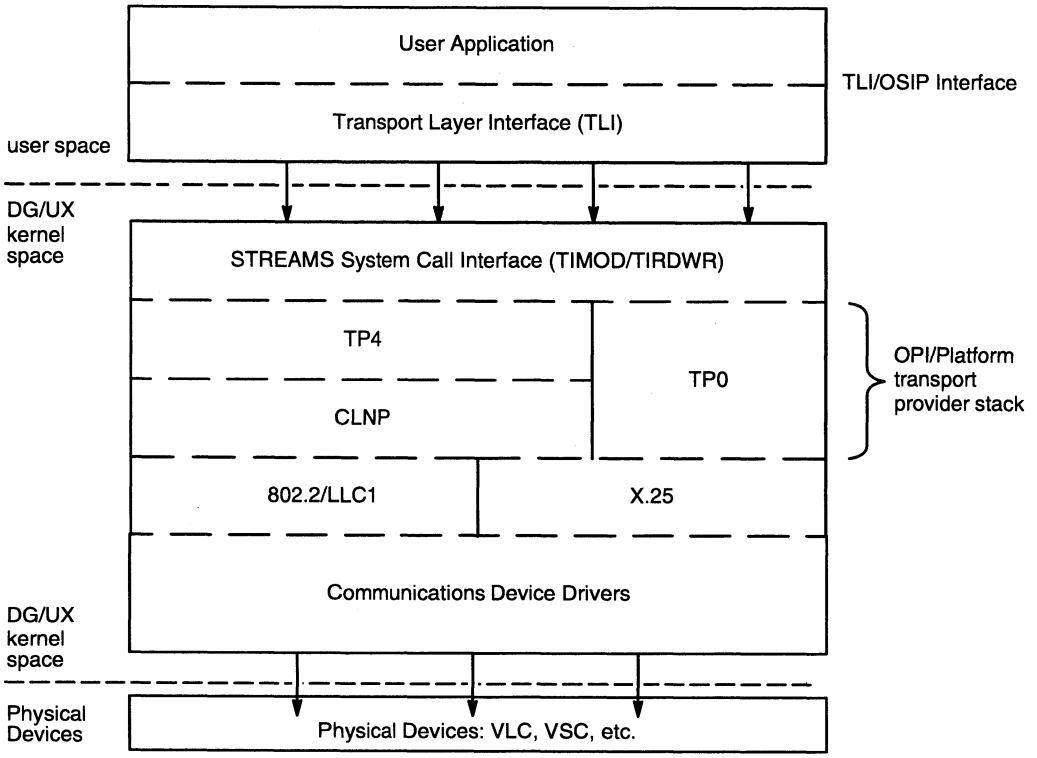


Figure 5–1 OSI/Platform Programming Interface

The TLI user interface for OSI/Platform conforms to the AT&T Transport Interface Library. Refer to the *AT&T UNIX V Documentation Set* for a description of the order and use of the calls. This chapter provides the details of the peculiarities of the OSI/Platform calls. Any TLI calls not mentioned in this chapter function with the OSI/Platform driver as specified in the *AT&T UNIX System V Network Programmer's Guide*.

Locating TLI–Related Documentation

As previously stated in the Preface, this manual assumes that you are familiar with the TLI interface and the DG/UX™ system programming environment. The following reading list is more specific to programming with the TLI on the OSI/Platform.

- *On–line DG/UX™ system man pages*

  For a complete description of the TLI calls, please refer to the on–line DG/UX™ system man pages or to the *Programmer's Reference for the DG/UX system, Volumes I and II*. (The on–line DG/UX™ system man pages may contain more up–to–date information.)

  Also for additional OSI/Platform information, refer to the **OSIP()**, **libnsap()**, **getnamebynsap()**, and **getnsapbyname()** on–line man pages.

- *OSI/Platform Release Notice*

  Refer to the Release Notice of the current revision of the OSI/Platform for AViiON Systems. The Release Notice lists additional error handling and troubleshooting information.

- *Setting up and Managing the OSI/Platform for AViiON Systems*

  This manual explains how to install, configure, manage, and troubleshoot the OSI/Platform for AViiON System.

- *AT&T UNIX System V Network Programmer's Guide*

  If you require additional information on the transport level interface, you may want to obtain a copy of the *AT&T UNIX System V Network Programmer's Guide*.

- *Programming with Transport Level (TLI) Interfaces for AViiON® Systems* (this manual)

  This manual explains the DG/UX system and TLI protocol–specific programming information.

See the Preface for additional information on these and other related documentation.

Compiling Your Program

For your program to access the transport layer interface at run–time, you must build your program with the TLI and OSI/Platform libraries. For example, use

```
% cc user.c -lnsl_s -lnsap ↵
```

The command first compiles your program (**user.c**) and then links the TLI Library (**nsl_s**) and the OSI/Platform Library (**nsap**) and your program.

OSI/Platform Calls

Table 5–1 lists the OSI/Platform TLI calls that require special consideration. The table also lists the **getnsapbyname()** and **getnamebynsap()** OSI/Platform calls. The chapter presents the calls in alphabetical order.

Table 5–1 OSI/Platform Calls

Function	Description
getnsapbyname()	Returns an OSI/P NSAP–to–name translation.
getnamebynsap()	Returns an OSI/P name–to–NSAP translation.
t_connect()	Attempts to initiate a connection with a remote user.
t_getinfo()	Returns the current characteristics of the OSI/Platform transport protocol for the file descriptor, *fd*.
t_listen()	Listens for incoming connect indications.
t_open()	Establish a Transport End Point.

getnsapbyname() Returns an OSI/P NSAP–to–name translation.

This call maps between a network name and a network address. The call extracts the information from the OSI/P NSAP network database and returns a pointer to the network address.

Usage

#include <tiuser.h>
#include <sys/osip/osip.h>

char ***getnsapbyname** (*name*, *nsaplen*)
char **name*;
int **nsaplen*;

Description

The **getnsapbyname()** call translates a null–terminated string pointed to by **name* and returns a pointer in **name* to a statically allocated area of memory containing the NSAP address (configured with the name or NSAP tag equivalent). On return from the call, the **nsaplen* integer contains the length (bytes) of the NSAP address.

Errors

The call returns a NULL if the NSAP–to–name translation fails for any reason.

getnamebynsap() Returns an OSI/P name–to–NSAP translation.

This call maps between a network address and a network name. The call extracts the information from the OSI/P NSAP network database and returns a pointer to the network name.

Usage

#include <tiuser.h>
#include <sys/osip/osip.h>

char ***getnamebynsap** (*nsap*, *nsaplen*)
char **nsap*;
int **nsaplen*;

Description

The **getnamebynsap**() call translates the NSAP address pointed to by **name*. The **nsaplen* integer contains the length (bytes) of the NSAP address.

The call returns a pointer in **name* to a statically allocated area of memory containing the null–terminated name or its NSAP tag equivalent configured for the NSAP address.

Errors

The call returns a NULL if the NSAP–to–name translation fails for any reason.

t_connect() Establish a Connection With Another Transport User.

This call attempts to initiate a connection with a remote user.

Usage

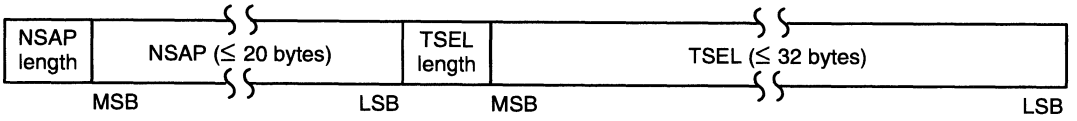
#include <tiuser.h>
#include <sys/osip/osip.h>

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

Description

This call works as specified in the *AT&T UNIX System V Network Programmers Guide.*

In the t_call structures, the addr buffer contains the OSI/Platform address – a maximum length of 54 bytes. Figure 5–2 shows the format of the address.



where

NSAP length is a 1–byte field that specifies the number of bytes in the network service access point (NSAP) field.

NSAP is the network service access point address of the remote host.

NOTE: When binding a file descriptor for the **t_listen()** call, the length of the NSAP field should be 0.

TSEL length is a 1–byte field that specifies the number of bytes in the TSEL field.

TSEL is the transport selector of a remote or local transport user. TSEL contains the transport selector of a remote transport user when connecting to a transport user on a remote host. TSEL contains the transport selector of a local transport user when binding to an address on the local host.

When you set TSEL to 0, the system generates a unique random value for TSEL.

Figure 5–2 OSI/Platform NSAP Address Format

The sysadm osipaddrs interface maintains an internal data base of NSAP tags or names. To translate one of these names to or from its NSAP address, use the **getnsapbyname()** or **getnamebynsap()** function. Refer to the man pages for descriptions of these functions.

In the t_call structures, the opt buffer passes the TLI options. The **osip.h** header file defines the option values. The options parameter is treated as follows:

option length = 0: connection is established with the transport class designated by the first NSAP–to–SNPA for NSAP mapping defined in OSI/Platform configuration.

option length = 1: the following values apply.

TP0_CONS	Transport Class 0 over X.25 connection is established, if possible.
TP4_CLNS	Transport Class 4 over CLNS connection is established, if possible.

option length > 1: connection is rejected with error code TBADOPT.

Make sure that the correct NSAP–to–SNPA mapping exists in the OSI/Platform configuration before you use any options with the **t_connect()** call. Thus if the option TP0_CONS is specified, an NSAP–to–SNPA mapping configured for the TP0/X.25 stack must exist in the OSI/Platform configuration.

t_getinfo() Get Protocol–Specific Service Information.

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

Usage

#include <tiuser.h>
#include <sys/osip/osip.h>

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

Description

This call works as specified in the *AT&T UNIX System V Network Programmers Guide*.

The maximum length of expedited data supported, as returned in the etsdu parameter in the t_info structure, is 16 bytes. Expedited data is only used with Transport Class 4, and it must be negotiated by the peer transport services.

On a successful return, the servtype parameter in the t_info structure is always set to T_COTS.

t_listen() Listen for a Connect Request.

This call enables the passive transport user to receive indications of connect requests from other transport users.

Usage

#include <tiuser.h>
#include <sys/osip/osip.h>

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

Description

This call works as specified in the *AT&T UNIX System V Network Programmers Guide.*

In the t_call structure, the addr buffer contains the OSI/Platform address – a maximum length of 54 bytes. Figure 5–2 shown on Page 5-6 shows the format of the address.

In the t_call structure, the opt buffer receives the TLI options. The **osip.h** header file defines the option values.

t_open() Establish a Transport End Point.

This call creates a transport end point and returns protocol–specific information associated with that end point. It also returns a file descriptor that serves as the local identifier of the end point.

Usage

#include <tiuser.h>
#include <sys/osip/osip.h>

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

Description

This call works as specified in the *AT&T UNIX System V Network Programmers Guide*.

The **t_open()** function creates a transport end point (e.g., a file descriptor) for use in subsequent network interface operations. The path variable is the name of the streams transport device driver (**/dev/cots**).

The maximum length of expedited data supported, as returned in the etsdu parameter in the t_info structure, is 16 bytes. Expedited data is only used with Transport Class 4, and it must be negotiated by both transport users.

On a successful return, the servtype parameter in the t_info structure is always set to T_COTS.

You cannot send user data during a connection establishment.

Unsupported Transport Interface Calls

Table 5–2 lists the TLI calls not supported by OSI/Platform for AViiON® Systems. If any of these calls are issued, errno will be set to TNOTSUPPORT.

Table 5–2 Unsupported OSI/Platform TLI Calls

Function	Description
t_rcvrel()	Receive an orderly disconnect
t_sndrel()	Send an orderly disconnect
t_sndudata()	Send a data unit (connectionless)
t_rcvudata()	Receive a data unit (connectionless)
t_rcvuderr()	Receive a unit data error indication (connectionless)

The **t_sndudata()**, **t_rcvudata()**, and **t_rcvuderr()** calls are not supported by OSI/Platform because OSI/Platform is a connection–based service; these calls are used for connectionless service only.

The **t_sndrel()** and **t_rcvrel()** calls are not supported by the OSI/Platform because it is a T_COTS service; these calls are used only for T_COTS_ORD service.

Troubleshooting

The OSI/Platform for AViiON Systems supports troubleshooting facilities for its transport and network services. The system allows you to log transport and network service errors using the standard DG/UX STREAMS *streer(1m)* error logging mechanism. For further troubleshooting information, refer to the *Setting up and Managing the OSI/Platform for AViiON Systems* manual.

The system also returns to your application system error codes. The **osip.h** header file defines each of the following error codes.

Mnemonic	Value	Description
EDRRNS	0	DR TPDU received, reason not specified. Get further diagnostics from the error log or trace.
EDRCAT	1	DR TPDU received, congestion at TSAP. This reflects a temporary resource shortage at the remote system.
EDRSENA	2	DR TPDU received, session entity not attached to TSAP. The remote application is not running or configured.
EDRAU	3	DR TPDU received, address unknown. Verify the proper configuration of addressing information on the local and remote systems.

ECFNCNA	4	Network characteristic not available at the remote TSAP. Verify the proper configuration of addressing information on the local and remote systems.
ECFNENC	9	Insufficient network connections available to support the connection requested. This reflects a resource shortage on the local system.
EDRND	128	DR TPDU received, normal disconnect from application. The remote application has either terminated or issued a normal transport disconnect request.
EDRRTEC	129	DR TPDU received, remote transport entity congestion at connect request time. This reflects a temporary resource shortage on the remote system.
EDRCNR	130	DR TPDU received, connection negotiation rejected. Verify that the configuration of transport parameters matches on the local and remote systems.
EDRDSRD	131	DR TPDU received, duplicate source references detected. Get further diagnostics from the error log or trace.
EDRMMR	132	DR TPDU received, mismatched references detected. Get further diagnostics from the error log or trace.
EDRPR	133	DR TPDU received, protocol error detected. Get further diagnostics from the error log or trace.
EDRRO	135	DR TPDU received, reference overflow. The maximum number of connections supported at the remote system may be insufficient.
EDRCRR	136	DR TPDU received, connection request refused on this network connection. Verify the proper configuration of addressing information on the remote system.
EDRHLI	138	DR TPDU received, header or parameter length invalid. This indicates a protocol error in either the local or remote transport provider.
ETPNRA	3827	No resource available. This is due to an application using Transport Class 4 attempting to reconnect before the frozen reference timer has expired. Either reconfigure the frozen reference timer value or retry the connect request later.
ETRAVF	3829	Address verification failure. The local transport provider could not verify the remote address on either a T–Connect Request or an N–Connect Indication. Verify the proper configuration of addressing information on the local system.

EDCAC	8212	Disconnection, abnormal condition. Get further diagnostics from the error log or trace.
EDCTUE	8213	Disconnection, transport services user error. Get further diagnostics from the error log or trace.
EDCCTE	8214	Disconnection, connection timer expired. Verify that the the remote transport is ready to receive connections.
EDCRTE	8215	Disconnection, retry timer expired. A network message has been retransmitted for the maximum amount of time and number of retries without acknowledgment. Most likely the remote system is not receiving network messages, or perhaps network messages are taking longer to reach the remote system than is allowed by the current setting of the maximum number of retransmissions allowed.
EDCITE	8216	Disconnection, inactivity timer expired. No network message has been received for the period of the inactivity timer. Verify that the local transport's inactivity timer is set longer than the remote transport's window timer.
EDCNCF	8221	Disconnection, network connection failure. The network connection being used for this transport connected has been terminated.
EDCNRR	8222	Disconnection, network connection reset. The network connection being used for this transport connected has been reset.
ETRUTA	8227	The called transport address is not configured. Verify the proper configuration of addressing information on the local system.
ECFUTA	8228	The calling transport address is not configured. This is likely to be due to an internal problem in the transport provider.
ESYSBS	20480	Disconnection, severe buffer shortage. This reflects a temporary (though severe) resource shortage on the local system.

End of Chapter

Glossary

The following are terms and definitions used throughout this document.

DLL

Data Link Layer. It resides between the Network layer and the Physical layer. It consists of a Logical Link Control sublayer and the MAC sublayer.

downstream

The direction from the Stream Head to driver.

driver

The end of the Stream closest to an external interface. The principal functions of the driver are handling any associated device, and transforming data and information between the external interface and the Stream.

LLC

Logical Link Control. The upper portion of the Data Link layer that supports media–independent data link functions, and uses the services of the MAC sublayer to provide services to the Network layer.

LLC provider

The Data Link layer protocol that provides the services of the Data Link Provider Interface.

Local Transport End Point

This is the local user application that is using the Internet Packet Exchange (IPX) or Sequence Packet Exchange (SPX) driver. A UNIX SPX application can have more than one local transport end point associated with it (i.e. a multiple–connection server).

MAC

Medium Access Control. The lower portion of the Data Link layer responsible for controlling transmission and reception of data packets over the wire.

message

One or more linked blocks of data or information, with associated STREAMS control structures containing a message type. Messages are the only means of transferring data and communicating with a Stream.

module

Software that performs functions on messages as they flow between Stream head and driver. A module is a STREAMS counterpart to the commands in a shell pipeline except that a module contains a pair of functions which allow independent bidirectional (downstream and upstream) data flow and processing.

multiplexor

A mechanism for connecting multiple Streams to a multiplexing driver. The mechanism supports the processing of the interleaved data Streams and the processing of internet working protocols. The multiplexing driver routes messages among the connected Streams. The other end of a Stream connected to a multiplexing driver is typically connected to a device driver.

NetBEUI

NetBIOS Extended User Interface. Typically refers to the transport layer protocol that is compatible with the IBM LAN Support Program.

NetBIOS

Network Basic Input Output System. Refers to the transport layer programming interface.

NTP

NetBEUI Transport Provider. The kernel–level protocol that accesses the services of the data link layer

OSI

Open System Interconnect. The term used by ISO, the International Organization for Standardization Organization, to refer to its seven–layer communication model.

Remote Transport End Point

This is the remote user application that is connected to the local transport end point.

source routing

Method by which stations are addressed across multiple rings.

SPX driver

This is the DG/UX SPX transport provider that services the SPX transport end points.

SPX user

This is the UNIX user application or UNIX application developer.

Stream

The kernel aggregate created by connecting STREAMS components, resulting from an application of the STREAMS mechanism. The primary components are a Stream head, a driver, and zero or more pushable modules between the Stream head and driver. A Stream forms a full duplex processing and data transfer path in the kernel, between a user process and a driver. A Stream is analogous to a Shell pipeline except that data flow and processing are bidirectional.

Stream head

The end of the Stream closest to the user process. The Stream head provides the interface between the Stream and the user process. The principal functions of the Stream head are processing STREAMS–related system calls, and bidirectional transfer of data and information between a user process and messages in STREAMS's kernel space.

STREAMS

A kernel mechanism that supports development of network services and data communication drivers. It defines interface standards for character input/output within the kernel, and between the kernel and user level. The STREAMS mechanism comprises integral functions, utility routines, kernel facilities and a set of structures.

upstream

The direction from driver to Stream head.

End of Glossary

Index

Within the index, a bold page number indicates a primary reference. A range of page numbers indicates the reference spans those pages.

U

W

TIPS ORDERING PROCEDURES

TO ORDER

1. An order can be placed with the TIPS group in two ways:
   A. MAIL ORDER – Use the order form on the opposite page and fill in all requested information. Be sure to include shipping charges and local sales tax. If applicable, write in your tax exempt number in the space provided on the order form.

   B. Send your order form with payment to:

   Data General Corporation
   ATTN: Educational Services/TIPS G155
   4400 Computer Drive
   Westboro, MA 01581–9973

   C. TELEPHONE – Call TIPS at (508) 870–1600 for all orders that will be charged by credit card or paid for by purchase orders over $50.00. Operators are available from 8:30 AM to 5:00 PM EST.

METHOD OF PAYMENT

2. As a customer, you have several payment options:
   A. Purchase Order – Minimum of $50. If ordering by mail, a hard copy of the purchase order must accompany order.

   B. Check or Money Order – Make payable to Data General Corporation. Credit Card – A minimum order of $20 is required for MasterCard or Visa orders.

SHIPPING

3. To determine the charge for UPS shipping and handling, check the total quantity of units in your order and refer to the following chart:

Total Quantity	Shipping & Handling Charge
1–4 Items	$5.00
5–10 Items	$8.00
11–40 Items	$10.00
41–200 Items	$30.00
Over 200 Items	$100.00

If overnight or second day shipment is desired, this information should be indicated on the order form. A separate charge will be determined at time of shipment and added to your bill.

VOLUME DISCOUNTS

4. The TIPS discount schedule is based upon the total value of the order.

Order Amount	Discount
$0–$149.99	0%
$150–$499.99	10%
Over $500	20%

TERMS AND CONDITIONS

5. Read the TIPS terms and conditions on the reverse side of the order form carefully. These must be adhered to at all times.

DELIVERY

6. Allow at least two weeks for delivery.

RETURNS

7. Items ordered through the TIPS catalog may not be returned for credit.
8. Order discrepancies must be reported within 15 days of shipment date. Contact your TIPS Administrator at (508) 870–1600 to notify the TIPS department of any problems.

INTERNATIONAL ORDERS

9. Customers outside of the United States must obtain documentation from their local Data General Subsidiary or Representative. Any TIPS orders received by Data General U.S. Headquarters will be forwarded to the appropriate DG Subsidiary or Representative for processing.

TIPS ORDER FORM

Mail To: Data General Corporation
Attn: Educational Services/TIPS G155
4400 Computer Drive
Westboro, MA 01581 - 9973

BILL TO:	**SHIP TO:** (No P.O. Boxes - Complete Only If Different Address)
COMPANY NAME _____	COMPANY NAME _____
ATTN: _____	ATTN: _____
ADDRESS _____	ADDRESS (NO PO BOXES) _____
CITY _____	CITY _____
STATE _____ ZIP _____	STATE _____ ZIP _____

Priority Code _____ (See label on back of catalog)

Authorized Signature of Buyer (Agrees to terms & conditions on reverse side)	Title	Date	Phone (Area Code)	Ext.

ORDER #	QTY	DESCRIPTION	UNIT PRICE	TOTAL PRICE

A SHIPPING & HANDLING

☐ UPS	ADD
1–4 Items	$5.00
5–10 Items	$8.00
11–40 Items	$10.00
41–200 Items	$30.00
200+ Items	$100.00

Check for faster delivery

Additional charge to be determined at time of shipment and added to your bill.

☐ UPS Blue Label (2 day shipping)
☐ Red Label (overnight shipping)

B VOLUME DISCOUNTS

Order Amount	Save
$0–$149.99	0%
$150–$499.99	10%
Over $500.00	20%

Tax Exempt # or Sales Tax (if applicable) _____

ORDER TOTAL	
Less Discount See B	–
SUB TOTAL	
Your local* sales tax	+
Shipping and handling – See A	+
TOTAL – See C	

C PAYMENT METHOD

☐ Purchase Order Attached ($50 minimum)
P.O. number is _____. (Include hardcopy P.O.)
☐ Check or Money Order Enclosed
☐ Visa ☐ MasterCard ($20 minimum on credit cards)

Account Number ☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐ Expiration Date ☐☐☐☐

_____
Authorized Signature
(Credit card orders without signature and expiration date cannot be processed.)

THANK YOU FOR YOUR ORDER

PRICES SUBJECT TO CHANGE WITHOUT PRIOR NOTICE.
PLEASE ALLOW 2 WEEKS FOR DELIVERY.
NO REFUNDS NO RETURNS.

* Data General is required by law to collect applicable sales or use tax on all purchases shipped to states where DG maintains a place of business, which covers all 50 states. Please include your local taxes when determining the total value of your order. If you are uncertain about the correct tax amount, please call 508–870–1600.

Form 702
Rev. 8/87

DATA GENERAL CORPORATION
TECHNICAL INFORMATION AND PUBLICATIONS SERVICE

TERMS AND CONDITIONS

Data General Corporation ("DGC") provides its Technical Information and Publications Service (TIPS) solely in accordance with the following terms and conditions and more specifically to the Customer signing the Educational Services TIPS Order Form. These terms and conditions apply to all orders, telephone, telex, or mail. By accepting these products the Customer accepts and agrees to be bound by these terms and conditions.

1. CUSTOMER CERTIFICATION
Customer hereby certifies that it is the owner or lessee of the DGC equipment and/or licensee/sub–licensee of the software which is the subject matter of the publication(s) ordered hereunder.

2. TAXES
Customer shall be responsible for all taxes, including taxes paid or payable by DGC for products or services supplied under this Agreement, exclusive of taxes based on DGC's net income, unless Customer provides written proof of exemption.

3. DATA AND PROPRIETARY RIGHTS
Portions of the publications and materials supplied under this Agreement are proprietary and will be so marked. Customer shall abide by such markings. DGC retains for itself exclusively all proprietary rights (including manufacturing rights) in and to all designs, engineering details and other data pertaining to the products described in such publication. Licensed software materials are provided pursuant to the terms and conditions of the Program License Agreement (PLA) between the Customer and DGC and such PLA is made a part of and incorporated into this Agreement by reference. A copyright notice on any data by itself does not constitute or evidence a publication or public disclosure.

4. LIMITED MEDIA WARRANTY
DGC warrants the CLI Macros media, provided by DGC to the Customer under this Agreement, against physical defects for a period of ninety (90) days from the date of shipment by DGC. DGC will replace defective media at no charge to you, provided it is returned postage prepaid to DGC within the ninety (90) day warranty period. This shall be your exclusive remedy and DGC's sole obligation and liability for defective media. This limited media warranty does not apply if the media has been damaged by accident, abuse or misuse.

5. DISCLAIMER OF WARRANTY
EXCEPT FOR THE LIMITED MEDIA WARRANTY NOTED ABOVE, DGC MAKES NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY AND FITNESS FOR PARTICULAR PURPOSE ON ANY OF THE PUBLICATIONS, CLI MACROS OR MATERIALS SUPPLIED HEREUNDER.

6. LIMITATION OF LIABILITY
A. CUSTOMER AGREES THAT DGC'S LIABILITY, IF ANY, FOR DAMAGES, INCLUDING BUT NOT LIMITED TO LIABILITY ARISING OUT OF CONTRACT, NEGLIGENCE, STRICT LIABILITY IN TORT OR WARRANTY SHALL NOT EXCEED THE CHARGES PAID BY CUSTOMER FOR THE PARTICULAR PUBLICATION OR CLI MACRO INVOLVED. THIS LIMITATION OF LIABILITY SHALL NOT APPLY TO CLAIMS FOR PERSONAL INJURY CAUSED SOLELY BY DGC'S NEGLIGENCE. OTHER THAN THE CHARGES REFERENCED HEREIN, IN NO EVENT SHALL DGC BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES WHATSOEVER, INCLUDING BUT NOT LIMITED TO LOST PROFITS AND DAMAGES RESULTING FROM LOSS OF USE, OR LOST DATA, OR DELIVERY DELAYS, EVEN IF DGC HAS BEEN ADVISED, KNEW OR SHOULD HAVE KNOWN OF THE POSSIBILITY THEREOF; OR FOR ANY CLAIM BY ANY THIRD PARTY.

B. ANY ACTION AGAINST DGC MUST BE COMMENCED WITHIN ONE (1) YEAR AFTER THE CAUSE OF ACTION ACCRUES.

7. GENERAL
A valid contract binding upon DGC will come into being only at the time of DGC's acceptance of the referenced Educational Services Order Form. Such contract is governed by the laws of the Commonwealth of Massachusetts, excluding its conflict of law rules. Such contract is not assignable. These terms and conditions constitute the entire agreement between the parties with respect to the subject matter hereof and supersedes all prior oral or written communications, agreements and understandings. These terms and conditions shall prevail notwithstanding any different, conflicting or additional terms and conditions which may appear on any order submitted by Customer. DGC hereby rejects all such different, conflicting, or additional terms.

8. IMPORTANT NOTICE REGARDING AOS/VS INTERNALS SERIES (ORDER #1865 & #1875)
Customer understands that information and material presented in the AOS/VS Internals Series documents may be specific to a particular revision of the product. Consequently user programs or systems based on this information and material may be revision–locked and may not function properly with prior or future revisions of the product. Therefore, Data General makes no representations as to the utility of this information and material beyond the current revision level which is the subject of the manual. Any use thereof by you or your company is at your own risk. Data General disclaims any liability arising from any such use and I and my company (Customer) hold Data General completely harmless therefrom.

Programming with Transport Layer Interface (TLI) for AViiON® Systems

069–000482–01

Cut here and insert in binder spine pocket

DataGeneral
Data General Corporation, Westboro, Massachusetts 01580